09

November 12-18, 2024

Graphs, Searching Algorithms, MST Algorithms

Announcements

- Project 3 due **Tuesday, November 12th at 11:59pm.**
- Project 4 will be available **Thursday, November 14th** and due **Monday, December 9th at 11:59pm.**
- Final Exam on **Friday, December 13th**
  - 8am-10am, in-person, rooms TBA
- Lab 8 AG + Quiz due **Monday**, **November 18th at 11:59pm**.
- Lab 9 written problem due in lab by **Monday**, **November 18th.**
- Lab 9 Quiz due by **Monday**, **November 25th at 11:59pm.**
  - No AG!
- Keep working hard, we're almost there!

Agenda

- Planning your education
- Lab 8 handwritten solution
- Overview of Graphs
- Minimum Spanning Trees
    - Prim's and Kruskal's Algorithms
- Handwritten Problem

Planning a CS Educational Career

CS and Departmental Info

- Computer Science has four main specializations:
  - Systems
  - Machine Learning
  - Theory
  - Hardware
- Department has no “official” specializations
- It’s recommended to explore classes across all specializations
- More common to specialize in a Master’s degree
- Atlas contains information on any class (topics, sections, evaluations, grade distribution, instructors…)
- Upper Level CS Info Sheet contains short testimonials and advice from previous students

CS Degree Requirements

- 15 credits of Upper Level CS Technical Electives (ULCS)
  - 12 credits from main ULCS list - other 3 can be from extended list
  - Many ULCS classes are available after finishing 281

- 10 credits of Flexible Technical Electives (flex techs)
  - ULCS plus any of the following courses

- 8 credits of Major Design Experience (MDE/Capstone)
  - 4 credits from MDE class found here
    - Note: you do **not** receive credit for taking more than one MDE class
  - 2 credits from EECS 496: Professionalism (CS-LSA doesn’t have this requirement)
  - 2 credits from TC 497 (CS) or TC 496 (CE) (CS-LSA doesn’t have this requirement)

- Computer Science CoE program guide has comprehensive information

- Computer Engineering program guide (slightly different from above)

Main ULCS Classes

Systems / Infrastructure (low-level software that supports other software)
Core prerequisites: 281, 370

- 482* Operating Systems - Design a kernel. Multi-threaded programming
- 484 Databases - Design, creation, and query of huge databases. SQL
- 489 Networking - Protocols of networks. Internet focused
- 491* Distributed Systems - High performance, fault tolerance, and implementation
- 483* Compilers - Theoretical exploration. Projects combine to make a compiler

Machine Learning / Artificial Intelligence (perceiving environment and taking intelligent actions).
Core prerequisites: 281, linear algebra

- 442 Computer Vision - 2D/3D computer vision. Image processing
- 445 Machine Learning - Theory and implementation of modern ML algorithms
- 492 Artificial Intelligence - Core AI concepts. Computational agents
- 487 Natural Language Processing - Theory and practical implementation of NLP

* = very popular class ⇒ huge waitlist

Main ULCS Classes

Theory (mathematics/little coding)
Core prerequisites: 376, mathematical maturity

- 475 Cryptography - Attack models, definition of security, reductions
- 477 Algorithms - Efficient algs, analyzing performance, efficient data structures
- 390 Programming Paradigms - Imperative, functional, logic programming along with language semantics and implementation
- 490 Programming Languages - Building languages from both language and mathematical principles

Hardware (implementing physical systems). Core prerequisites: 270, 370

- 373 Intro to Embedded Systems - How to make microcontrollers do things (simplified)
- 427 VLSI Design - Design techniques, rule checking, logic, and circuit simulation
- 470* Computer Architecture - Implement a processor
- 478 Logic Circuit Synthesis and Optimization - CAD development of logic circuits

* = very popular class ⇒ huge waitlist

Main ULCS Classes + EECS 201

Other cool stuff (concepts from several fields, often highly applied)

- 201 Sharpen essential skills at the command line and beyond (Git, Make, Regex, Bash, etc)
- 388* Computer Security - Software, host system, and network security
- 485* Web Systems - develop web applications/internet scale distributed systems
- 481 Software Engineering - Pragmatic approach to software in industry
- 493 User Interfaces - Make easy-to-use system user interfaces
- 471 GPU Programming - Parallel computing and development for GPUs
- 479 Quantum Computing - Impact/limitation of quantum computing. Write quantum programs and test on real quantum hardware.

EECS 498: special topics classes (vary every semester - list may not be accurate)

- Formal Verification - Mathematical verification that a specification matches code
- Election Cybersecurity - Analyze election infrastructure and vulnerabilities
- Advanced Data Structures - Design and analyze complex structures across new domains

* = very popular class ~ huge waitlist

MDE (Capstone) Classes

MDE Classes are usually high-workload, with more open-ended projects (not autograded)

- 440 System Design of a Search Engine
- 441 Mobile App Development for Entrepreneurs
- 443 CS Honors Thesis Course (only for LSA engineering honors)
- 448 Applied Machine Learning for Modeling Human Behavior
- 449 Conversational Artificial Intelligence
- 467 Autonomous Robotics
- 470 Computer Architecture
- 473 Advanced Embedded Systems
- 494 Computer Game Design and Development
- 495 Software for Access
- 497 Human-Centered Software Design & Development
- 498 Special Topics

Apply to be an IA (next semester)

- Responsibilities include running office hours, running a lab, answering on Ed, writing and grading assignments and exams...
- Applications for Winter 2025 closed **Nov 4** (some classes have specific deadlines)
- $26 per hour (for EECS courses)
- 10 hours per week
- Application typically involves short essay questions, and a 5-minute video of you teaching a concept of your choice
- Some classes interview candidates. Some classes require a certain grade to apply
- You can apply for any class you've taken, or most that you're currently taking
- Application is embedded in an email sent to all students from Karen Liska
- Information Page

Handwritten Problem Review

Handwritten Problem: Background

Let's say the *diameter* of a tree is the maximum number of edges on any path connecting two nodes of the tree. For example, here are three sample trees and their diameters. In each case the longest path is bolded and shown in purple. Note that there can be more than one longest path.

```
           A
         /   \
        B     C
       / \   / \
      D   E F   G
     /   /     / \
    H   I     J   K
         \       /
          L     M
```

Diameter: 8

```
      A
     / \
    B   C
   /   / \
  D   E   F
     / \   \
    G   H   I
       /     \
      J       K
```

Diameter: 6

```
  A
   \
    C
   /
  E
 / \
G   H
   /
  J
```

Diameter: 4

Handwritten Problem

Consider the following Node definition of a binary tree:

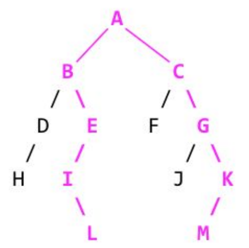
```
class BinaryTreeNode {
public:
    BinaryTreeNode* left;
    BinaryTreeNode* right;
    int value;
    BinaryTreeNode(int n)
     : value(n), left(nullptr),
       right(nullptr) {}
};
```

Your task: Implement the function diameter that computes the diameter of a *binary* tree represented by a pointer to an object of BinaryTreeNode class. Assume that nullptr represents an empty tree or a missing child. Do not modify the definition of BinaryTreeNode class, but you may write helper functions.

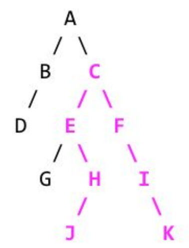
Implement diameter in $O(n^2)$ or better time (it can be done in $O(n)$).

```
int diameter(const BinaryTreeNode* tree) {

}
```



Diameter: 8



Diameter: 6

Diameter: 4

A Quadratic Time Solution

```
int heightOf(const BinaryTreeNode* tree) {
    // number of nodes on longest path leaf-to-root (edges is 1 less than this)
    if (tree == nullptr) {
        return 0;
    } else {
        return max(heightOf(tree->left), heightOf(tree->right)) + 1;
    }
}

int diameter(const BinaryTreeNode* tree) {
    if (tree == nullptr) {
        return 0;
    } else {
        // the diameter exists in left/right subtree:
        int childrenDiameters = max(diameter(tree->left), diameter(tree->right));
        // the diameter is a path through this node (each node has one edge up):
        int nodeDiameter = heightOf(tree->left) + heightOf(tree->right);
        return max(childrenDiameters, nodeDiameter);
    }
}
```

$O(n^2)$

Minor Optimizations

You only need to check the diameter of the subtree with greater height!

- If the longest path doesn't go through the node, it can't occur in the shorter of the two subtrees
- However, the worst-case is still $O(n^2)$

```
int diameter(const BinaryTreeNode* tree)  {
    if (tree == nullptr)  {
        return 0;
    }
    int left_height = heightOf(tree->left);
    int right_height = heightOf(tree->right);
    int diam_taller = left_height >= right_height ? diameter(tree->left) : diameter(tree->right);
    return max(left_height + right_height, diam_taller);
}
```

A Linear Time Solution

```
struct Desc { int height; int diam; };
Desc helper(const BinaryTreeNode* tree) {
    if (tree == nullptr) {
        return Desc{/*height*/ 0, /*diam*/ 0};
    }
    Desc left  = helper(tree->left);
    Desc right = helper(tree->right);
    int diam_children = max(left.diam, right.diam); // diameter in left/right subtree
    int diam_self = left.height + right.height; // longest path through current node
    int diam_whole = max(diam_children, diam_self);
    return Desc {
        /*height*/ 1 + max(left.height, right.height),
        /*diam*/   diam_whole
    };
}
```

<-- **We can return a struct if we need to return multiple values from our helper!**

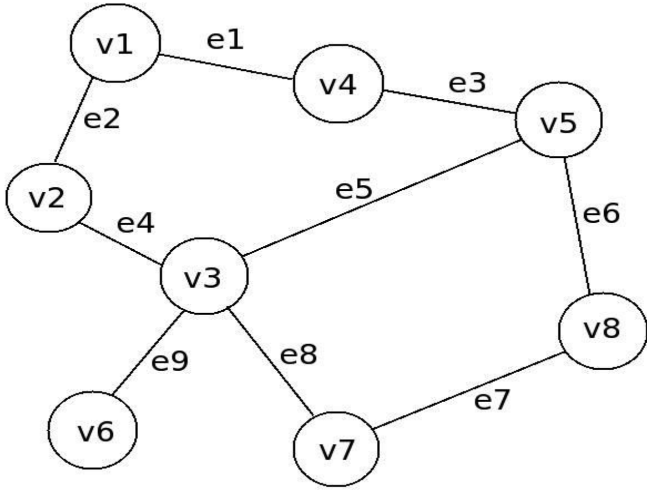
```
int diameter(const BinaryTreeNode* tree) {
    return helper(tree).diam;
}
```

O(n)

Graph Terminology and ADTs

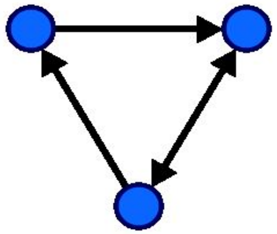
Graph Definition

- A set of vertices and edges that connect them
- Formally:
  - $G = \{V, E\}$ where $V = \{v_1, v_2, ...\}$ and $E = \{ e_1, e_2, ...\}$.
  - Each edge is defined by the pair of vertices that it connects: $e_i = \{v_s, v_t\}$



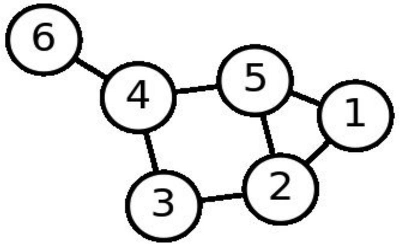
Types of Graphs

- **Directed**: Edges have specified direction(s)
- **Undirected:** All edges are bidirectional
- **Weighted:** Each edge has a weight (e.g. distance, cost, capacity, etc.)



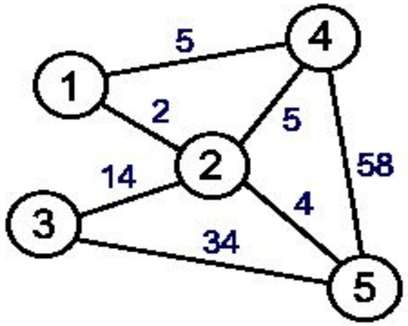
Directed

Twitter follow graph



Undirected

Friendship graph



Undirected and Weighted

Road network

Graph Terminology

- **Simple path:** sequence of edges leading from one vertex to another with no vertex appearing more than once
- **Connected graph**: a path exists between any two vertices
- **Cycle:** a simple path from a vertex to itself with no edge appearing more than once
- **Dense:** number of edges is close to the maximal number of edges i.e. $O(V^2)$
- **Sparse:** graph with few edges (opposite of dense)
- **Cost:**
  - In a weighted graph, the cost is the sum of the weights of all the edges
  - In an unweighted graph, assume the cost of each edge is 1

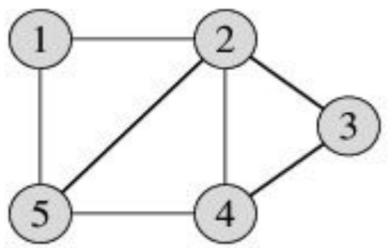
Graph Representations

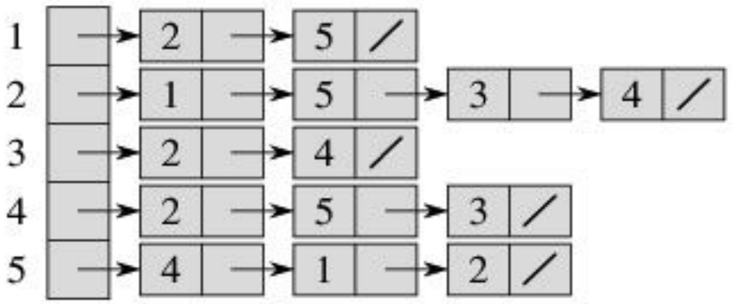
Adjacency List:

- For each vertex, list out all neighboring vertices (i.e. vertices that are connected to it by an edge)

Adjacency Matrix:

- A Boolean matrix which specifies whether an edge exists between each pair of vertices

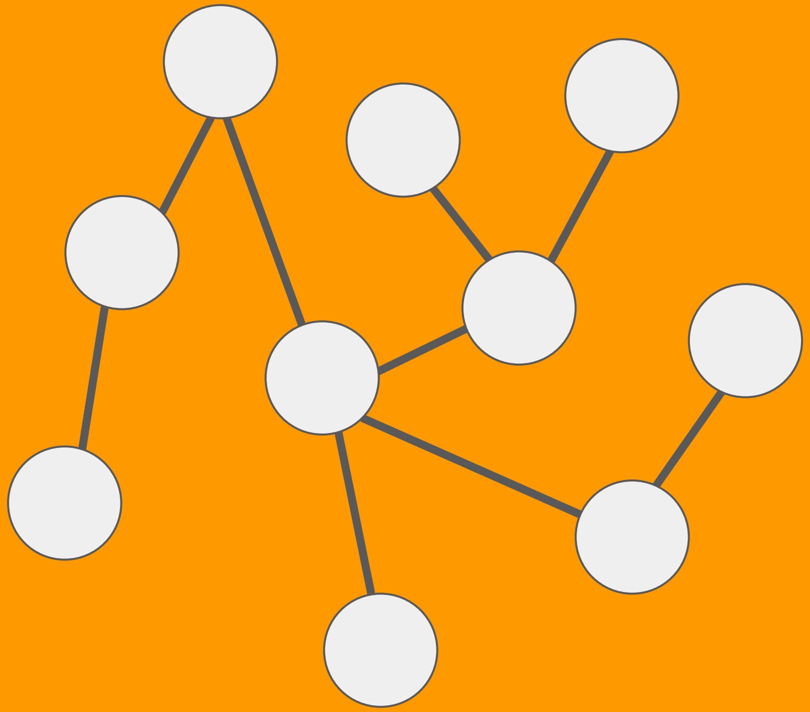




	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0

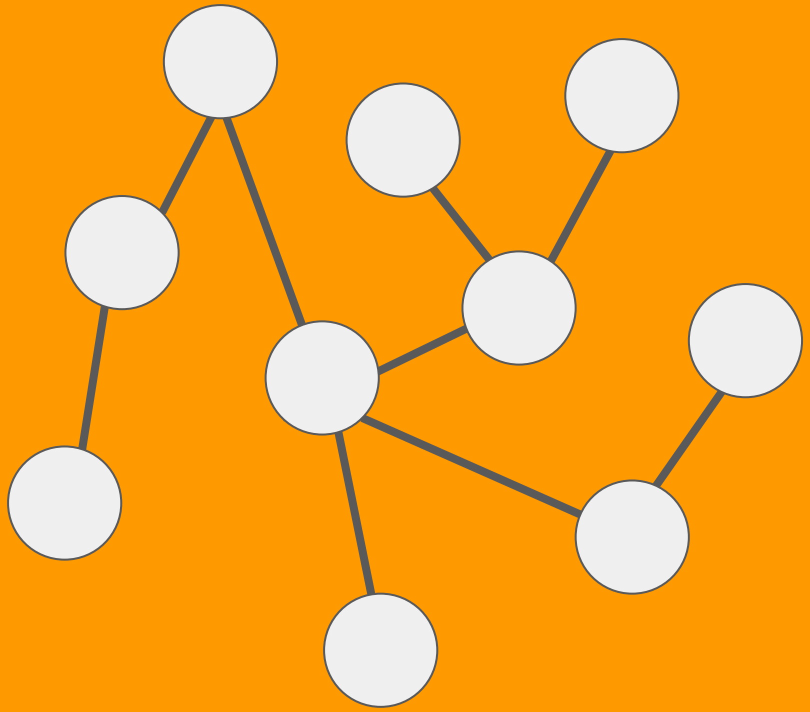
Sparse vs. Dense Graphs

A tree (graph with no cycles)?

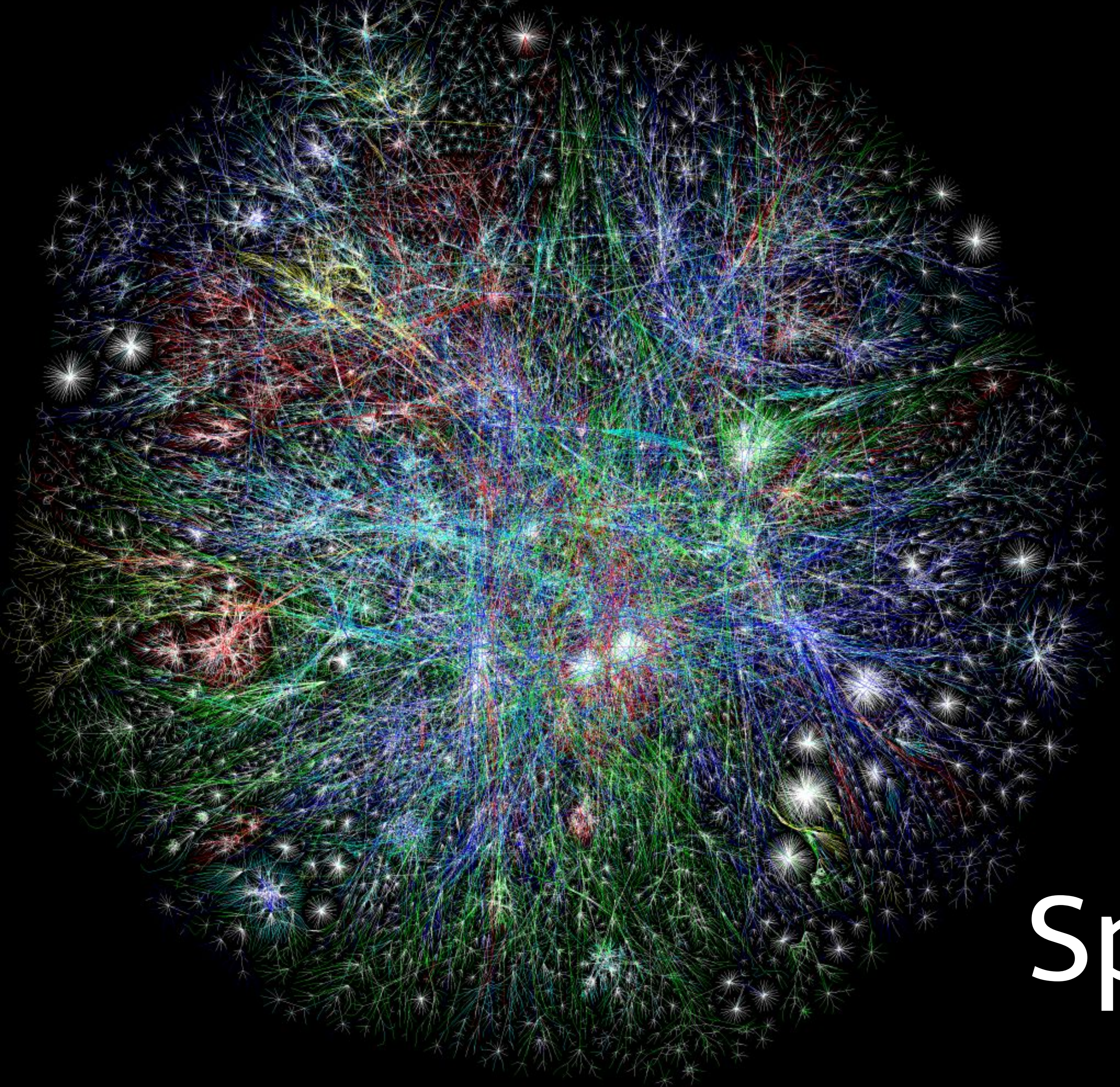


Sparse or Dense?

A tree (graph with no cycles)?

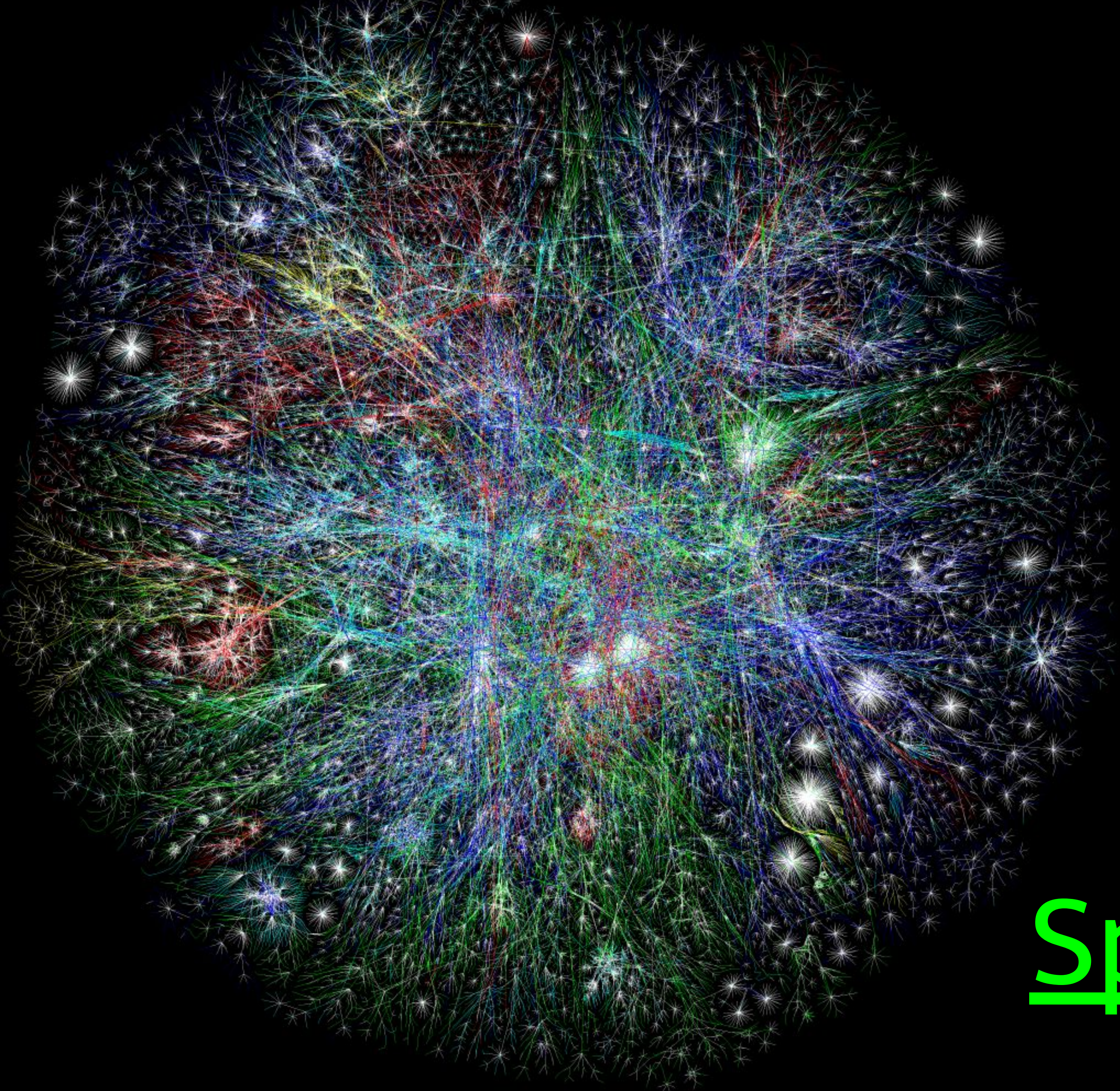


<u>Sparse</u> or Dense?



- G = The internet
- V = web pages
- E = hyperlinks

Sparse or Dense?



- G = The internet
- V = web pages
- E = hyperlinks

<u>Sparse</u> or Dense?

- **Vertices:**
  - **Words in the dictionary**
- **Edges:**
  - **u, v are adjacent iff they start with the same letter**

{apple, alligator} in E

{apple, angry} in E

{dog, doodle} in E

{spoon, sapphire} in E

…

Sparse or Dense?

- **Vertices:**
  - **Words in the dictionary**
- **Edges:**
  - **u, v are adjacent iff they start with the same letter**

{apple, alligator} in E

{apple, angry} in E

{dog, doodle} in E

{spoon, sapphire} in E

…

Sparse or Dense?

Graph Traversal Algorithms

Graph Traversals

Depth-First Search (O(|V| + |E|)):

- Uses a stack or recursion
- Visit the child nodes before visiting the sibling nodes; this allows you to traverse the depth of any particular path before exploring its breadth

Breadth-First Search (O(|V| + |E|)):

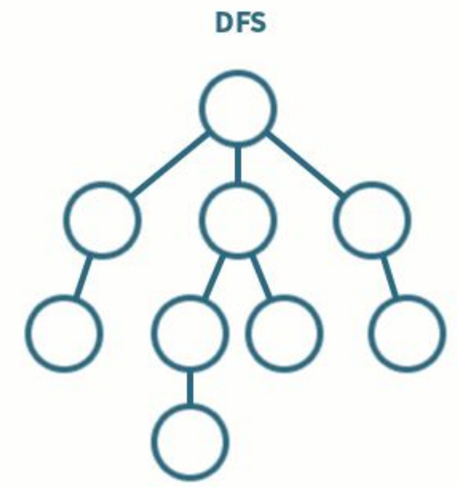
- Uses a queue
- Visit the sibling nodes before visiting the child nodes
- When all edges have the same cost, BFS finds the shortest path between nodes
    - More general: If the cost of the path is non-decreasing function of the depth of the node, then BFS finds the shortest path between nodes

Depth-First Search (DFS) - Recursive

```
def DFS(G, v):
    label v as explored
    for each neighbor w of v with edge e = (v, w):
        if vertex w is not explored:
            if w is goal:
                return w
            r = DFS(G, w)
            if r is not None:
                return r
    return None
```

Lots of variations possible!

Example to draw on

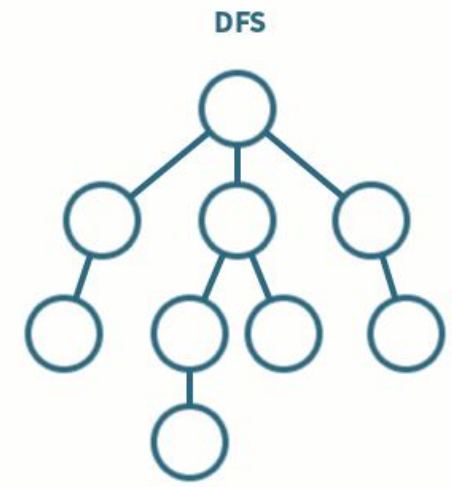


Depth-First Search (DFS) - Iterative

```
def DFS(G, v):
     S = Stack()
     mark v as explored
     add v to S
     while S is not empty:
        t ← S.pop()
        for each neighbor s of t with edge e = (t, s):
              if s is not explored:
                   if s is goal:
                        return Found
                   mark s as explored
                   add s to S
   return NotFound
```

Lots of variations possible!

Example to draw on

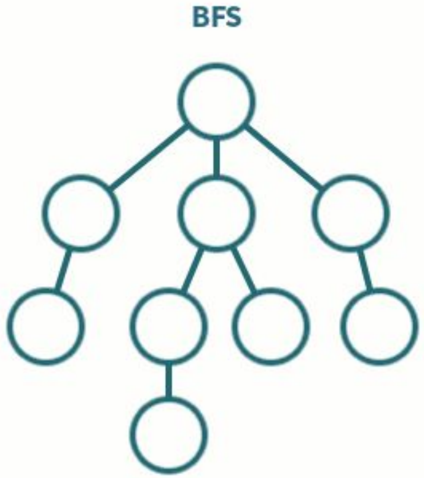


Breadth-First Search (BFS)

Lots of variations possible!

```
def BFS(G, v):
    Q = Queue()
    mark v as explored
    add v to Q
    while Q is not empty:
        t ← Q.pop()
        for each neighbor s of t with edge e = (t, s):
            if s is not explored:
                if s is goal:
                    return Found
                mark s as explored
                add s to Q
  return NotFound
```

Example to draw on

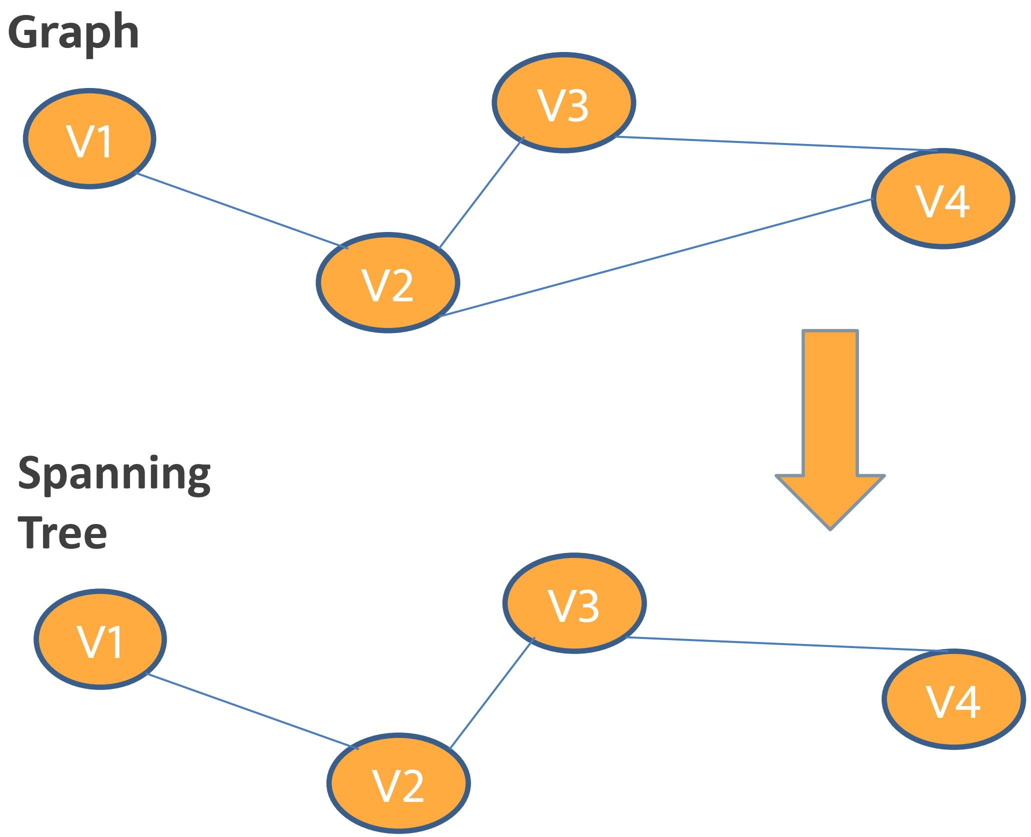


Minimum Spanning Trees

Prim’s and Kruskal’s Algorithms

Spanning Trees

- A subset of a graph G that:
  - Contains all vertices in V
  - Is connected
  - Is acyclic

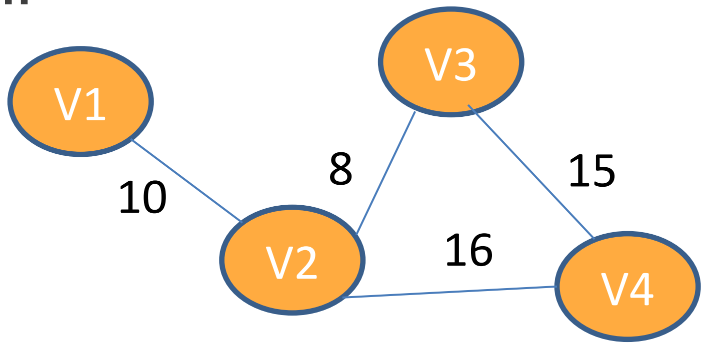


Minimum Spanning Trees (MST)

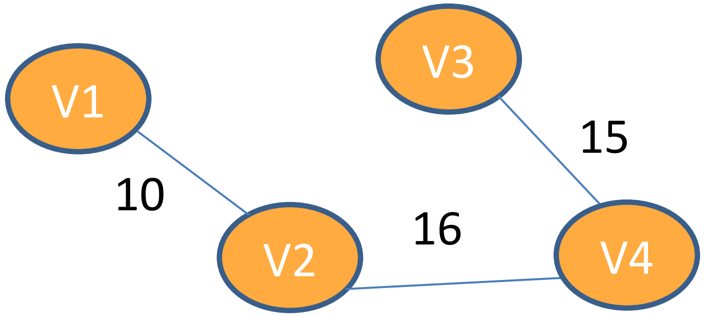
- The spanning tree of graph G that has the lowest total cost
- This means that an MST:
    - is a subgraph of G
    - contains all vertices in V
    - is connected
    - is acyclic
    - contains edges whose weights have a smaller sum than any other spanning tree

MST Example

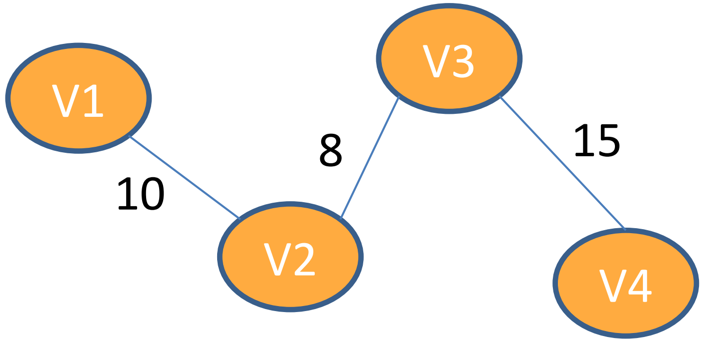
Graph



Spanning Tree, but not MST



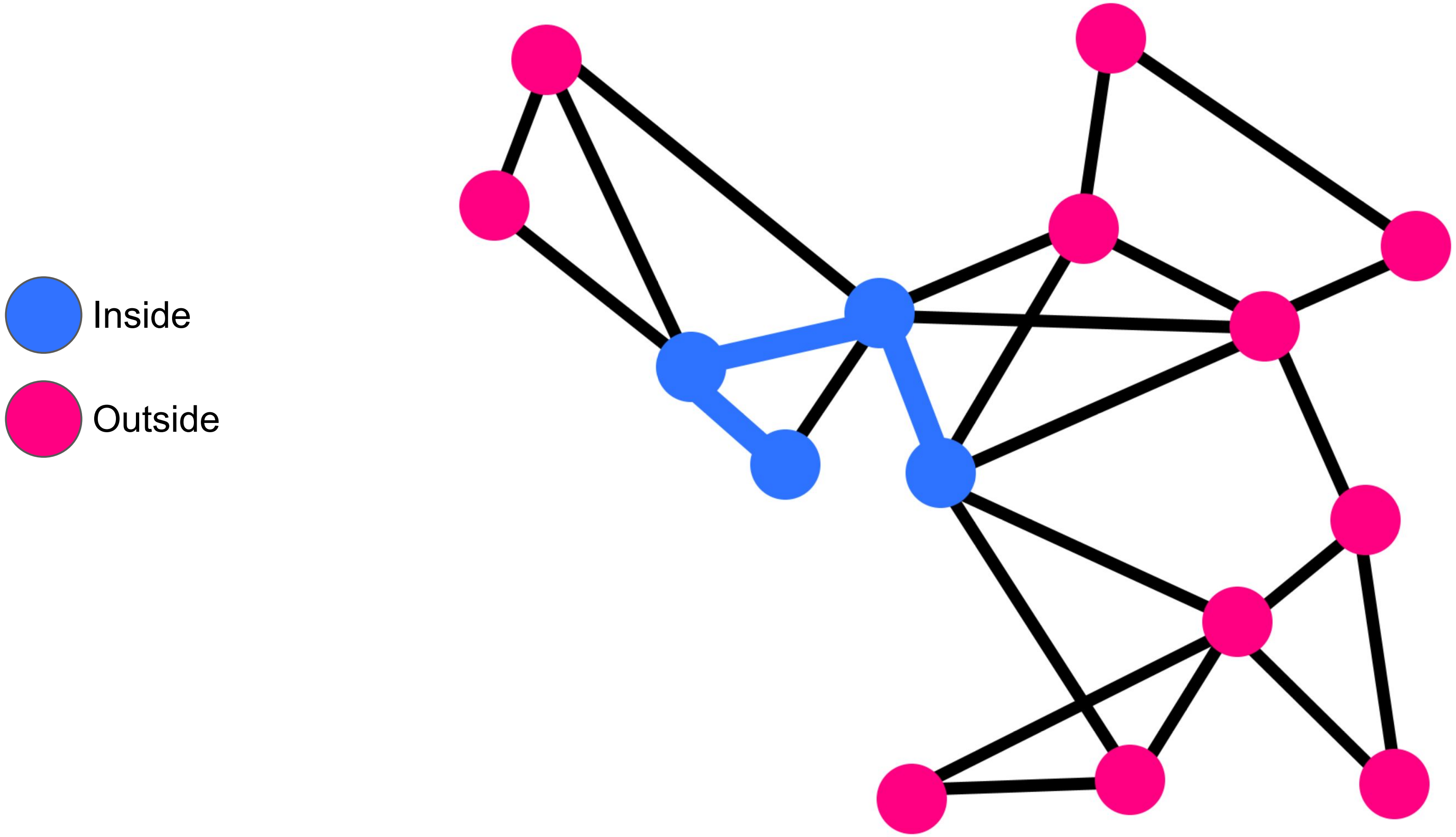
MST



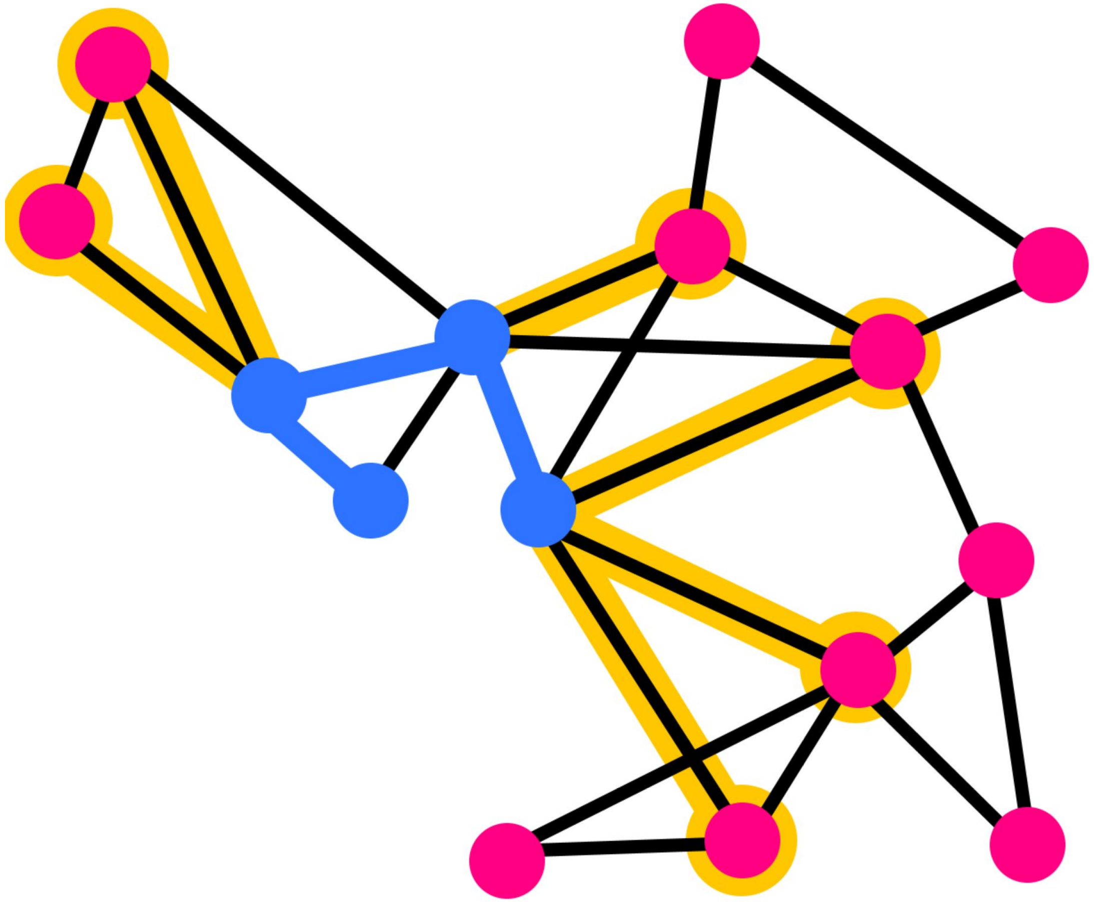
Prim’s Algorithm

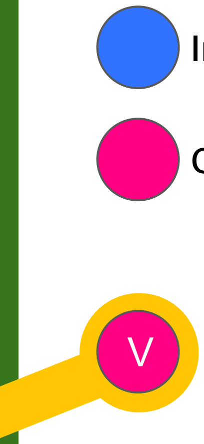
- Greedy Approach
- Separate vertices into two groups:
    - “Innies” are vertices that are present in your partial MST at any point in time
    - “Outies” are the other vertices
- Iteratively add **nearest** outie, converting to an innie
- Best for dense graphs

Prim’s Algorithm



Prim's Algorithm





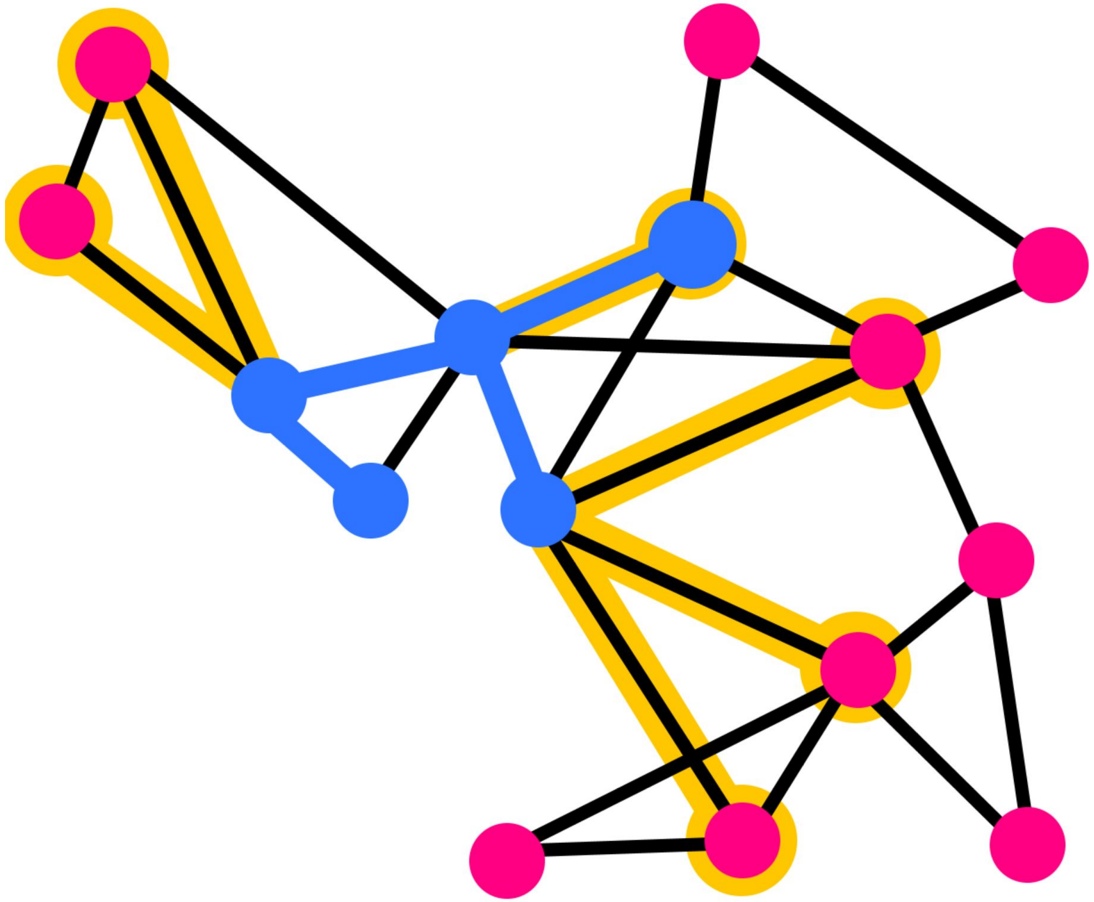
Inside

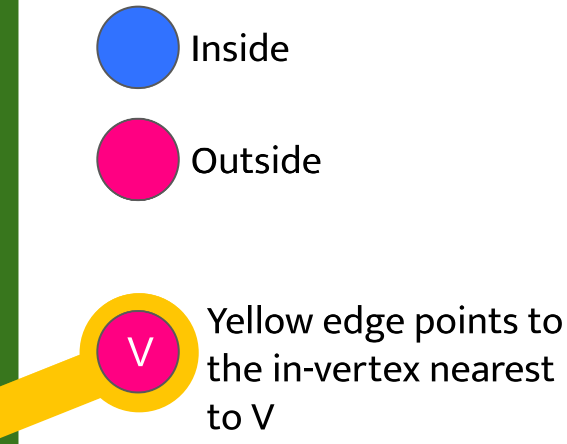
Outside

Yellow edge points to the in-vertex nearest to V

1 Find closest out-vertex **X**

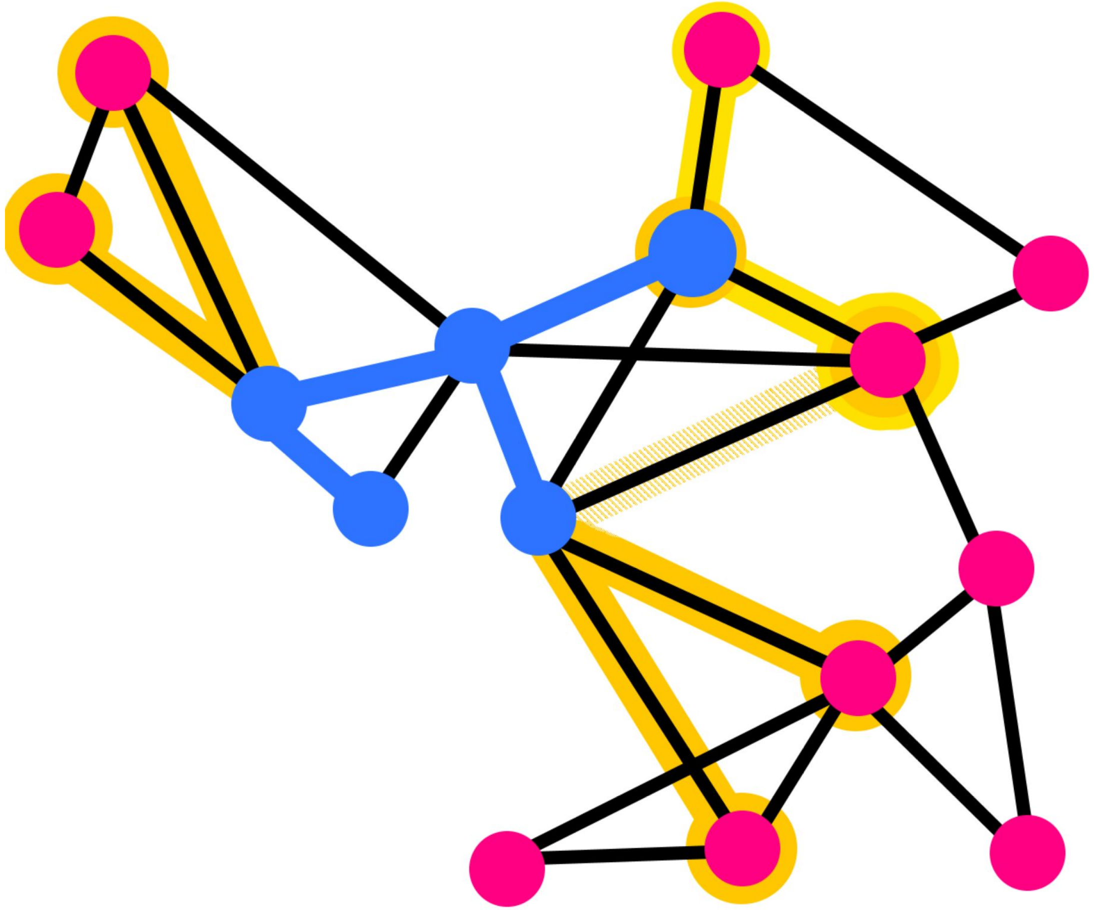
Prim’s Algorithm





1	Find closest out-vertex **X**
2	Mark **X** as “in”

Prim's Algorithm



Inside

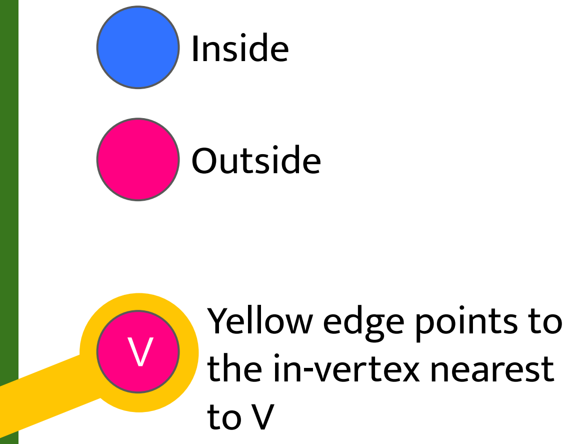
Outside

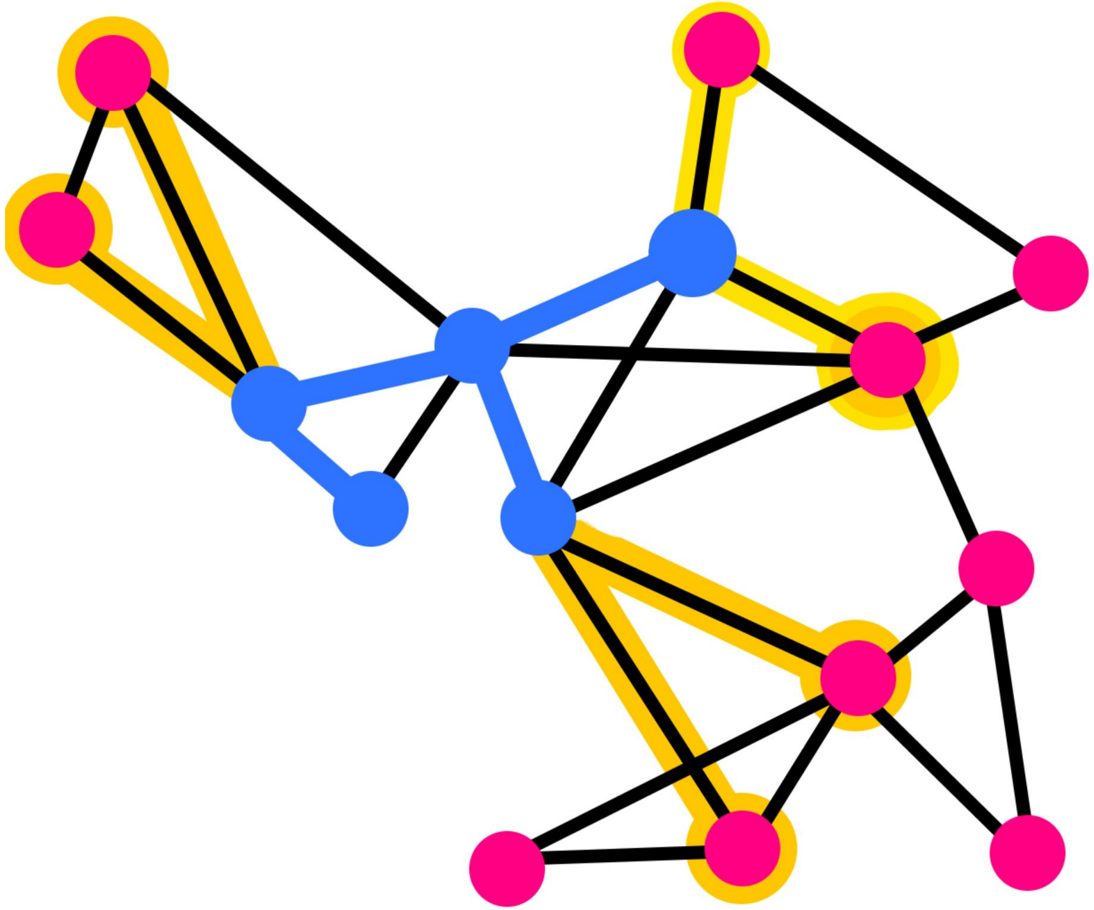
V

Yellow edge points to the in-vertex nearest to V

1	Find closest out-vertex **X**
2	Mark **X** as "in"
3	Update "nearest" for each out-vertex **Y** that is closer to **X** than **Y**'s old nearest.

Prim’s Algorithm





1	Find closest out-vertex **X**
2	Mark **X** as “in”
3	Update “nearest” for each out-vertex **Y** that is closer to **X** than **Y**’s old nearest.
4	Repeat until no out-vertex

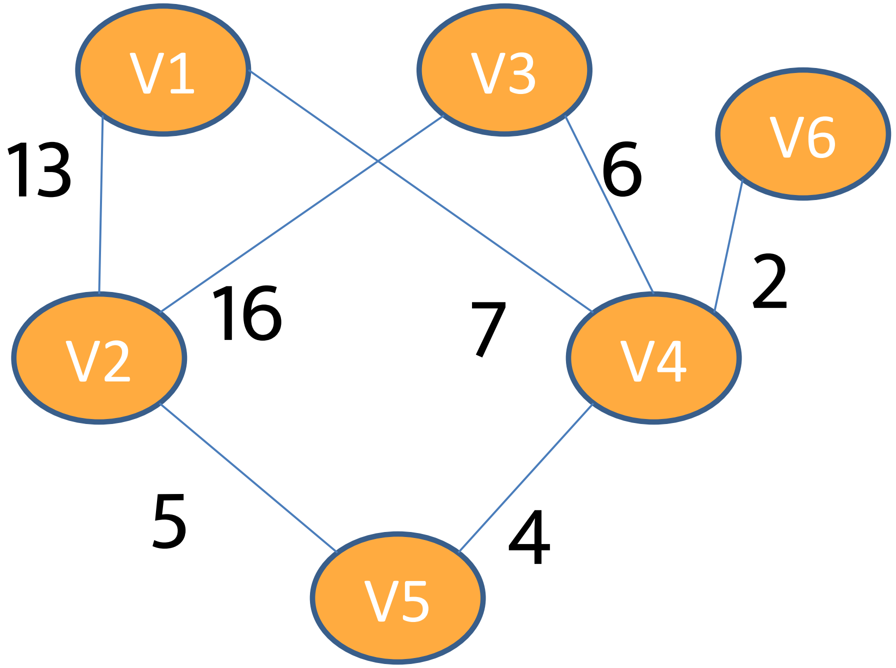
Prim’s Algorithm: Step by Step

- Begin with arbitrary node and mark it as an “innie”
- Until all vertices have been marked as “innies”:
  - Choose the “outie” vertex **X** that is the smallest distance from any “innie”
  - Add **X** to the innie tree; keep track of its “parent” (closest innie), so you know what edge connects it to the tree
  - Update vertices connected to the new “innie” with new weights and a new parent index if this weight is smaller than their previous weight

Example

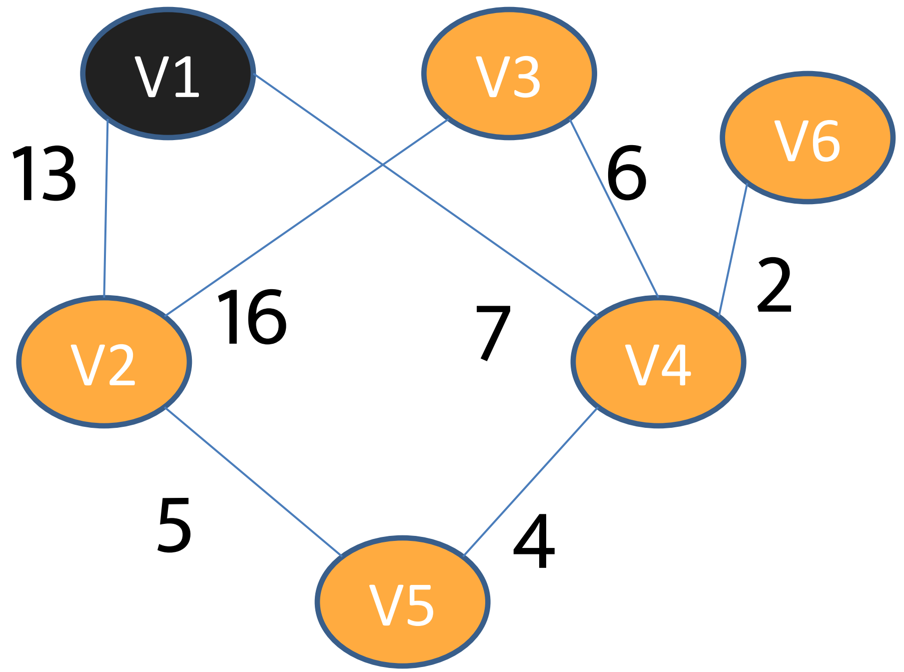
inside MST? — how far from nearest innie? — which innie is closest?

v	k_v	d_v	p_v
$V1$	***F***	0	
$V2$	***F***	∞	
$V3$	***F***	∞	
$V4$	***F***	∞	
$V5$	***F***	∞	
$V6$	***F***	∞	



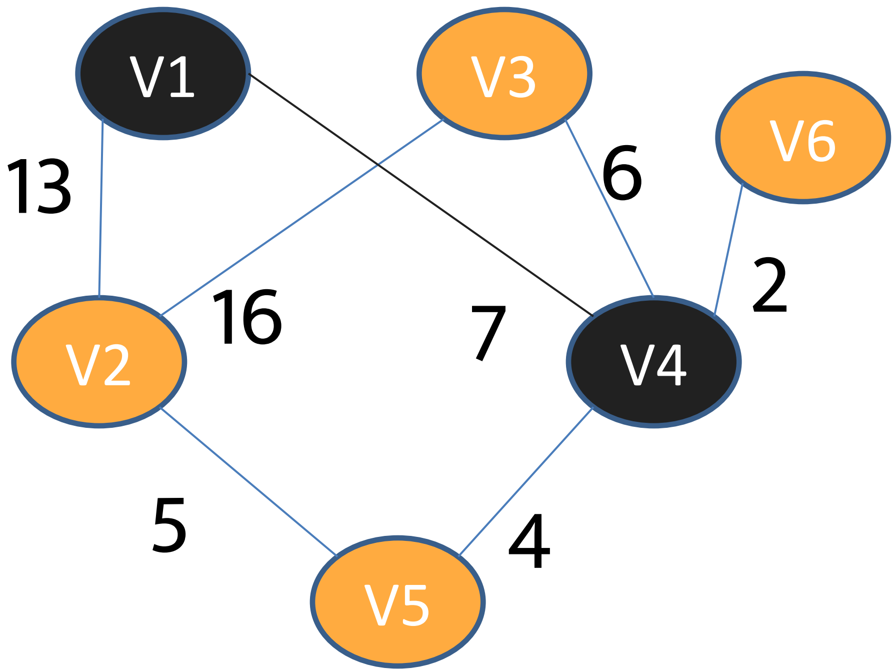
Example

v	k_v	d_v	p_v
	inside MST?	how far from nearest innie?	which innie is closest?
V1	**T**	0	
V2	**F**	13	V1
V3	**F**	∞	
V4	**F**	7	V1
V5	**F**	∞	
V6	**F**	∞	



Example

v	k_v	d_v	p_v
	inside MST?	how far from nearest innie?	which innie is closest?
V1	***T***	0	
V2	***F***	13	*V1*
V3	***F***	6	*V4*
V4	***T***	7	*V1*
V5	***F***	4	*V4*
V6	***F***	2	*V4*



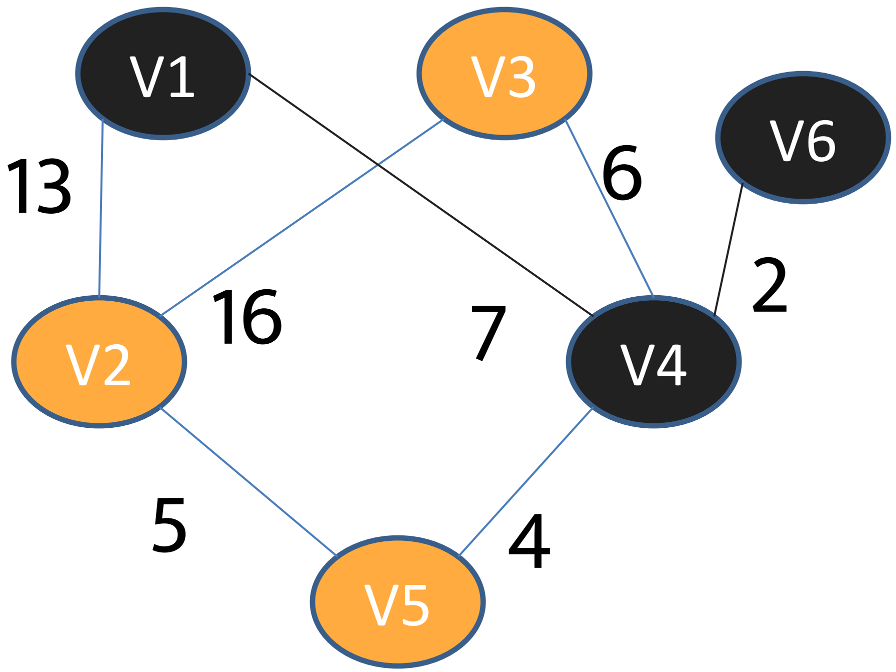
Example

v	k_v	d_v	p_v
V1	**T**	0	
V2	**F**	13	V1
V3	**F**	6	V4
V4	**T**	7	V1
V5	**F**	4	V4
V6	**T**	2	V4

inside MST?

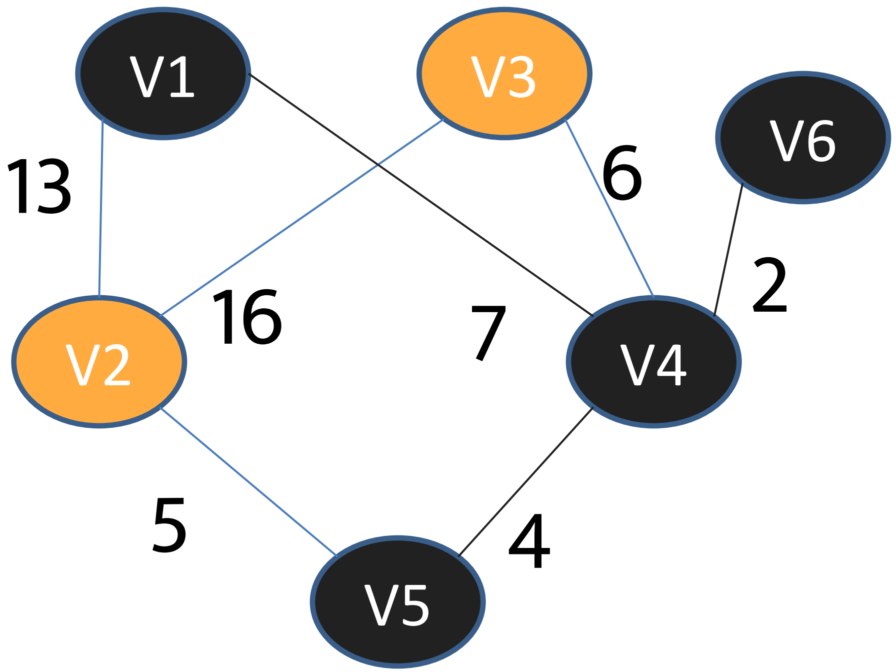
how far from nearest innie?

which innie is closest?



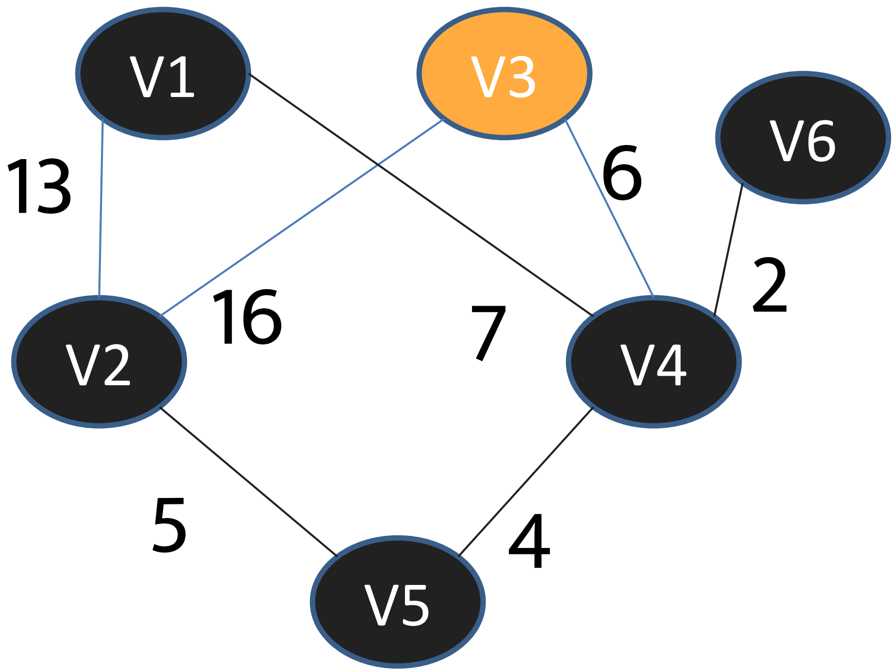
Example

v	k_v	d_v	p_v
	inside MST?	how far from nearest innie?	which innie is closest?
V1	**T**	0	
V2	**F**	5	V5
V3	**F**	6	V4
V4	**T**	7	V1
V5	**T**	4	V4
V6	**T**	2	V4



Example

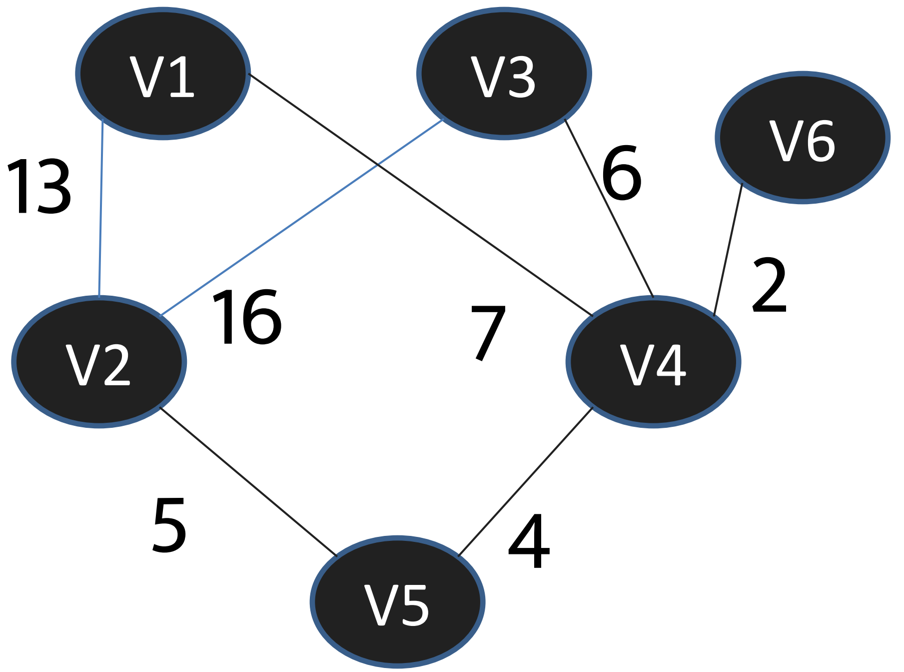
v	k_v (inside MST?)	d_v (how far from nearest innie?)	p_v (which innie is closest?)
V1	***T***	0	
V2	***T***	5	*V5*
V3	***F***	6	*V4*
V4	***T***	7	*V1*
V5	***T***	4	*V4*
V6	***T***	2	*V4*



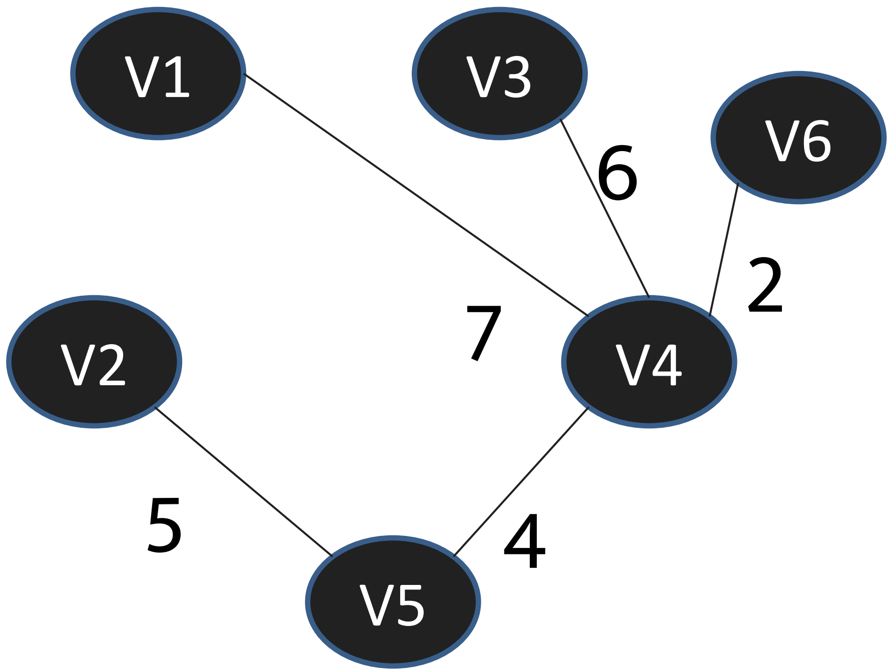
Example

inside MST? | how far from nearest innie? | which innie is closest?

v	k_v	d_v	p_v
V1	T	0	
V2	T	5	V5
V3	T	6	V4
V4	T	7	V1
V5	T	4	V4
V6	T	2	V4



Complete MST:



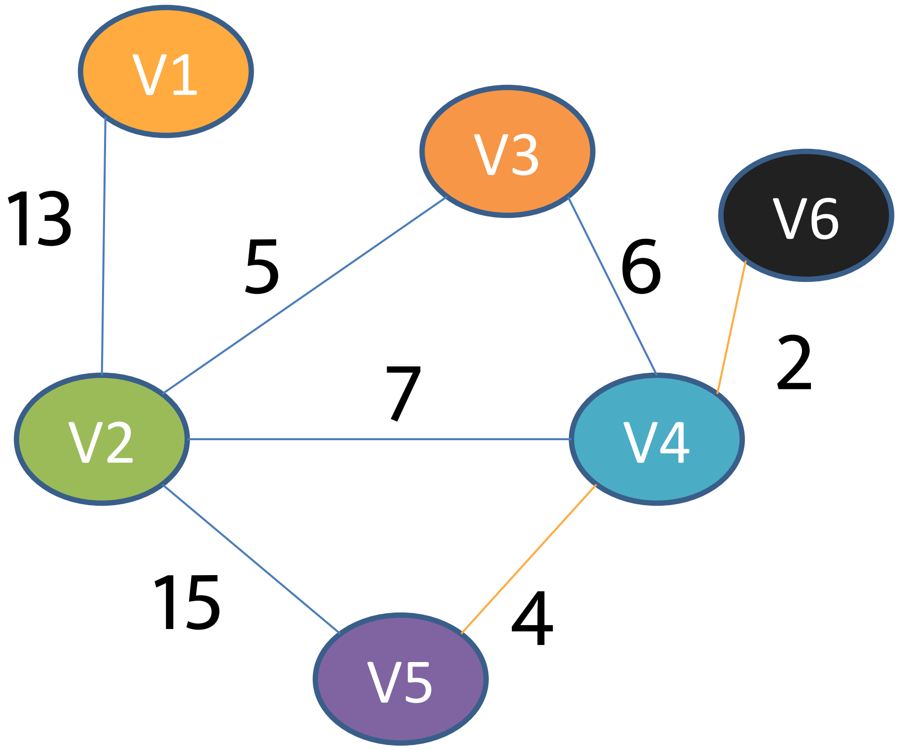
Prim’s Algorithm Complexity

Data Structure Implementation	Time Complexity
Adjacency matrix as graph; **linear search** to find next vertex to add	$O(V^2)$
Adjacency list as graph; use **binary heap** to determine next vertex to add	$O((V + E) \log V)$ since E >= V-1, this is the same as $O(E \log V)$
Adjacency list as graph; use **Fibonacci heap** to determine next vertex to add [Fib heap has O(1) amortized updateElt]	$O(E + V \log V)$

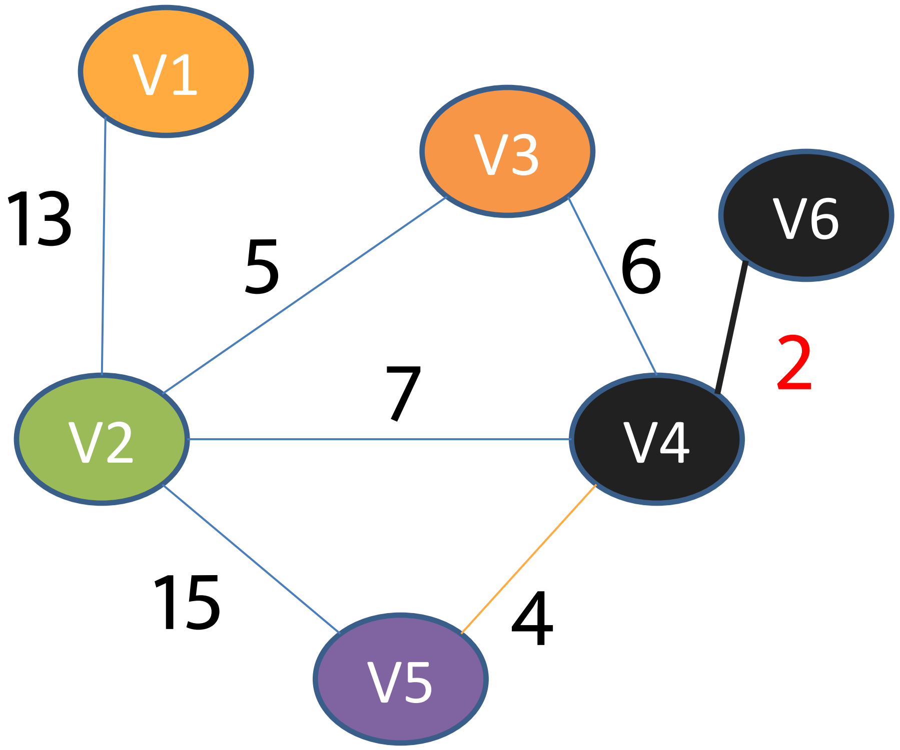
Kruskal’s Algorithm

- Greedy approach
- Maintain a “forest,” or a group of trees /disjoint sets
- Iteratively select cheapest edge in graph
    - If adding the edge forms a cycle, don’t add it
    - Otherwise, add it to the forest
- Continue until all vertices are part of the same set
- Best for sparse graphs

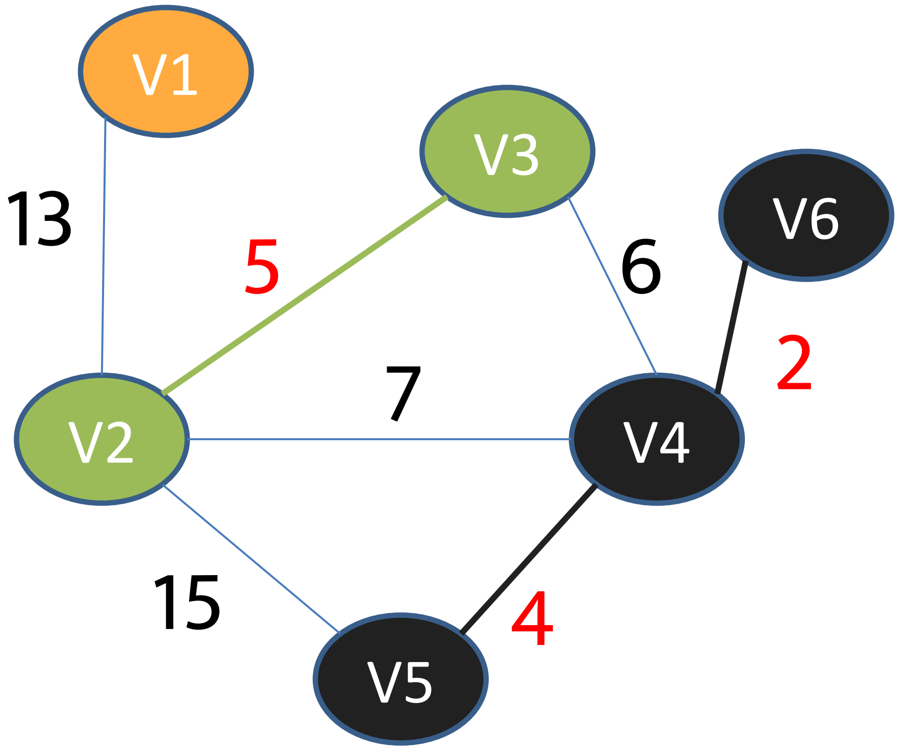
Example



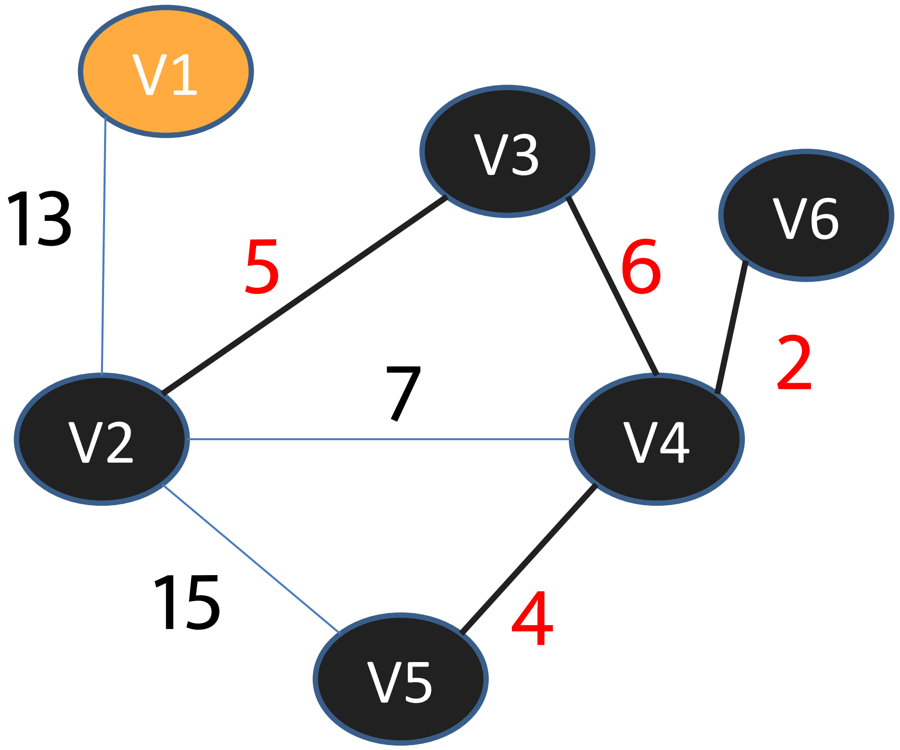
Example



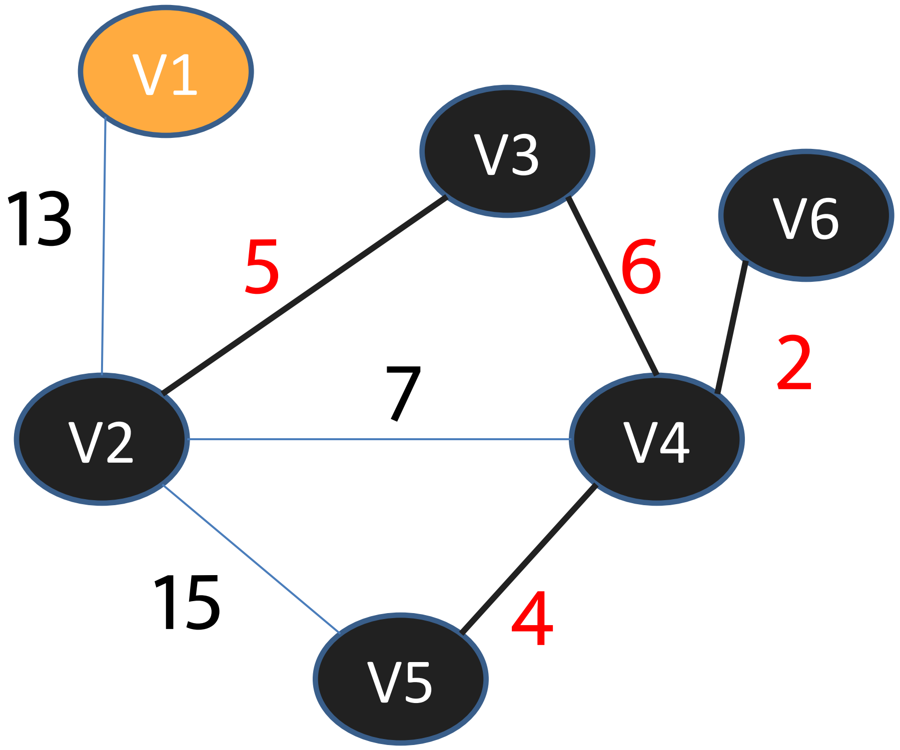
Example



Example

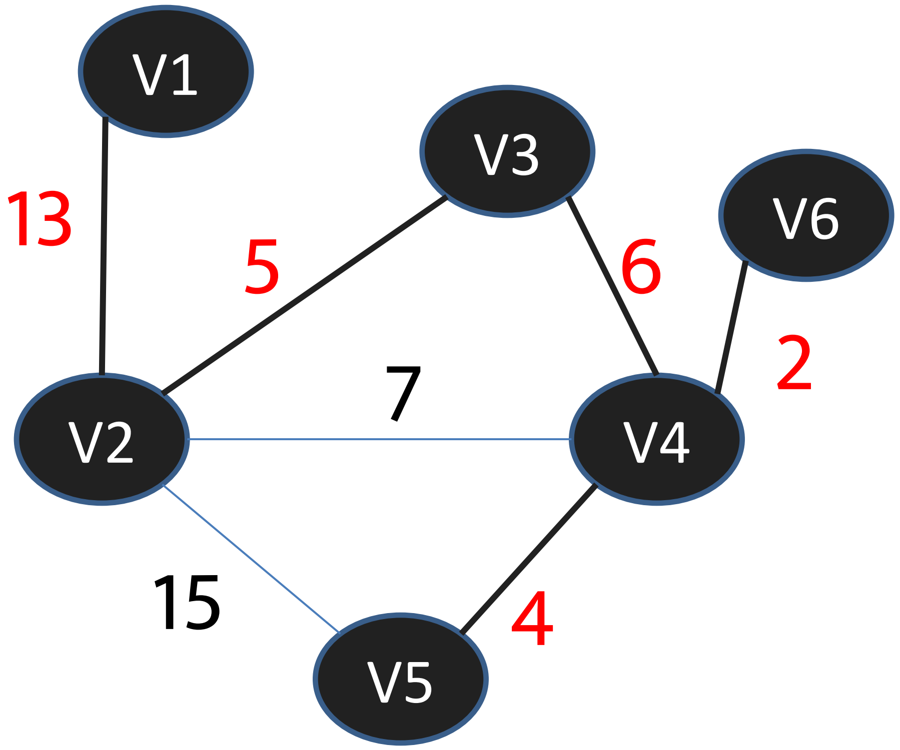


Example



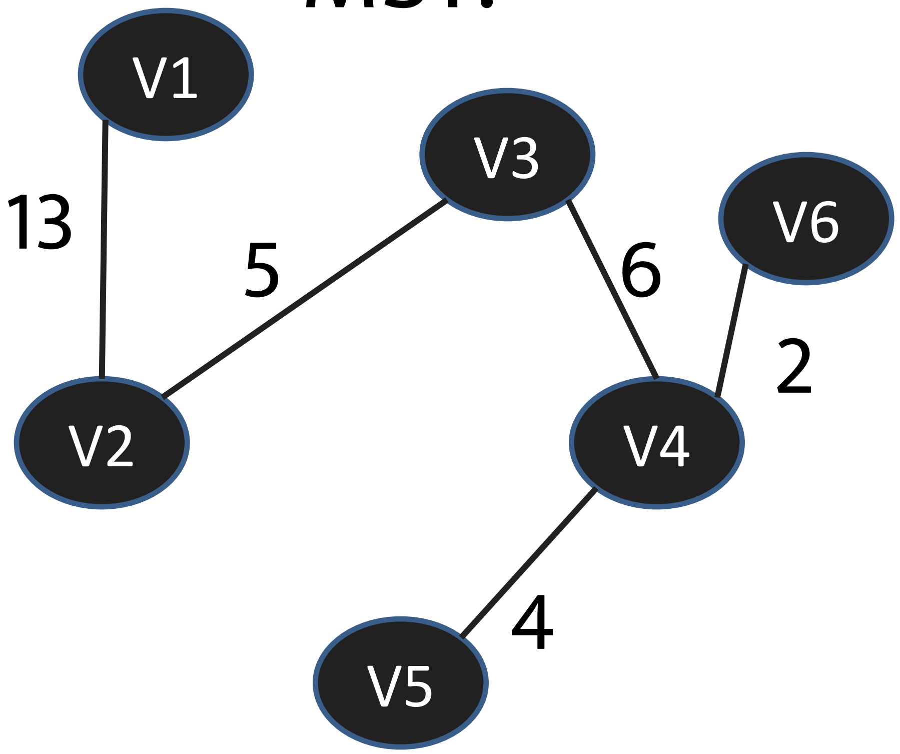
We don’t connect the edge with weight 7 because it would create a cycle

Example



Example

MST:



Kruskal's Algorithm Complexity

1) Sort edges by weight in O(E log E) time

2) Use a **disjoint-set data** structure (**Union-Find**) to keep track of the vertices in each component.

- 2 find operations per edge, up to 1 union
- each operation takes α(V) time, which is almost O(1)

Total time complexity: O(E log E) or O(E log V)

These are the same in the worst case:

$E \leq V^2$ and $O(\log V^2) = O(2 \log V) = O(\log V)$

Faster in practice (but same time complexity) than Prim's with binary heap for very sparse graphs

Modifying a MST

Finding a New MST

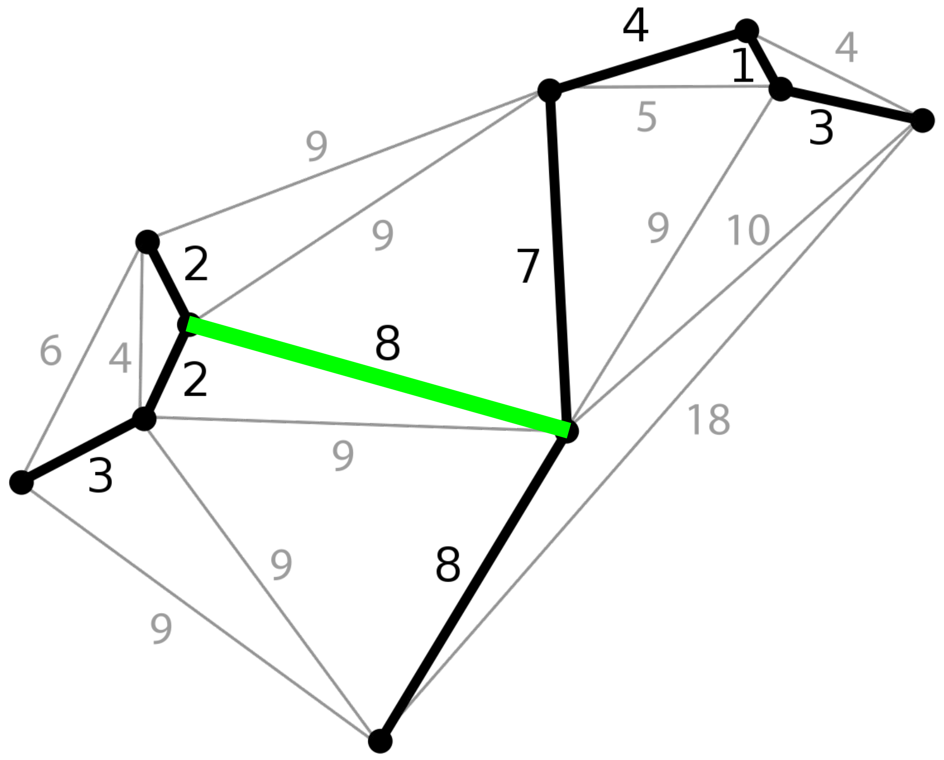
Consider the following cases. How would you find the new MST?

1. Edge is in MST and you are decreasing its weight
2. Edge is not in MST and you are decreasing its weight
3. Edge is not in MST and you are increasing its weight
4. Edge is in MST and you are increasing its weight

Finding a New MST Part 1

Edge is in MST and you are decreasing its weight

(8 -> 3) in the right



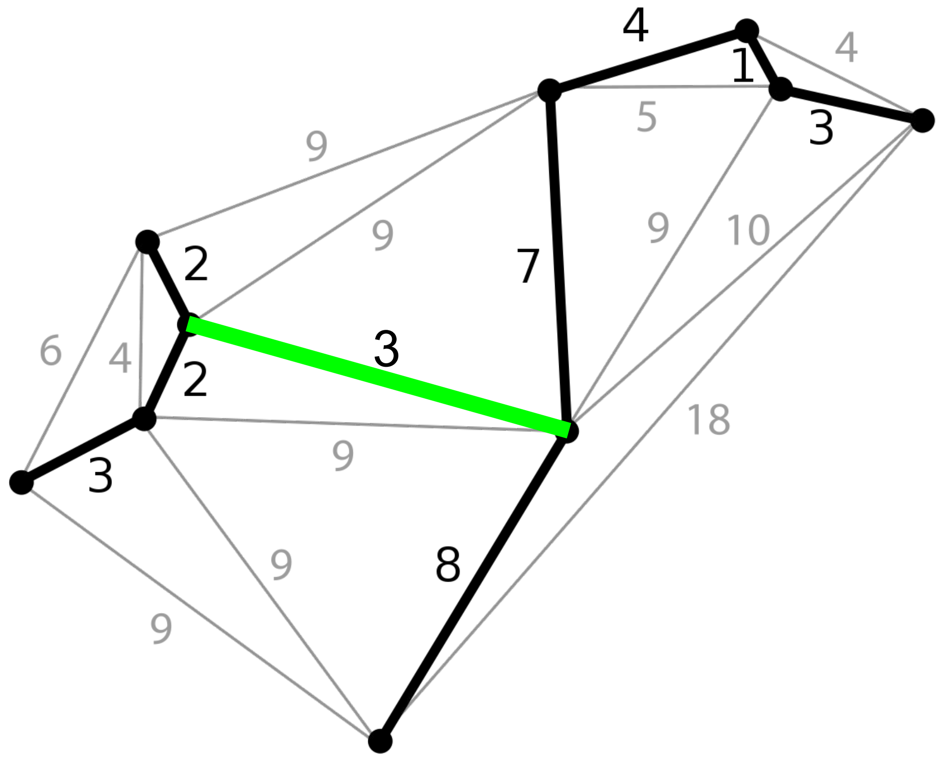
Finding a New MST Part 1

Edge is in MST and you are decreasing its weight

Nothing needs to be done

The MST just got even better!

No other tree can improve more than it did.

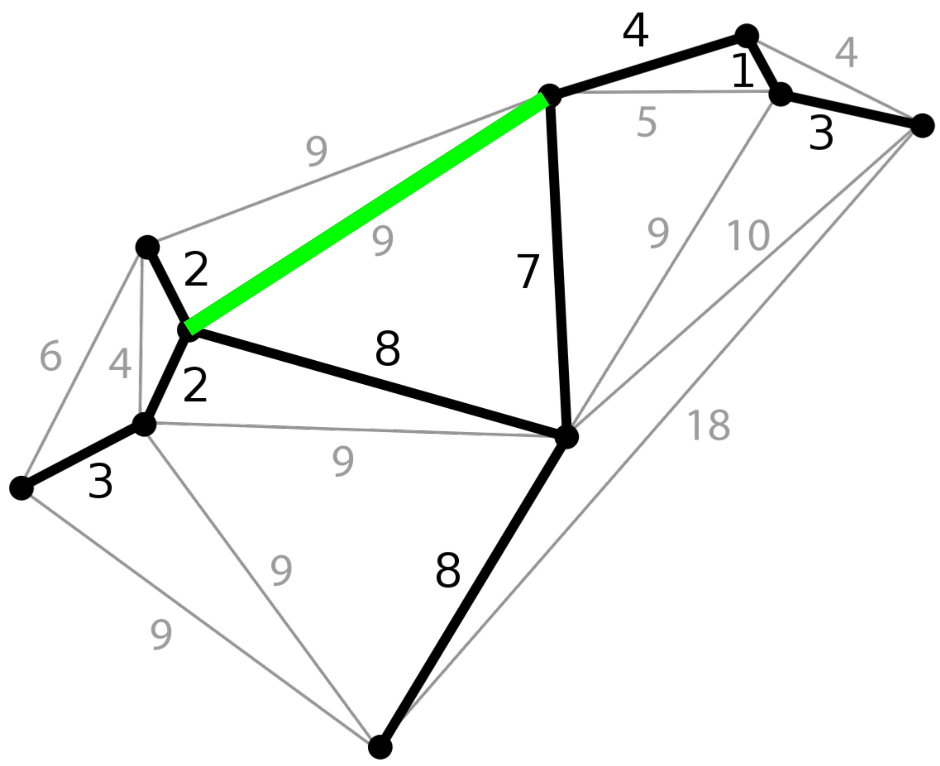


Finding a New MST Part 2

Edge is not in MST and you are decreasing its weight

9 -> 1 in the right.

Can the MST change?



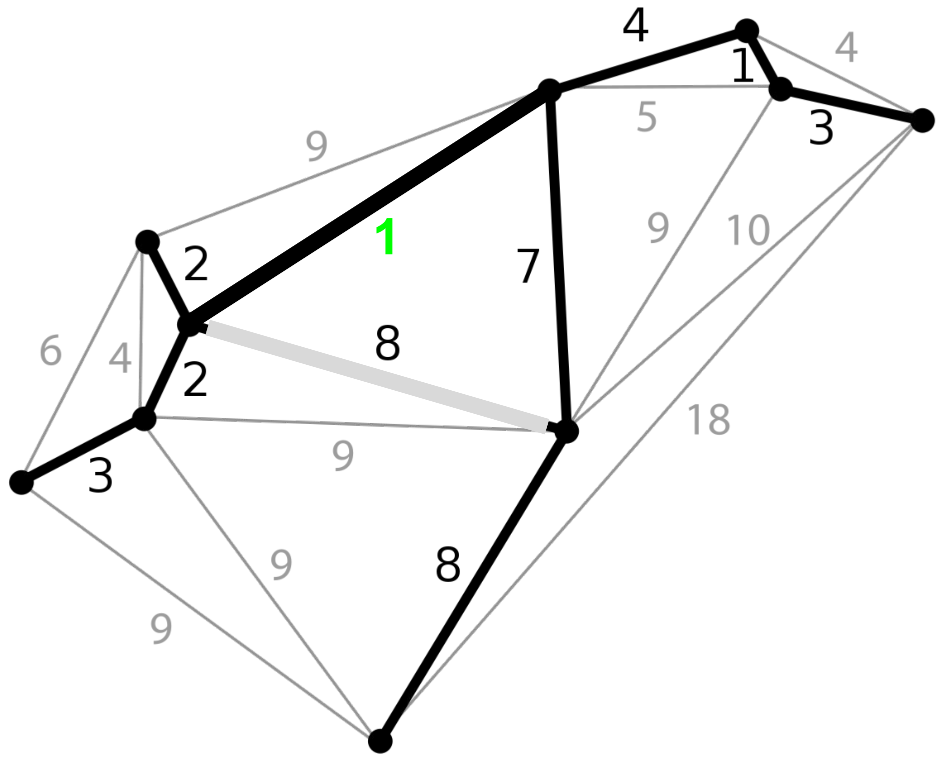
Finding a New MST Part 2

Edge is not in MST and you are decreasing its weight

9 -> 1 in the right.

Can the MST change?

Yes: In this case, the 1 (prev 9) is added to the MST.

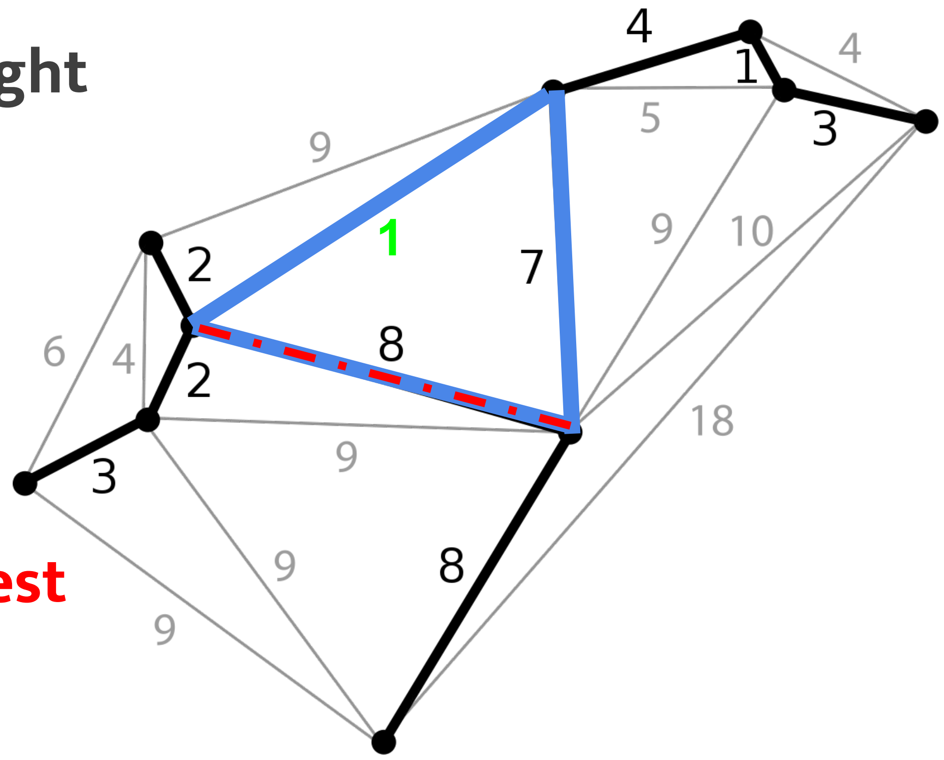


Finding a New MST Part 2

Edge is not in MST and you are decreasing its weight

How can we find the new MST?

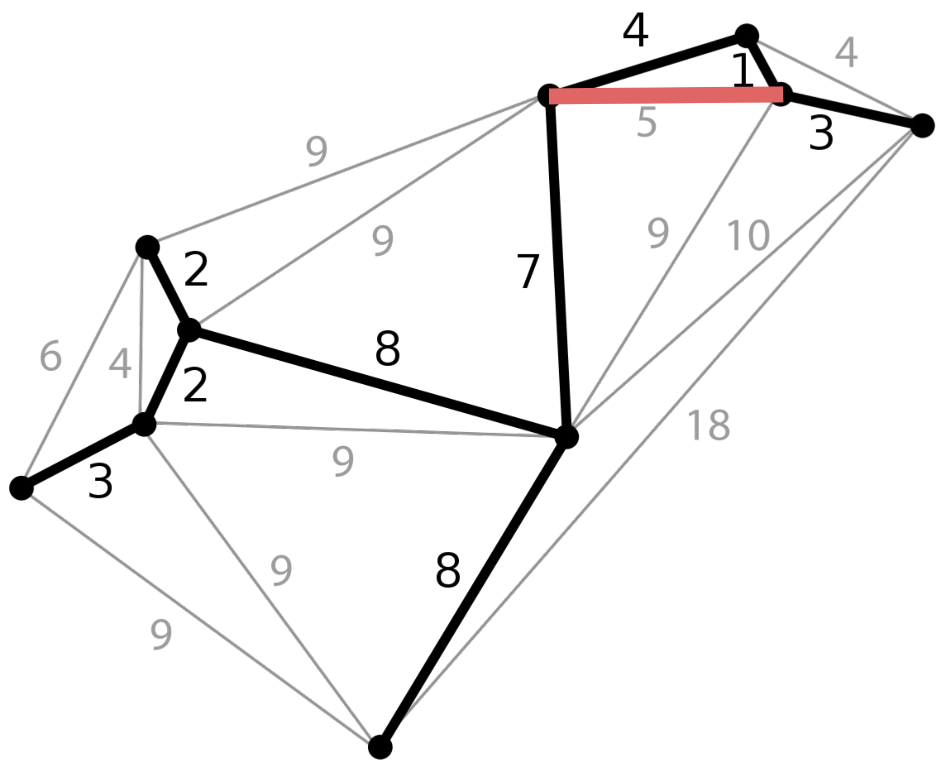
1. Add the edge whose weight was decreased to the MST; now you've got exactly 1 **cycle**
2. Remove the edge in this cycle that has the **highest** weight; you can do this using DFS or BFS
   - Complexity is O(|V|)



Finding a New MST Part 3

Edge is not in MST and you are increasing its weight

- 5->7 in the right

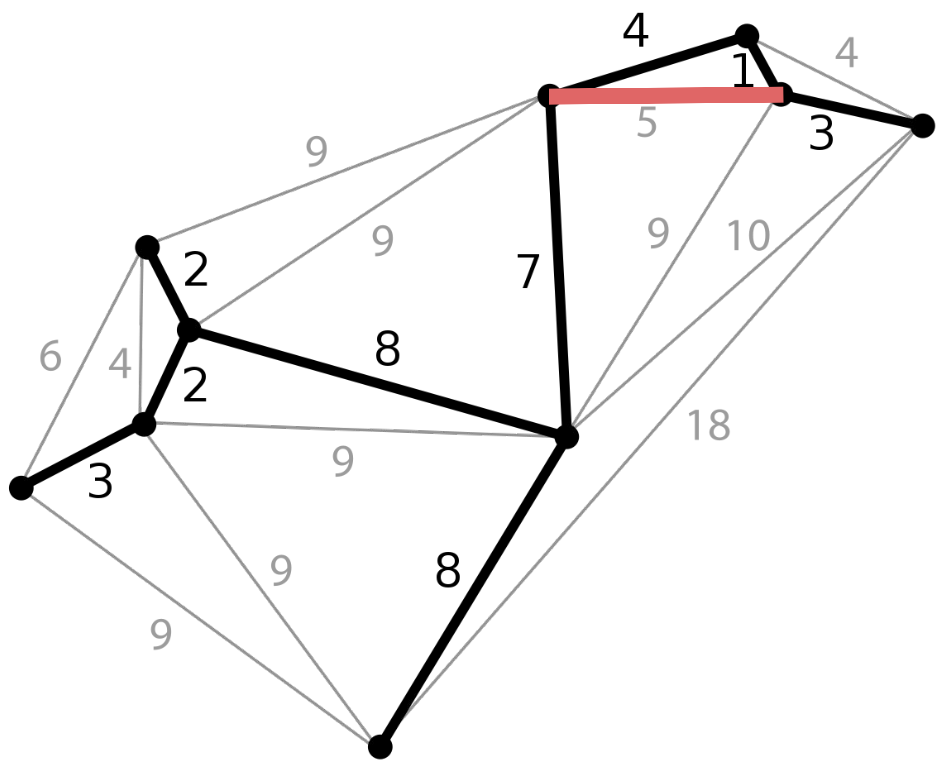


Finding a New MST Part 3

Edge is not in MST and you are increasing its weight

5->7 in the right

Nothing needs to be done; non-MSTs just got worse, but the MST is unchanged.

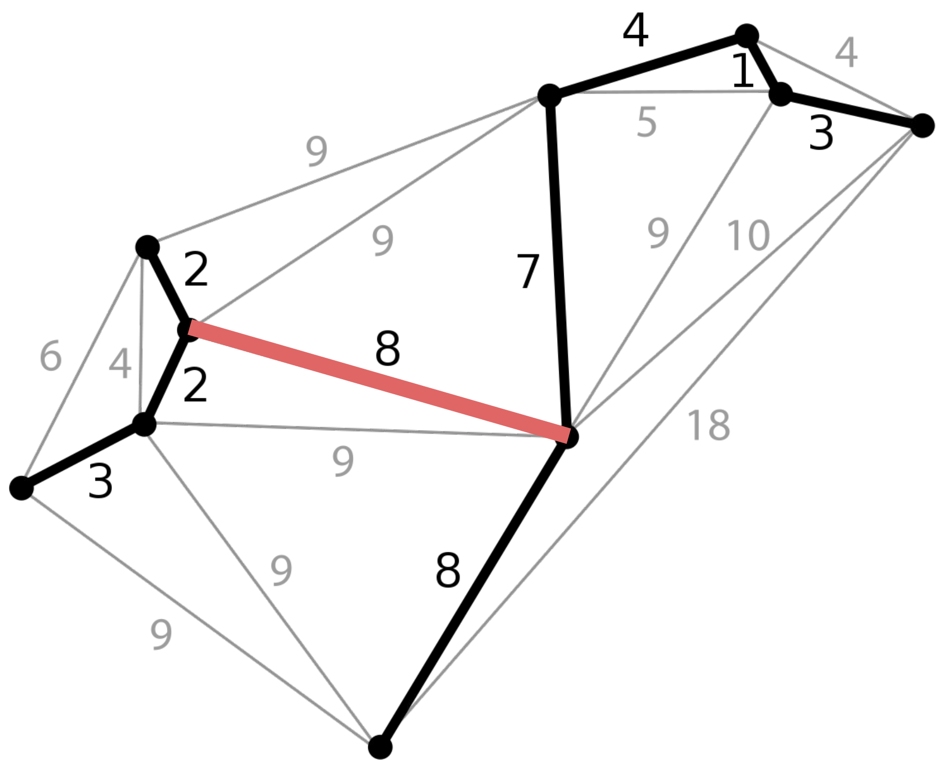


Finding a New MST Part 4

Edge is in MST and you are increasing its weight

Can the MST change?

8 -> 11 in the right?

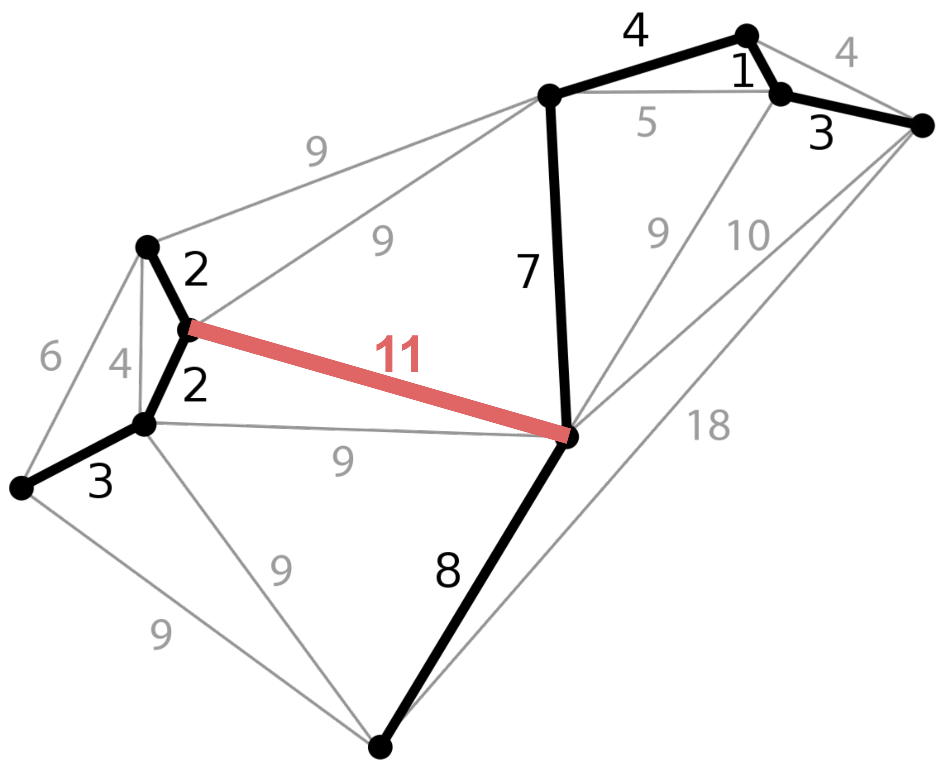


Finding a New MST Part 4

Edge is in MST and you are increasing its weight

Can the MST change?

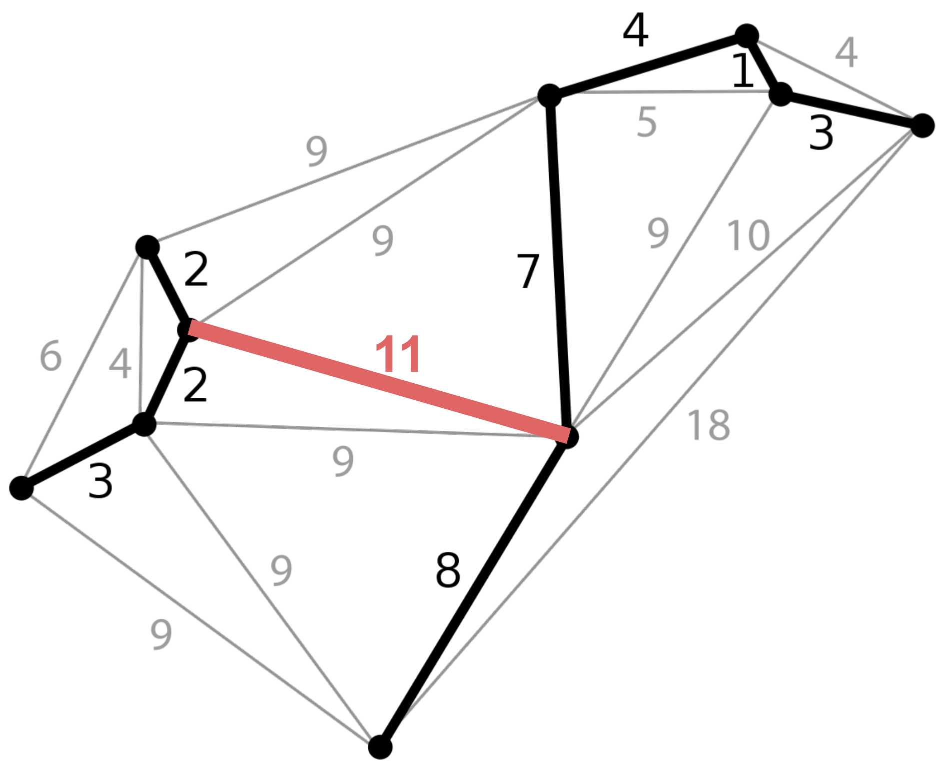
- 8 -> 11 in the right? **Yes.**



Finding a New MST Part 4

Edge is in MST and you are increasing its weight

How can we find the new MST?

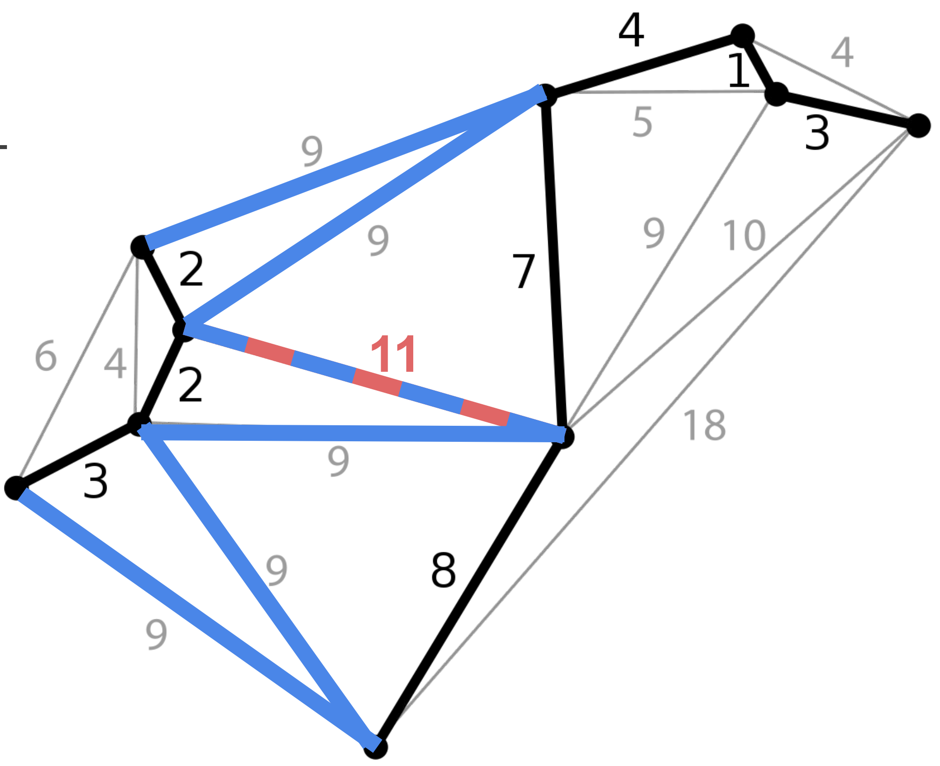


Finding a New MST Part 4

Edge is in MST and you are increasing its weight

How can we find the new MST?

1. Remove the edge whose weight was increased from the MST
   - Now there are two connected components
   - You want to find the lowest weight **edge that connects** these two components



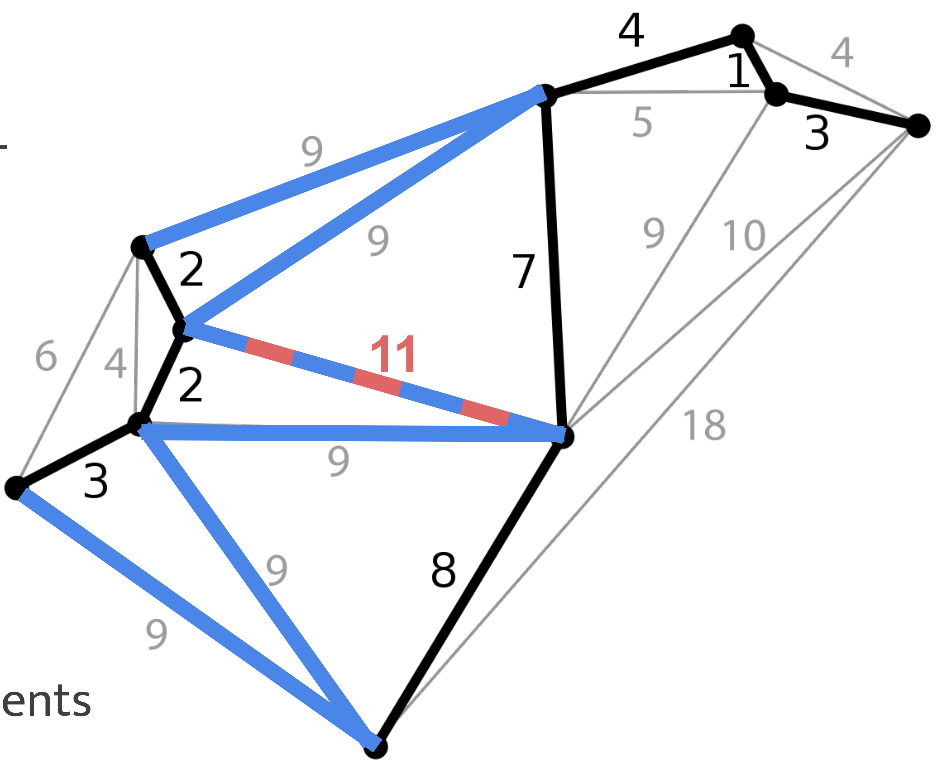
Finding a New MST Part 4

Edge is in MST and you are increasing its weight

How can we find the new MST?

1. Remove the edge whose weight was increased from the MST
    - Now there are two connected components
    - You want to find the lowest weight **edge that connects** these two components

2. You can find the new edge to add in O(E) time
    - Traverse through the components using a BFS or DFS, building hash tables for quick look-ups
    - Find shortest edge whose ends are in opposite components
        - O(1) time per edge for hash table lookup



Handwritten Problem

Handwritten Problem

Given an **undirected** connected graph, check if the graph contains a cycle.

```
bool is_graph_cyclic(const vector<vector<int>>& adj_list);
```

Feel free to use a helper function!

You can assume that if u and v are adjacent, then `adj_list[u]` will contain v and `adj_list[v]` will contain u. Also, u will not appear in `adj_list[u]`.

Complexity: O(V + E) time