CS 401

Minimum Spanning Tree / Midterm review

Xiaorui Sun

Reminder

Homework 2 due 11:59pm today

Midterm exam: Thursday 12:30pm-1:15pm this classroom

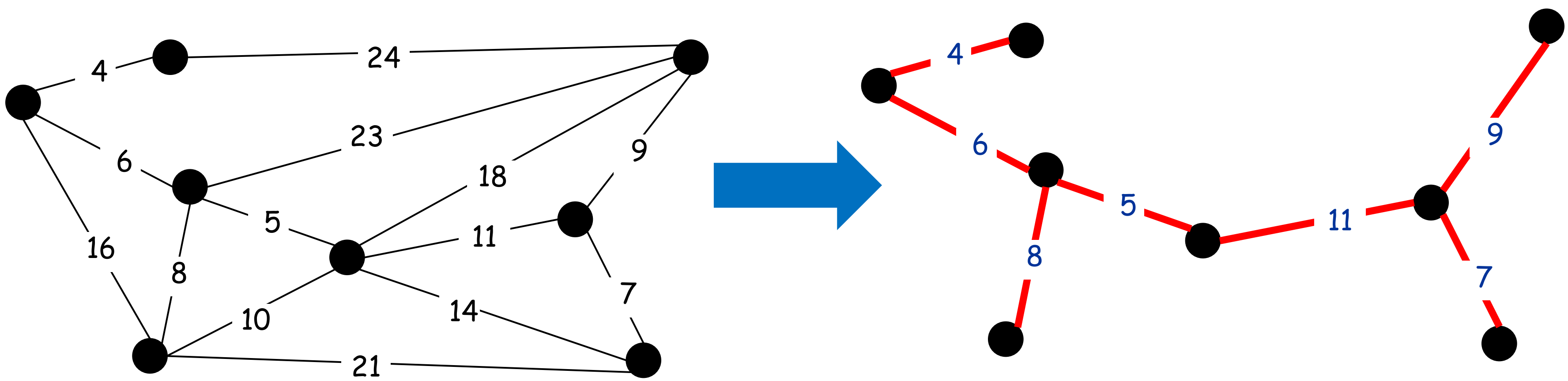
Midterm review later this lecture

Minimum Spanning Tree

Minimum Spanning Tree (MST)

Given a connected undirected graph $G = (V, E)$ with real-valued edge weights $c_e \geq 0$.

An MST T is a spanning tree whose sum of edge weights is minimized.



$G = (V, E)$

$$c(T) = \sum_{e \in T} c_e = 50$$

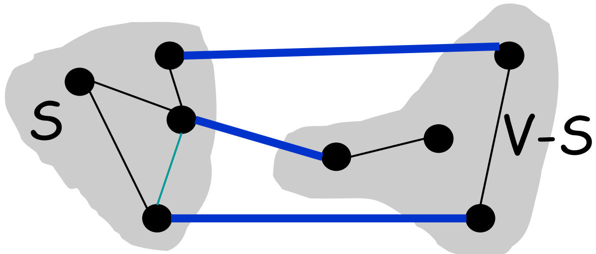
Kruskal's Algorithm [1956]

```
Kruskal(G, c) {
   Sort edges weights so that c_1 ≤ c_2 ≤ ··· ≤ c_m.
   T ← ∅

   for i = 1 to m
      Let (u,v) = e_i
      if (u and v are in same connected component of T) {
         T ← T ∪ {e_i}
   }
   return T
}
```

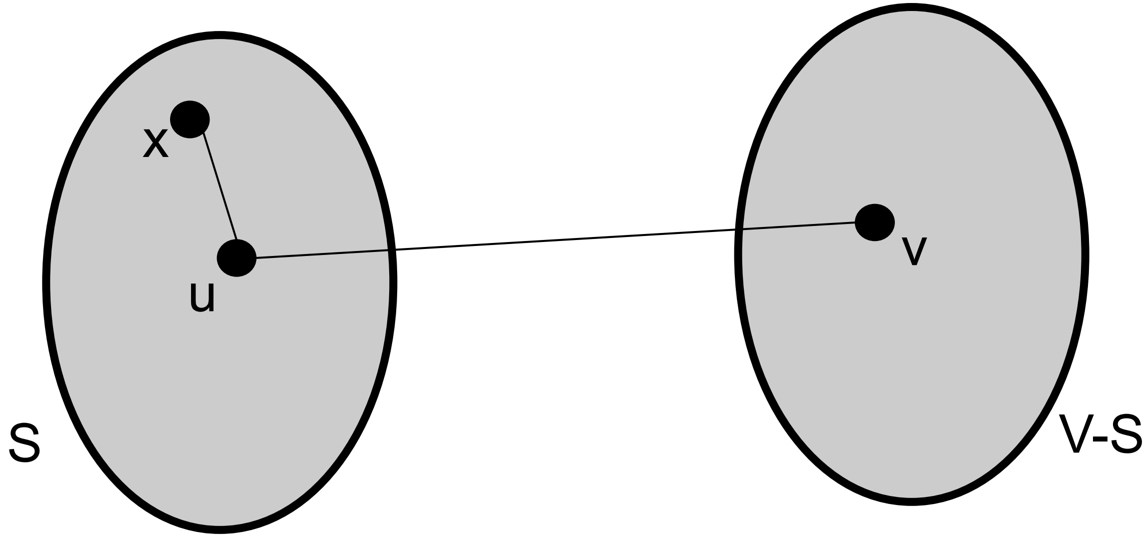
Add edges whenever it does not create cycle.

Cuts



In a graph $G = (V, E)$, a cut is a bipartition of V into disjoint sets $S, V - S$ for some $S \subseteq V$. We denote it by $(S, V - S)$.

An edge $e = \{u, v\}$ is in the cut $(S, V - S)$ if exactly one of u, v is in S.

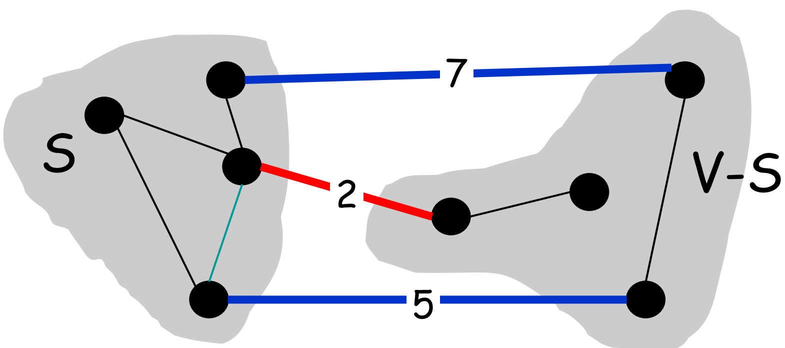


Properties of the OPT

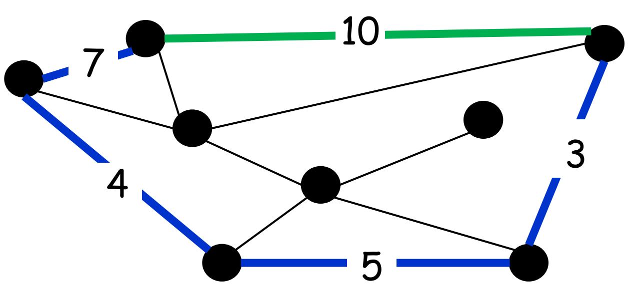
Simplifying assumption: All edge costs c_e are distinct.

Cut property: Let S be any subset of nodes (called a cut), and let e be the min cost edge with exactly one endpoint in S. Then every MST contains e.

Cycle property. Let C be any cycle, and let f be the max cost edge belonging to C. Then no MST contains f.



red edge is in the MST



Green edge is not in the MST

Proof of Correctness (Kruskal)

Consider edges in ascending order of weight.

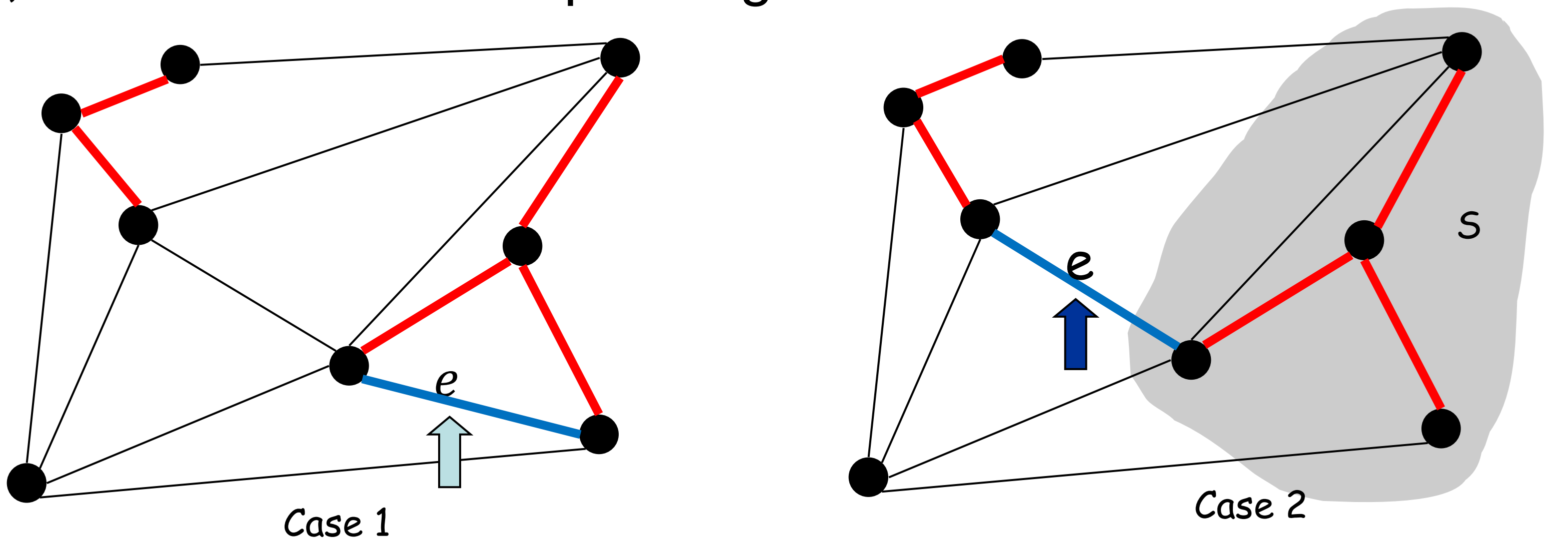
Case 1: adding e to T creates a cycle,

e is the maximum weight edge in that cycle.

cycle property show e is not in any minimum spanning tree.

Case 2: $e = (u, v)$ is the minimum weight edge in the cut S where S is the set of nodes in u's connected component.

So, e is in all minimum spanning tree.



This proves MST is unique if weights are distinct.

Cycle Property: Proof

Simplifying assumption: All edge costs c_e are distinct.

Cycle property: Let C be any cycle in G, and let f be the max cost edge belonging to C. Then the MST T^* does not contain f.

Every connected graph has a spanning tree.
Hence it has at least $n-1$ edges.

Proof. By contradiction

Suppose f belongs to T^*.

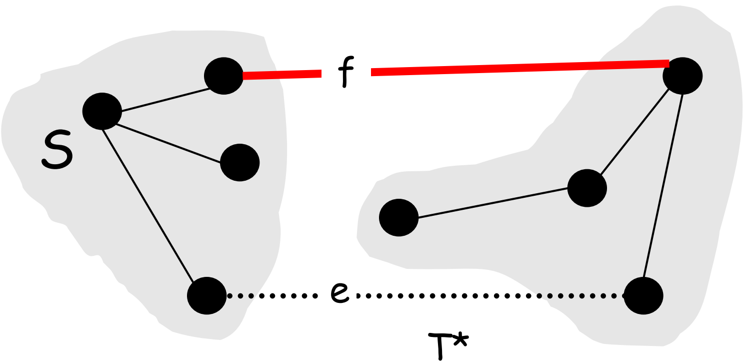
Deleting f from T^* cuts T^* into two connected components.

There exists another edge, say e, that is in the cycle and connects the components.

$T = T^* \cup \{e\} - \{f\}$ is also a spanning tree.

Since $c_e < c_f$, $c(T) < c(T^*)$.

This is a contradiction.



Summary

- Greedy algorithm: 'Best' current partial solution at each step

- Design greedy algorithm:
  - How to order your input
  - Strategy for every step

- Greedy Analysis Strategies
  - Greedy algorithm stays ahead
  - Structural
  - Exchange argument

Midterm Review

Midterm Exam

Midterm exam Feb 29 (Thursday) 12:30pm-1:15pm

- Location: TBH 180F
- Closed textbook exam
- You may use a sheet with notes on both sides, but not textbook and any other paper materials
- You may use a calculator, but not any device with transmitting functions, especially ones that can access the wireless or the Internet

Midterm Exam

True or false

- Only answer true or false, no justification

Short answer

- Answer questions, no justification

Algorithm design

- Graph algorithm
- Greedy algorithm
- Each problem have several questions, understand and answer each question, no justification/correctness proof

A midterm exam example has been released early today

Topics

- Analysis of running time
- Graphs
- Greedy algorithms

Time Complexity

The time complexity of an algorithm associates a number **T(N)**, the “time” the algorithm takes on problem size **N**.

Mathematically, **T** is a function that maps positive integers giving problem size to positive integers giving number of simple operations

Worst Case Complexity: **max** # simple operations algorithm takes on any input of size **N**

Analysis of running time

Given two positive functions **f** and **g**

f(N) is **O(g(N))** iff there is a constant $c>0$ and $N_0 \geq 0$ s.t.,
$0 \leq f(N) \leq c \cdot g(N)$ for all $N \geq N_0$

f(N) is **Ω(g(N))** iff there is a constant $c>0$ and $N_0 \geq 0$ s.t.,
$f(N) \geq c \cdot g(N) \geq 0$ for all $N \geq N_0$

f(N) is **Θ(g(N))** iff there are $c_0>0$, $c_1>0$ and $N_0 \geq 0$ s.t.

$c_0 \cdot g(N) \leq f(N) \leq c_1 \cdot g(N)$ for all $N \geq N_0$

- f(N) is Θ(g(N)) iff f(N) is both O(g(N)) and Ω(g(N)).

Properties

Reflexivity. **f** is **$O(f)$**.

Constants. If **f** is **$O(g)$** and **$c > 0$**, then **$c \cdot f$** is **$O(g)$**.

Products. If **f_1** is **$O(g_1)$** and **f_2** is **$O(g_2)$**, then **$f_1 \cdot f_2$** is **$O(g1 \cdot g2)$.**

Sums. If **f_1** is **$O(g_1)$** and **f_2** is **$O(g_2)$**, then **$f_1 + f_2$** is **$O(\max \{g_1, g_2\})$**.

Transitivity. If **f** is **$O(g)$** and **g** is **$O(h)$**, then **f** is **$O(h)$**

Asymptotic Bounds for common fns

Polynomials:

$a_0 + a_1 n + \cdots + a_d n^d$ is $O\left(n^d\right)$

Logarithms:

$\log_a n = O(\log_b n)$ for all constants $a, b > 0$

Logarithms: log grows slower than every polynomial

For all $k > 0$, $\log n = O(n^k)$

$$n \log n = O(n^{1.01})$$

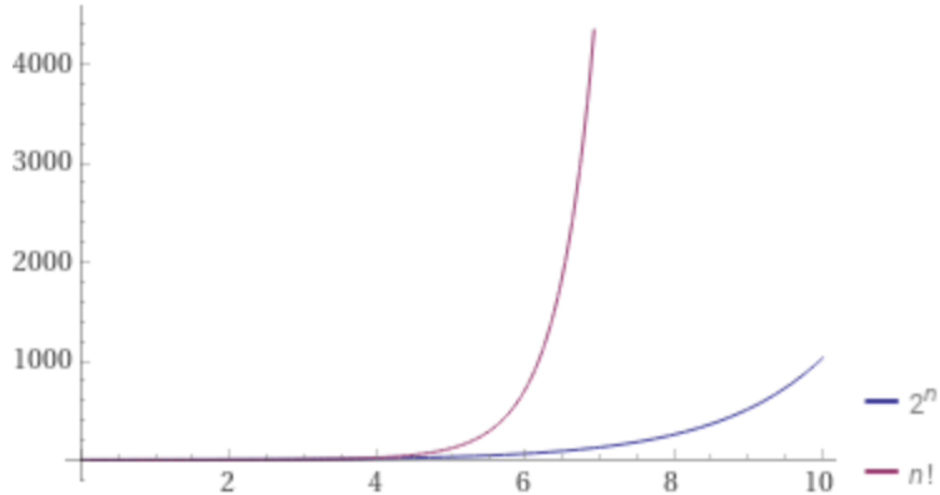
For two functions f and g, if $\log f$ is $O(\log g)$, but $\log g$ is not $O(\log f)$ then f is $O(g)$.

Exercise

Suppose $f(n) = n!, g(n) = 2^n$

Is $f = O(g), f = \Omega(g)$ or $f = \Theta(g)$?

Solution 1: Plot the two functions



Since n! is consistently larger than 2^n, $f = \Omega(g)$

Exercise

Suppose $f(n) = n!, g(n) = 2^n$

Is $f = O(g), f = \Omega(g)$ or $f = \Theta(g)$?

Solution 2: Consider

$$\frac{f(n)}{g(n)} = \underbrace{\left(\frac{1}{2}\right)\left(\frac{2}{2}\right)\dots\left(\frac{n}{2}\right)}_{n \text{ terms}} \geq \underbrace{\left(\frac{n}{4}\right)\dots\left(\frac{n}{2}\right)}_{n/2 \text{ terms}} \geq \left(\frac{n}{4}\right)^{n/2}$$

Solution 3: Take log.

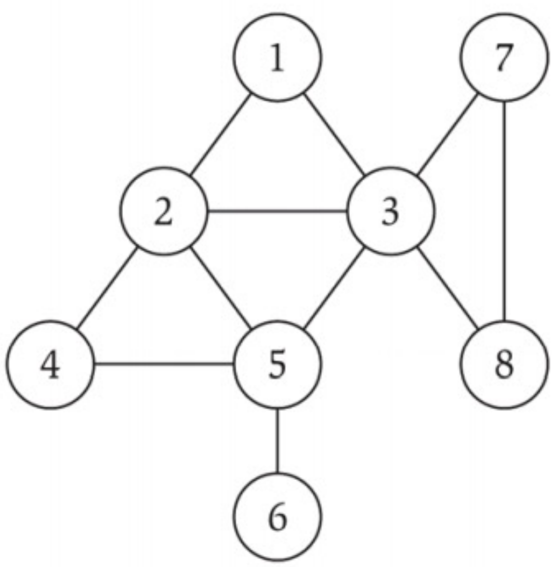
- $\log f(n) = \log 1 + \log 2 + \cdots + \log n = \Theta(n \log n)$
- $\log g(n) = n \log 2 = \Theta(n)$
- So, $\log f(n) = \Omega(\log g(n))$ and $\log f(n)$ is not $O(\log g(n))$, hence $f(n) = \Omega(g(n))$

If f= Ω(g) and f is not O(g), then $2^f = \Omega(2^g)$

Undirected Graphs G=(V,E)

Notation. G = (V, E)

- V = nodes (or vertices)
- E = edges between pairs of nodes
- Captures pairwise relationship between objects
- Graph size parameters: n = |V|, m = |E|

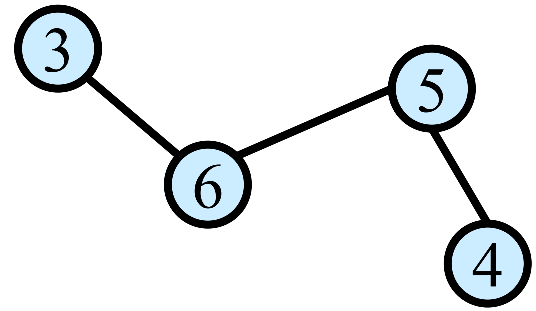


No self-loop, no multiedge

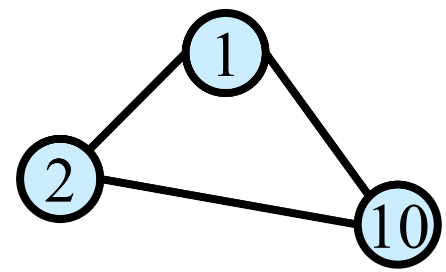
V = {1, 2, 3, 4, 5 ,6, 7, 8}
E = {(1,2), (1,3), (2,3), (2,4), (2,5), (3,5), (3,7), (3,8), (4,5), (5,6), (7,8)}
m=11, n=8

Terminology

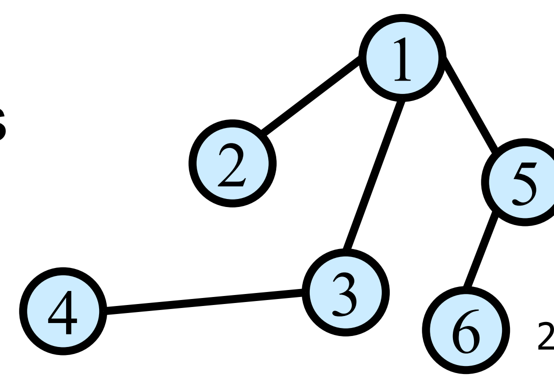
Path: A sequence of vertices s.t. each vertex is connected to the next vertex with an edge



Cycle: Path of length > 2 that has the same start and end



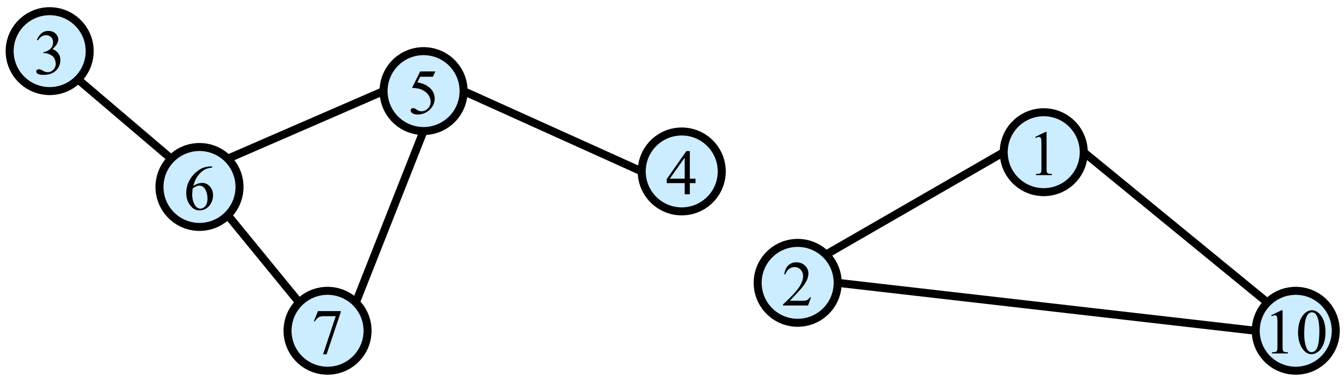
Tree: A connected graph with no cycles



Terminology

Degree of a vertex: # edges that touch that vertex

deg(6)=3



Connected: Graph is connected if there is a path between every two vertices

Connected component: Maximal set of connected vertices

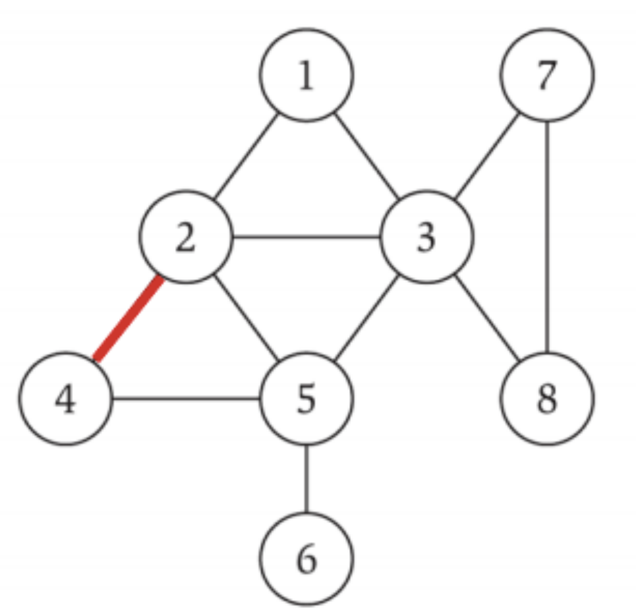
Graph representation

Adjacency matrix. n-by-n matrix with $A_{uv} = 1$ if (u, v) is an edge.

Space proportional to n^2.

Checking if (u, v) is an edge takes $\Theta(1)$ time.

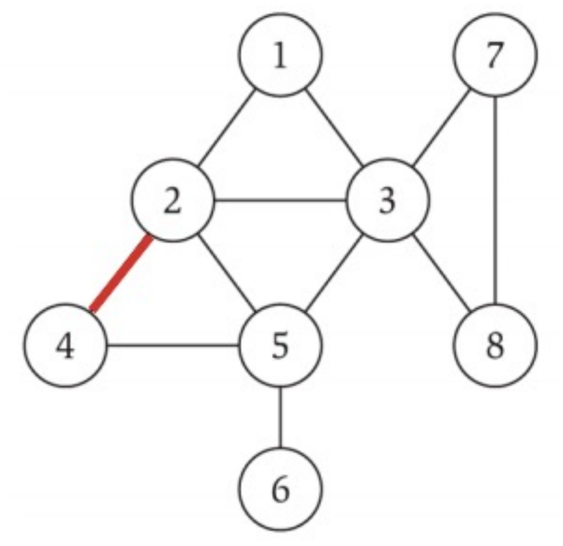
Identifying all edges takes $\Theta(n^2)$ time.

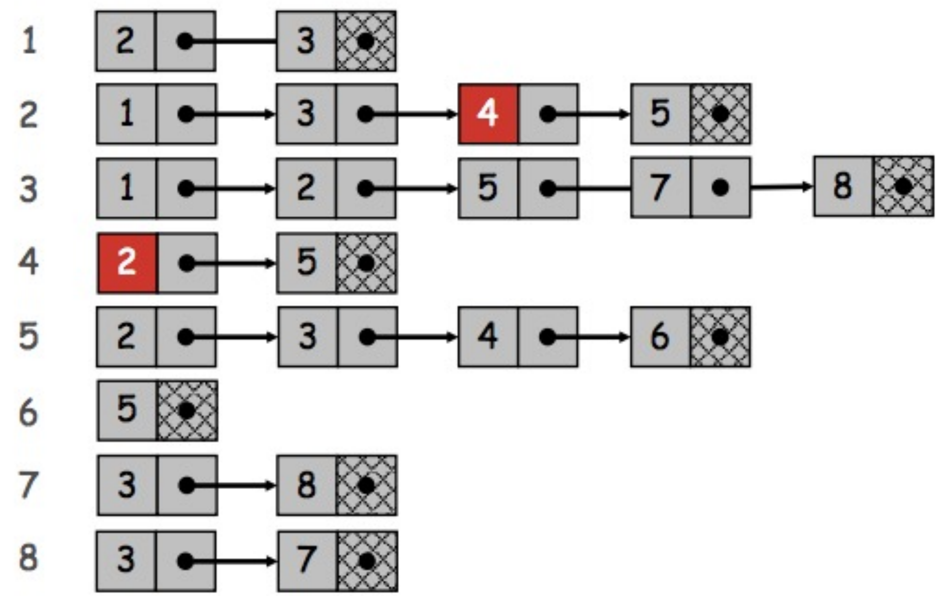


	1	2	3	4	5	6	7	8
1	0	1	1	0	0	0	0	0
2	1	0	1	1	1	0	0	0
3	1	1	0	0	1	0	1	1
4	0	1	0	0	1	0	0	0
5	0	1	1	1	0	1	0	0
6	0	0	0	0	1	0	0	0
7	0	0	1	0	0	0	0	1
8	0	0	1	0	0	0	1	0

Graph representation

Adjacency list. Node indexed array of lists.
Space proportional to $m+n$.
Checking if (u, v) is an edge takes $O(deg(u))$ time.
Identifying all edges takes $\Theta(m+n)$ time.





Graph Traversal

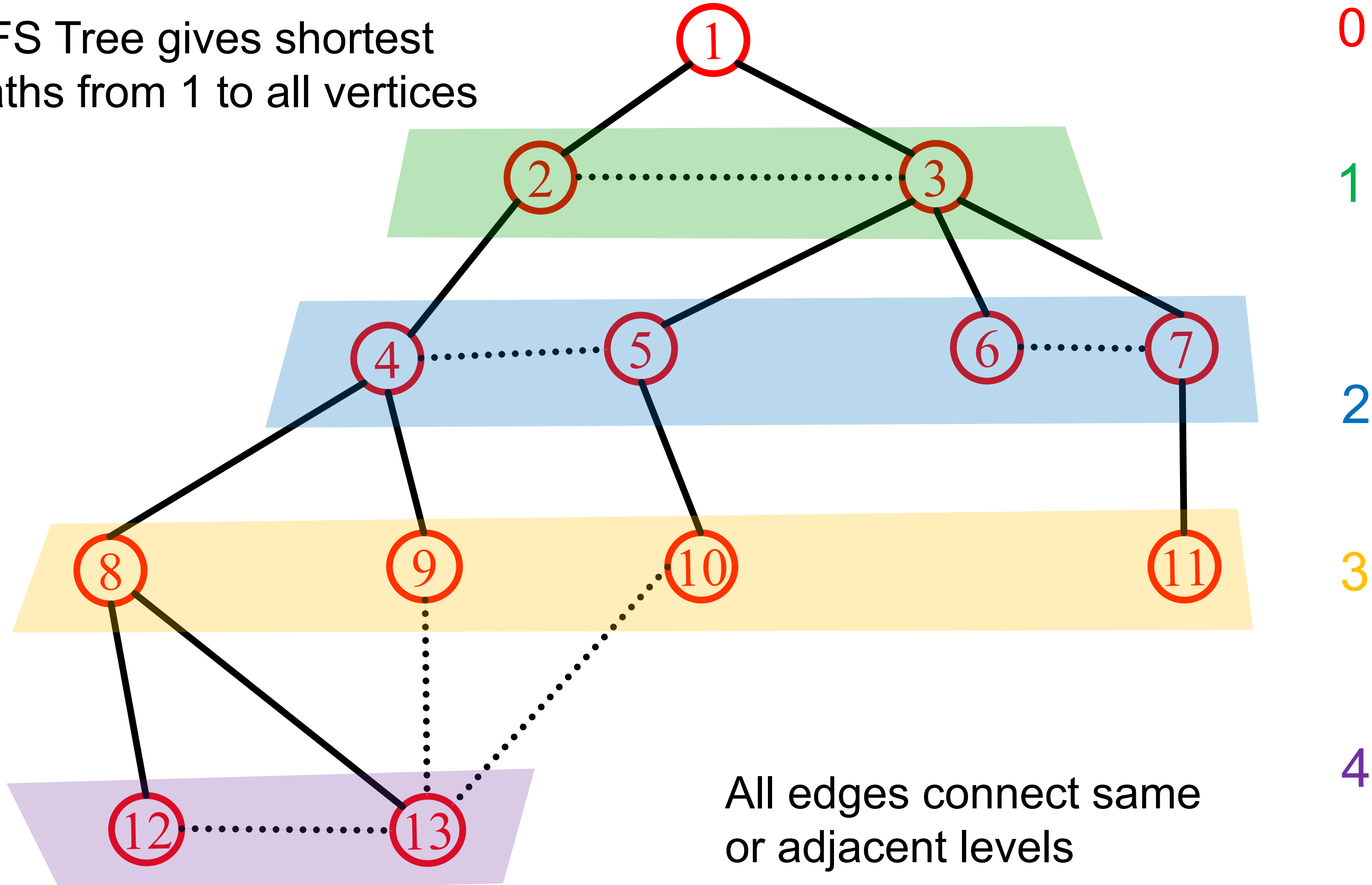
Walk (via edges) from a fixed starting vertex s to all vertices reachable from s.

Breadth First Search (BFS): Order nodes in successive layers based on distance from s

Depth First Search (DFS): More natural approach for exploring a maze;

BFS

BFS Tree gives shortest paths from 1 to all vertices



All edges connect same or adjacent levels

BFS

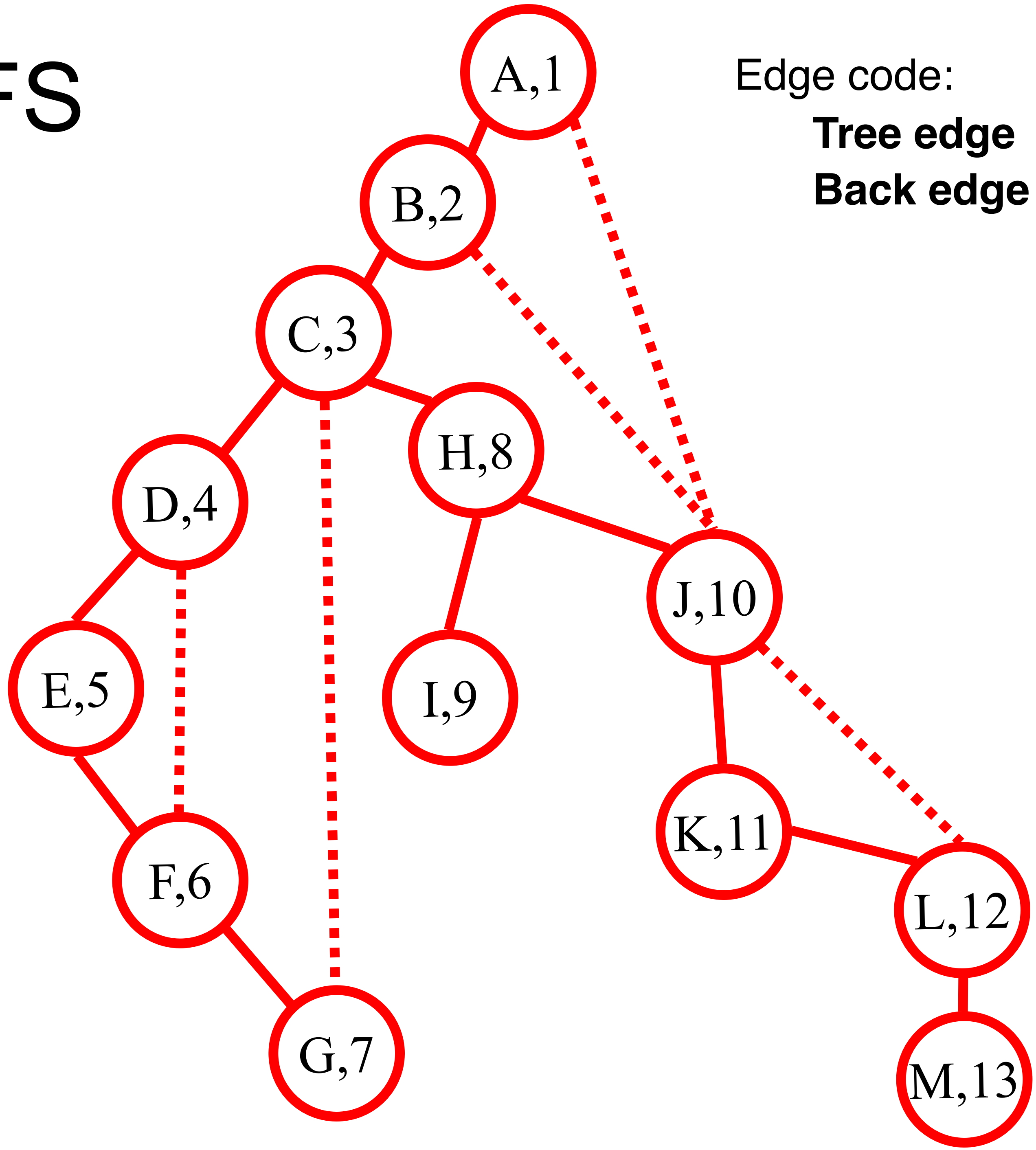
Properties:

- Edges into then-undiscovered vertices define a tree – the "Breadth First spanning tree" of G
- Level i in the tree are exactly all vertices v s.t., the shortest path (in G) from the root s to v is of length i
- All nontree edges join vertices on the same or adjacent levels of the tree

Applications:

- Find connected components
- Single source shortest part on unweighted undirected graph
- Testing bipartiteness

DFS



Edge code:

Tree edge

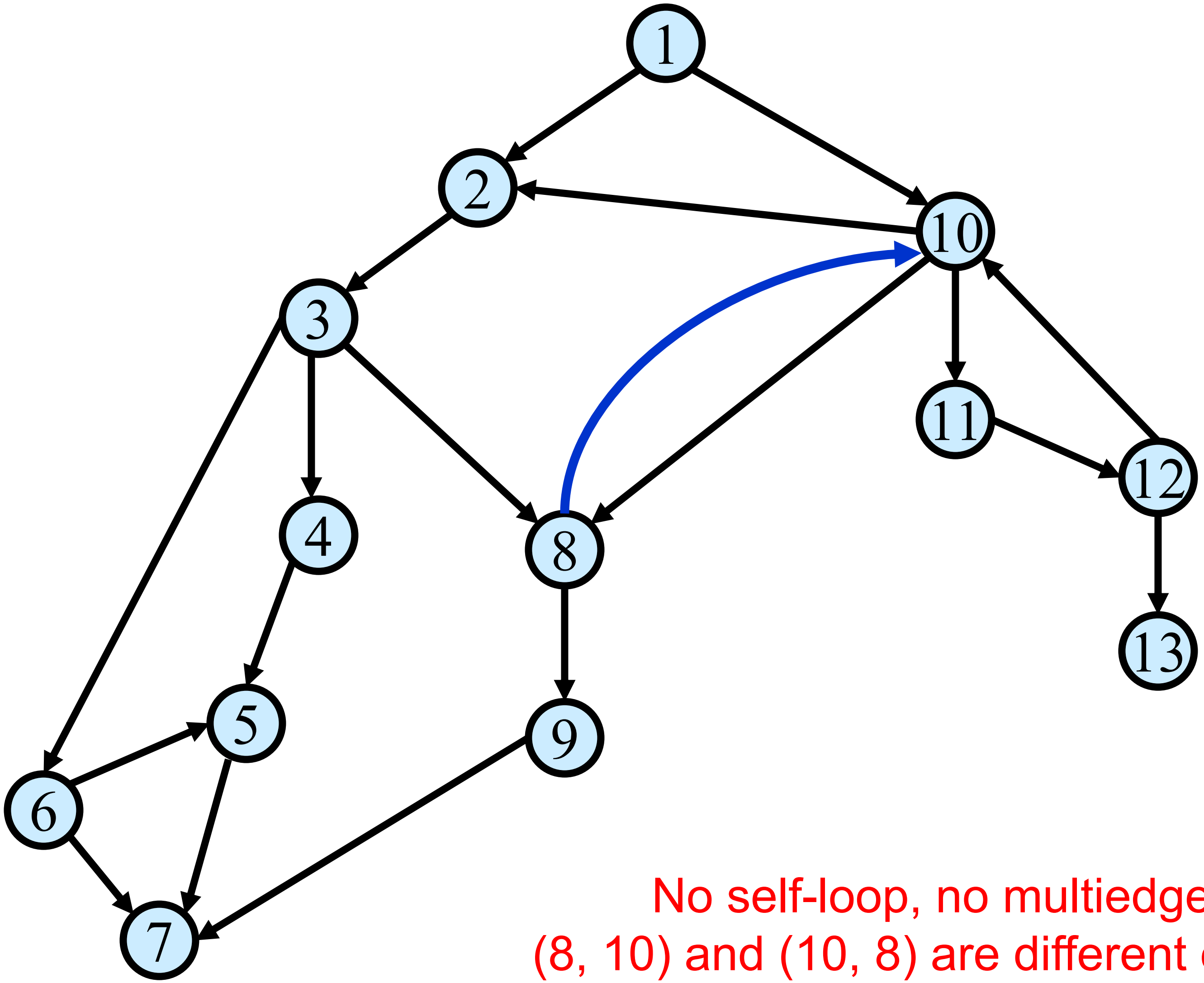
Back edge

DFS

Properties:

- Edges into then-undiscovered vertices define a “DFS tree” of G
- All nontree edges $\{x, y\}$, one of x or y is an ancestor of the other in the DFS tree.

Directed Graphs

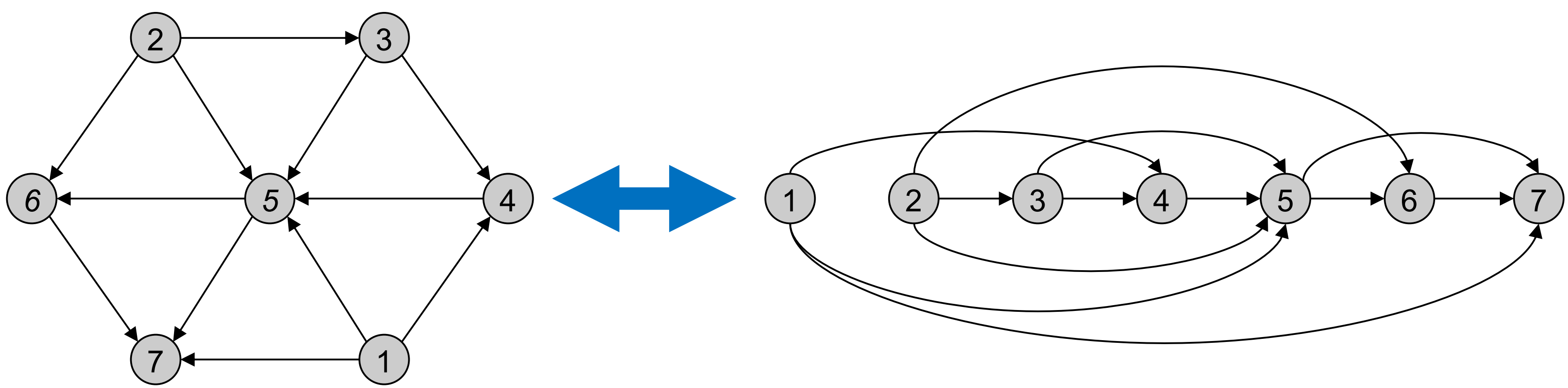


No self-loop, no multiedge
(8, 10) and (10, 8) are different edges

Directed Acyclic Graphs (DAG)

Def: A DAG is a directed acyclic graph, i.e., one that contains no directed cycles.

Def: A topological order of a directed graph G = (V, E) is an ordering of its nodes as $v_1, v_2, \ldots, v_n$ so that for every edge (v_i, v_j) we have $i < j$.



a DAG

a topological ordering of that DAG– all edges left-to-right

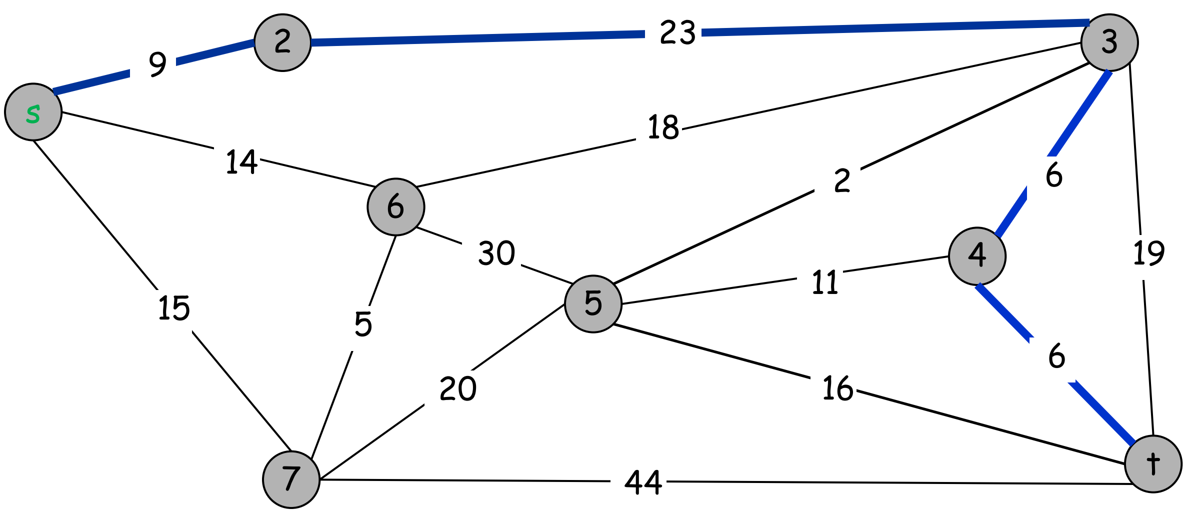
Single Source Shortest Path

Given an (un)directed connected graph $G = (V, E)$ with non-negative edge weights $c_e \geq 0$ and a start vertex s.

Find length of shortest paths from s to each vertex in G

length of path = sum of edge weights in path

Dijkstra's algorithm



Cost of path s-2-3-4-t
= 9 + 23 + 6 + 6
= 44.

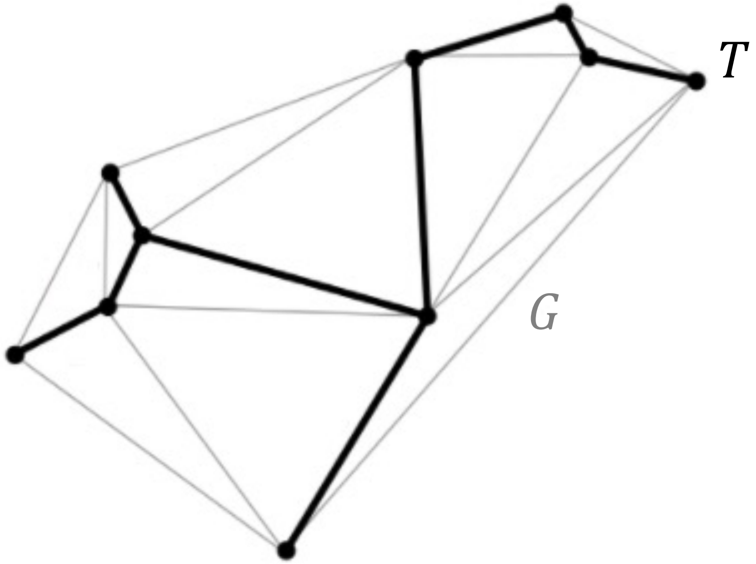
Spanning Tree

Given a connected undirected graph $G = (V, E)$.

We call T is a spanning tree of G if

All edges in T are from E.

T includes all of the vertices of G.



Kruskal's algorithm

Homework 1 Problem 3

Give an algorithm to detect whether a given undirected connected graph contains a cycle. If the graph contains a cycle, then your algorithm should output one cycle (do not output all cycles in the graph, just any one of them). Justify the running time bound of your algorithm.

- Run BFS with an arbitrary vertex v as the start vertex, and let T be the BFS tree
- If there is no off-tree edge, then there is no cycle of the graph
- If there is an off-tree edge, then the graph must contain a cycle.
  - How do we find the cycle? Find the cycle contains the off-tree edge
  - Let (x, y) be the off-tree edge. Find the path from x to y in T. The path and with edge (x, y) form a cycle.

- DFS also works

Greedy Algorithms

'Best' current partial solution at each step

- Solution is built in small steps
- Decisions on how to build the solution are made to maximize some criterion without looking to the future
  - Want the 'best' current partial solution as if the current step were the last step

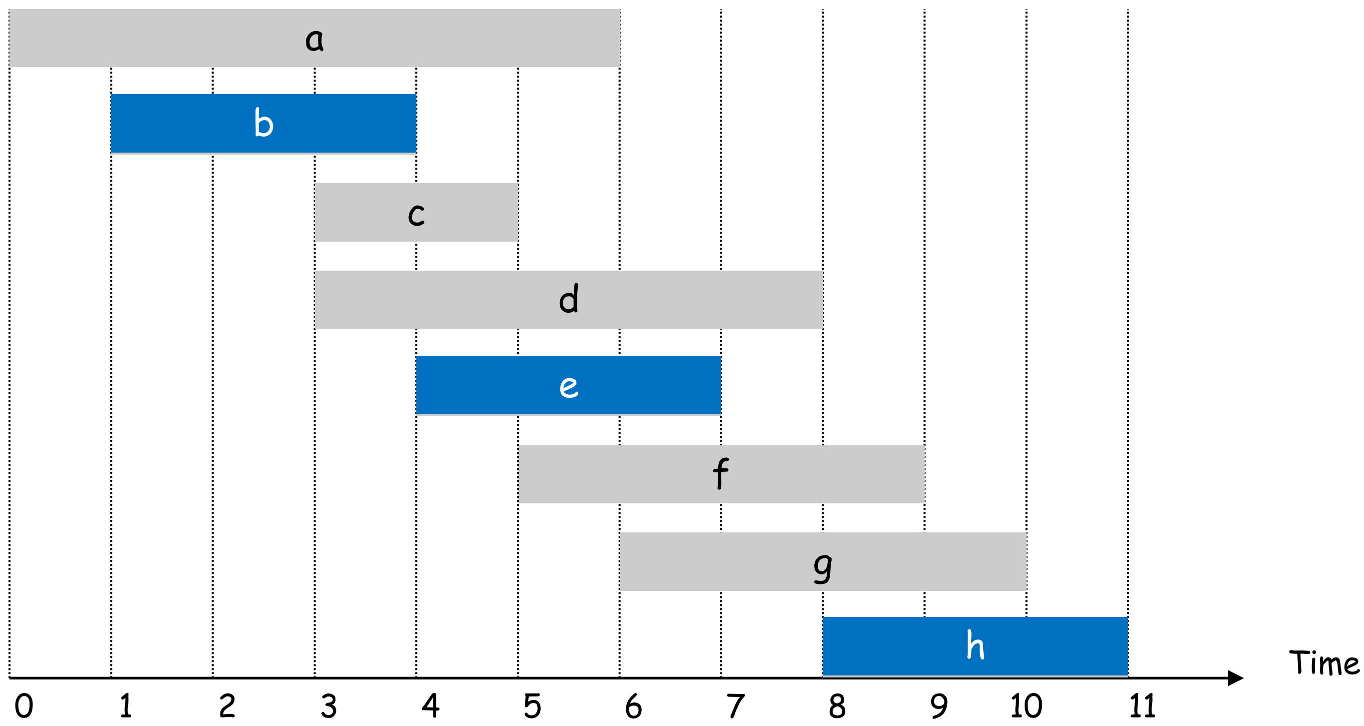
How to define each step?

What is the strategy of each step?

Interval Scheduling

Interval Scheduling

- Job j starts at $s(j)$ and finishes at $f(j)$.
- Two jobs compatible if they don't overlap.
- Goal: find maximum subset of mutually compatible jobs.



Interval Scheduling

Every step we consider a single job

In each step, we decide if the job will be in the solution set

- Strategy: if the job is compatible with the current solution set, put it into the solution set

How do we order jobs?

- Sort in the ascending order of the finish times

Homework 2 Problem 3

A shop is selling candies at a discount. For **every two** candies sold, the shop gives a **third** candy for **free**.

The customer can choose **any** candy to take away for free as long as the cost of the chosen candy is less than or equal to the **minimum** cost of the two candies bought.

- For example, if there are `4` candies with costs `1`, `2`, `3`, and `4`, and the customer buys candies with costs `2` and `3`, they can take the candy with cost `1` for free, but not the candy with cost `4`.

Each step we buy two (remaining) candies and get another for free

- Since we want to buy all the candies, we maximize the cost of free candy
- So we buy the most expensive two, and get the third expensive one for free

Algorithm: Sort candies by descending order of cost, for the remaining candies, by the first two and get the third for free