CSC148H Lecture 11

Dan Zingaro
OISE/UT

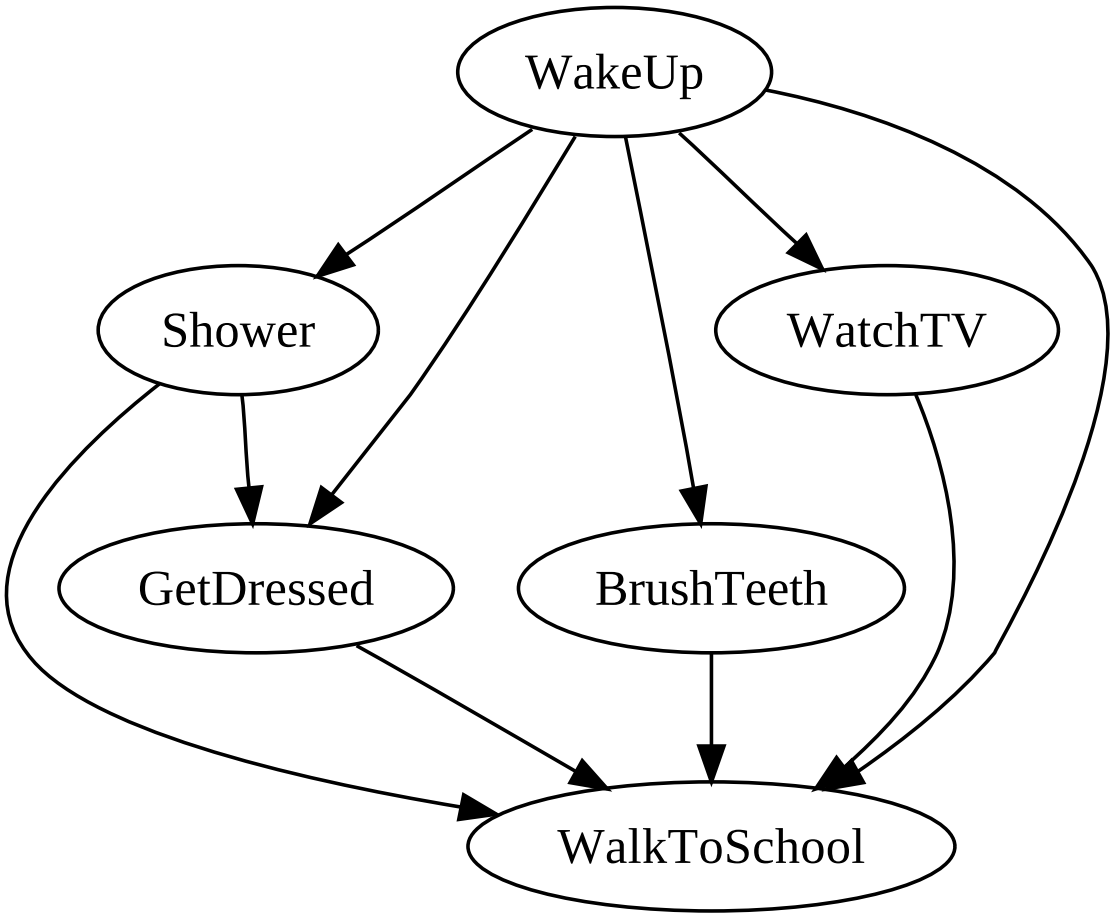
November 24, 2008

Motivating Graphs

- Let's say we want to represent the constraints on the stuff Dan does in the morning
- Dan has to wake up, shower, brush teeth, watch TV (obviously!), get dressed, and walk to school. (Sadly, I'd be lying if I included "have breakfast")
- Constraints include
  - Must wake up before shower
  - Must get dressed before walking to school
  - Must shower before getting dressed

Motivating Graphs...

Let the nodes represent the activities, and let a directed edge from *a* to *b* indicate that *a* must be done before *b*

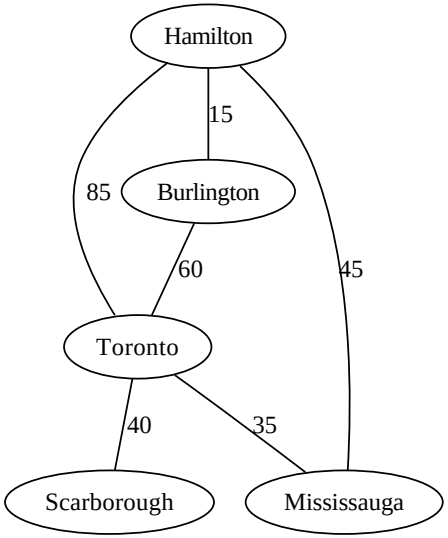


Motivating Graphs...

- The diagram on the previous slide contains nodes and edges, but it's not a tree, because some nodes have more than one parent
- A graph is a set of nodes (or vertices) and edges that relaxes the restrictions on trees
- Two types of graphs: undirected and directed
  - Undirected: edges have no orientation (i.e. if edge (v, w) exists, so does edge (w, v))
  - Directed: edges have an orientation (i.e. we can have edge (v, w) without edge (w, v))
- When we draw directed graphs (called digraphs), we use arrows to indicate the direction of the edges

Labeled Graphs

- Sometimes, the fact that an edge does or does not exist between two vertices is not enough information
- Consider an undirected graph whose vertices are cities, and where the existence of an edge connecting two cities means that you can travel between them
- We can associate a label with each edge to give the time in minutes that it takes to travel between the cities at its ends (e.g. 60 mins between Burlington and TO, 35 mins between Mississauga and TO, 40 mins between Scarborough and TO)
- Edge labels are usually referred to as weights



Definitions

- A path is a sequence of vertices $v_1, v_2, \ldots, v_n$ such that $(v_1, v_2), (v_2, v_3), \ldots, (v_{n-1}, v_n)$ are edges in the graph
- A path is simple if all vertices, except possibly the first and last, are distinct
- The length of a path is the number of edges it contains
- In a graph where edges are labeled with weights, the weighted path length is the sum of the weights of its edges
- The weight of the shortest path between two vertices is the minimum weighted path length between them
- Question: what is the weight of the shortest path between Hamilton and Toronto on the previous slide?

Definitions...

- If we have edge (v, w), we say that
  - v is the tail of the edge
  - w is the head of the edge
  - w is adjacent to (or incident to) v
- It's common to see edge (v, w) written as $v \longrightarrow w$
- The degree of a vertex is the number of vertices adjacent to that vertex

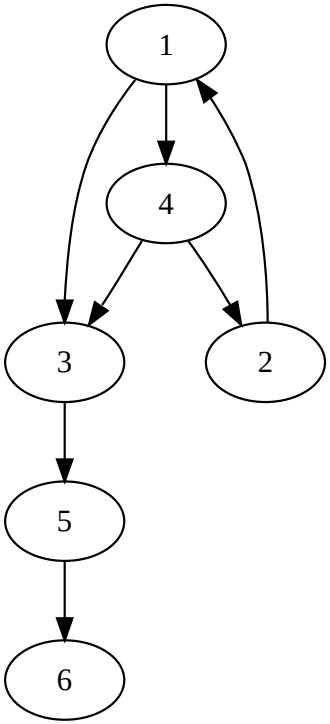
The Graph ADT

- Like trees, stacks, queues, binary search trees, heaps, priority queues, etc., a graph is an ADT
- Its operations include
  - Retrieve weight of an edge
  - Add or remove an edge
  - Retrieve the first vertex adjacent to vertex v
  - Retrieve the next vertex adjacent to v (like an iterator)
  - Retrieve all vertices adjacent to v

Graphs as Adjacency Matrices

- We can represent a graph using a matrix
- Unweighted graph (i.e. no edge weights)
  - If entry $(i, j) = 1$, there is an edge from i to j
  - If entry $(i, j) = 0$, there is no edge from i to j
- Weighted graph
  - The value at entry (i, j) gives us the weight of the edge from i to j
  - If 0 is a valid weight, we must have some other way to represent "no edge"
- Adjacency matrix for an undirected graph is symmetric (i.e. entry (i, j) and entry (j, i) are equal)

Example: Graph and its Adjacency Matrix



	1	2	3	4	5	6
1	0	0	1	1	0	0
2	1	0	0	0	0	0
3	0	0	0	0	1	0
4	0	1	1	0	0	0
5	0	0	0	0	0	1
6	0	0	0	0	0	0

Graphs as Adjacency Lists

- An alternative way to represent the graph ADT is with adjacency lists
- For each vertex v, we maintain a list containing the vertices adjacent to v
- e.g. the adjacency list representation of the graph on the previous slide is below

1	[3, 4]
2	[1]
3	[5]
4	[2,3]
5	[6]
6	[]

Adjacency Matrix vs. Adjacency List

- Consider a graph with v vertices and e edges
- Storage
  - Adjacency matrix: $O(v^2)$ space
  - Adjacency list:$O(v + e)$ space (v space for the vertices, e space for the edges)
- Determining if a vertex is adjacent to another
  - Adjacency matrix: $O(1)$ (just index into the matrix)
  - Adjacency list: $O(v)$ (i.e. a vertex may have every other vertex adjacent to it)
- Bottom line: which is "better" depends on what you're using it for

Adjacency Matrix Implementation

```
class Graph:

    def __init__(self, num_vertices, is_directed = False):
        self.num_vertices = num_vertices
        self.matrix = [[0]*num_vertices for i in range(num_vertices)]
        self.is_directed = is_directed

    def add_edge(self, v1, v2):
        self.matrix[v1][v2] = 1
        if not self.is_directed:
            self.matrix[v2][v1] = 1

    def has_edge(self, v1, v2):
        return self.matrix[v1][v2] == 1
```

Adjacency List Implementation

```
class Graph:

    def __init__(self, num_vertices, is_directed = False):
        self.num_vertices = num_vertices
        self.vertices = [[] for i in range(num_vertices)]
        self.is_directed = is_directed

    def add_edge(self, v1, v2):
        self.vertices[v1].append(v2)
        if not self.is_directed:
            self.vertices[v2].append(v1)

    def has_edge(self, v1, v2):
        return v2 in self.vertices[v1]
```

Graph Traversals

- Two ways to traverse a graph and "visit" its nodes: depth-first search (DFS) and breadth-first search (BFS)
- Depth-first is a generalization of the preorder traversal on trees
- It works by selecting a start vertex v, marking it visited, then recursively doing a depth-first search on each unvisited vertex adjacent to v
- This visits all nodes if there is a path between the start vertex and every other node
- Otherwise, it visits all nodes that can be reached from the start vertex; to do a DFS of the whole graph, we'd have to start a DFS from each unvisited vertex that remains

DFS Pseudocode

```
def dfs(graph, start_vertex):
    dfs_helper(graph, start_vertex, set([]))

def dfs_helper(graph, vertex, discovered):
    add vertex to discovered
    visit vertex
    for vertex2 in neighbors of vertex:
        if vertex2 is not in discovered:
            dfs_helper(graph, vertex2, discovered)
```

Graph Traversals...

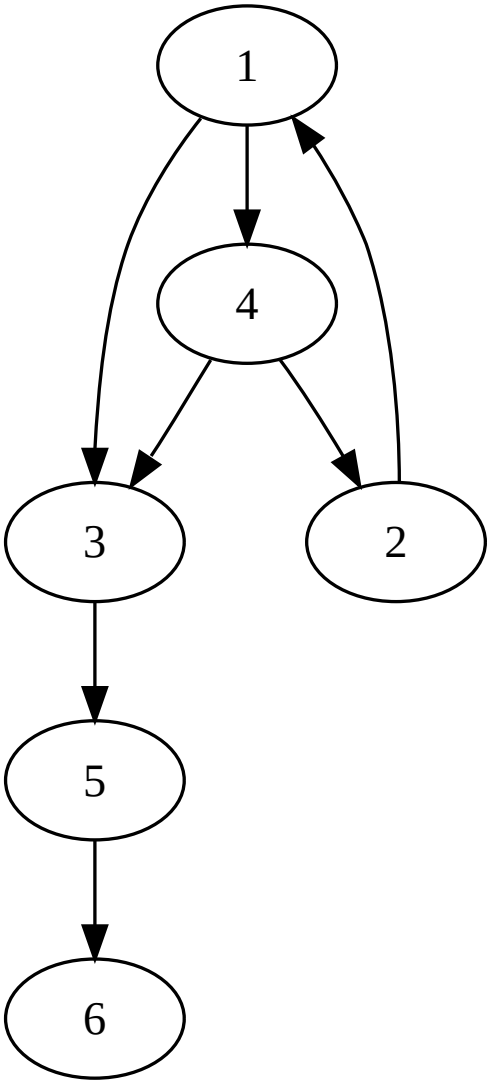
- Breadth-first search works by selecting a start vertex v, marking it visited, then visiting the vertices adjacent to v
- Assume w is the first vertex adjacent to vertex v, x is the second vertex adjacent to v, etc.
- After we visit all vertices adjacent to v, we visit all vertices adjacent to w, visit all vertices adjacent to x, and so on
- We use a queue to organize this graph search

BFS Pseudocode

```
def bfs (graph, start_vertex):
    add start_vertex to discovered
    enqueue start_vertex into queue
    while queue is not empty:
        vertex = dequeue from queue
        visit vertex
        for vertex2 in neighbors of vertex:
            if vertex2 is not in discovered:
                add vertex2 to discovered
                enqueue vertex2 into queue
```

DFS and BFS

In which order does DFS visit the nodes? How about BFS?



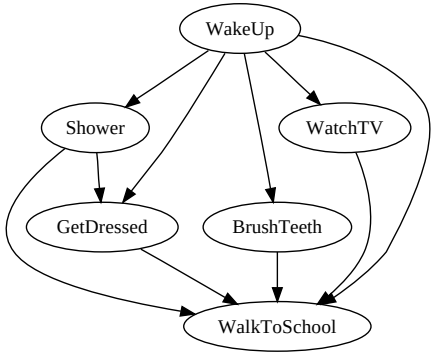
Topological Sort

- A graph has a cycle if there is a path (of at least one edge) that begins and ends at the same node
- If a graph is directed and contains no cycles, we have a directed acyclic graph (DAG)
- DAGs are more general than trees but less general than arbitrary graphs
- They have many uses, including representing scheduling constraints (like in our wakeup example)
- A topological sort on a DAG is a linear list of vertices such that if i must be done before j, then i appears in the list before j
- We can then solve a scheduling problem by carrying out the tasks in the order they appear in the list
- There is no topological sort of a graph with a cycle

Topological Sort...

- We can base a topological sort directly on DFS
- (DFS is often thought of as a skeleton, from which many graph algorithms can be built.)
- If we insert the key of node n at the beginning of a list right after we have finished a DFS from n, we will have created a topological sort
- Just like for regular DFS, we must perform a topological sort from each unvisited vertex if we want to consider the whole graph

Sample Topological Sort



- DFS from WakeUp
  - DFS from Shower
    - DFS from WalkToSchool
    - 'WalkToSchool'
    - DFS from GetDressed
    - 'GetDressed', 'WalkToSchool'
  - 'Shower', 'GetDressed', 'WalkToSchool'
  - DFS from BrushTeeth
  - 'BrushTeeth', 'Shower', 'GetDressed', 'WalkToSchool'
  - DFS from WatchTV
  - 'WatchTV', 'BrushTeeth', 'Shower', 'GetDressed', 'WalkToSchool'
- 'WakeUp', 'WatchTV', 'BrushTeeth', 'Shower', 'GetDressed', 'WalkToSchool'

Proof of Topological Sort

We will prove: if there is an edge (v, w), the DFS on w will be completely executed prior to completion of the DFS on v

- Case 1: we are performing a DFS from v and find that w has not yet been discovered
  - Proof: we will make the recursive call to perform a DFS from w; only then will we finish the DFS on v
- Case 2: we are performing a DFS from v and find that a full DFS from w has already occurred
  - Proof: nothing to prove here
- Case 3: we are performing a DFS from v and find that a DFS from w has been initiated but not completed (i.e. the recursive call from w is still on the recursion stack)
  - This cannot happen. If it did, we must have earlier initiated a DFS from w and have therefore found a path beginning at w, passing through v and ending at w (a cycle!)

Topological Sort Implementation

- The implementation calls a helper function for each vertex that hasn't been explored already; the helper function performs a topological sort from the given vertex
- At the end, `v_list` will contain one possible topological sort of the DAG

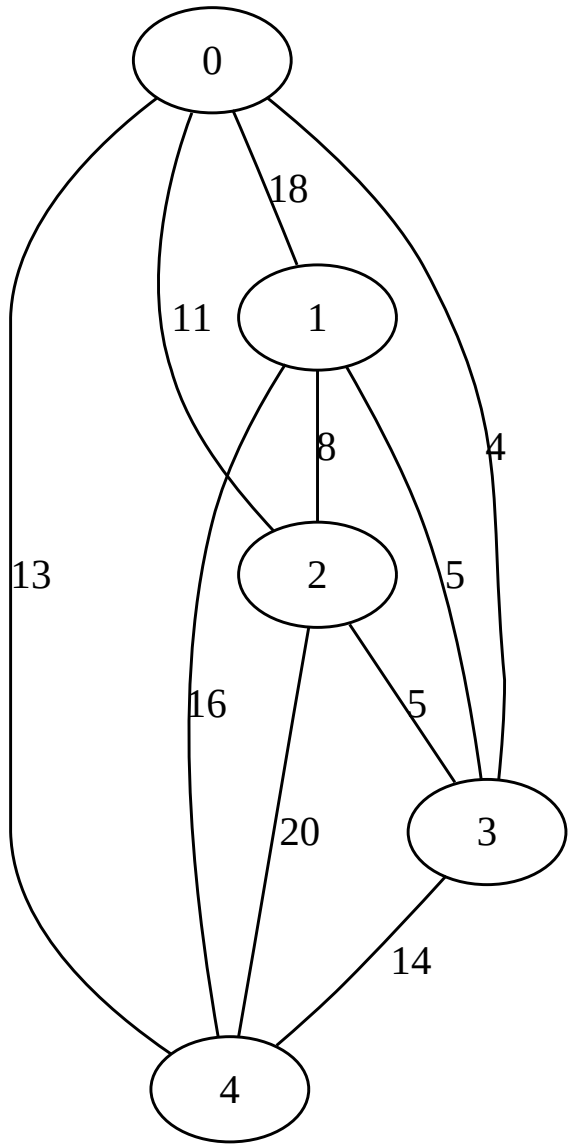
```
def topsort(g):
  v_list = []
  discovered = set([])
  for v in range(g.num_vertices):
    if v not in discovered:
      topsort_helper (g, v, discovered, v_list)
  return v_list
```

Topological Sort Implementation...

```
def topsort_helper (g, vertex, discovered, v_list):
  discovered.add(vertex)
  adj = g.get_neighbours(vertex)
  for v in adj:
    if v not in discovered:
      topsort_helper (g, v, discovered)
  v_list.insert (0, vertex)
```

Minimum Spanning Trees

Consider this graph whose nodes are locations and whose edges are the costs to run cabling between those locations. (i.e. it costs 5 to



connect locations 2 and 3.)

Minimum Spanning Trees...

- Our goal is to minimize the amount of cable but still have a path of cable from each location to every other location
- This involves selecting a subset of the edges, called the minimum-cost spanning tree (MST)
- A tree is the best we can do, because if we remove one of its edges, we will disconnect the graph
- We will use Kruskal's algorithm to find an MST
- It is a "greedy" algorithm: it makes the locally best choice to arrive at a global optimal
- A well-known greedy algorithm: counting Canadian change

Kruskal's Algorithm

- Kruskal first sorts the edges in order of nondecreasing cost
- For the above graph: (0, 3), (1, 3), (2, 3), (1, 2), (0, 2), (0, 4), (3, 4), (1, 4), (0, 1), (2, 4)
- Then, starting with the empty MST, Kruskal considers each edge in the sorted order
- If the edge can be added to the MST without creating a cycle, it is added; otherwise it is skipped

Execution on Sample Graph

Edges: **(0, 3)**, (1, 3), (2, 3), (1, 2), (0, 2), (0, 4), (3, 4), (1, 4), (0,

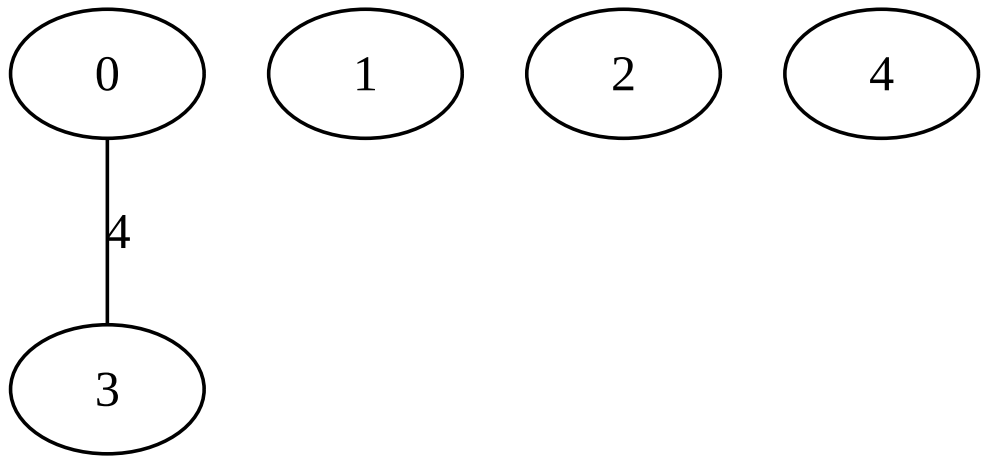


Figure: Edge (0, 3) is added.

Execution on Sample Graph...

Edges: (0, 3), **(1, 3)**, (2, 3), (1, 2), (0, 2), (0, 4), (3, 4), (1, 4), (0,

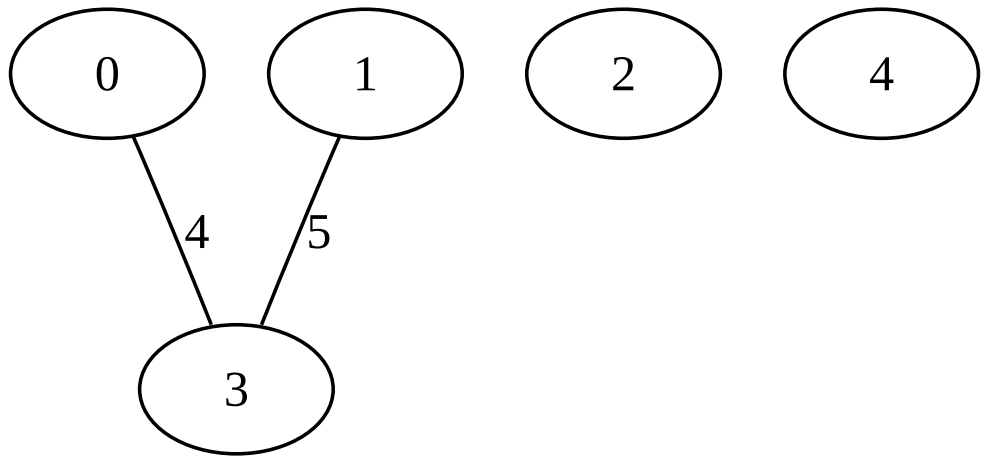


Figure: Edge (1, 3) is added.

Execution on Sample Graph...

Edges: (0, 3), (1, 3), **(2, 3)**, (1, 2), (0, 2), (0, 4), (3, 4), (1, 4), (0,

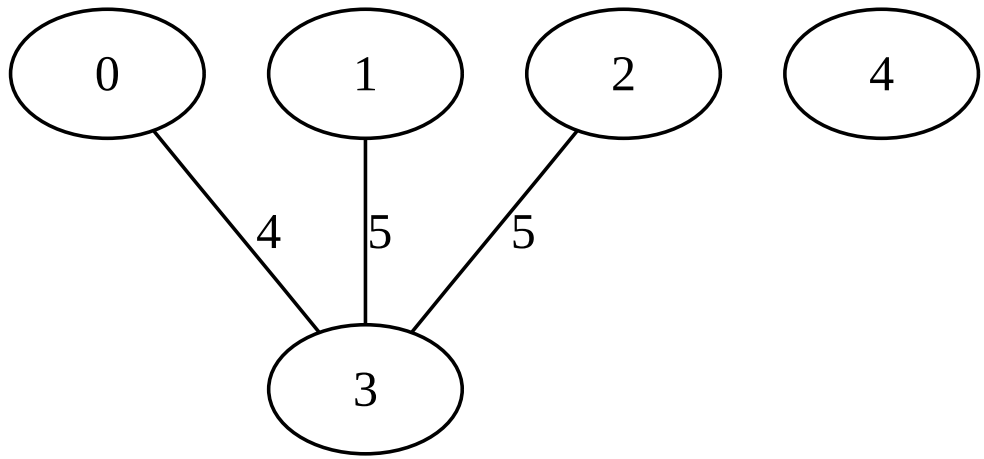


Figure: Edge (2, 3) is added.

Execution on Sample Graph...

Edges: (0, 3), (1, 3), (2, 3), (1, 2), (0, 2), **(0, 4)**, (3, 4), (1, 4), (0,

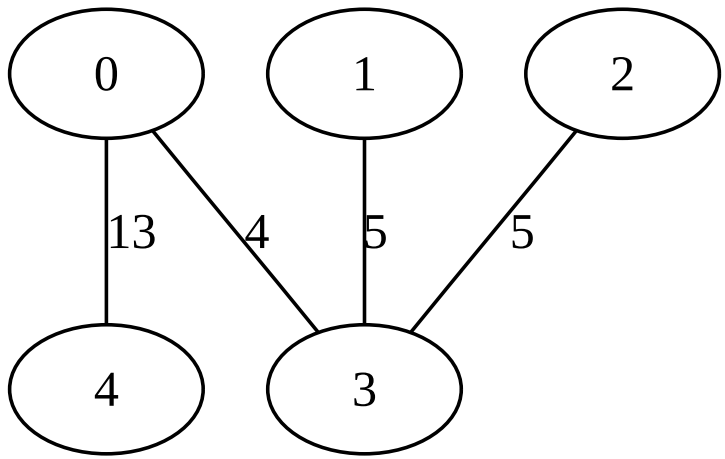


Figure: Edge (0, 4) is added. Edges (1, 2) and (0, 2) were rejected.

Proof of Kruskal

- The essential loop invariant is that the edges e we have selected so far can be completed into an MST t (I1)
- In other words, e is a subset of t
- Before we start adding edges, e is empty and so is certainly part of any MST of the graph
- Now, on each iteration where we include or exclude an edge, we assume that I1 holds prior to the iteration and show that it still holds after
- If we exclude an edge, I1 still holds since we haven't added anything to e to violate it
- If we add an edge that belongs to MST t, I1 can still be completed into the MST t
- So the only problematic case is when we add an edge to e that is not in t

Proof of Kruskal...

- Invariant: the edges e we have selected so far can be completed into an MST t (I1)
- Consider adding edge (u, v) to e, where (u, v) is not in t
- Adding (u, v) to t creates a cycle in t; call the result t_1
- Since we have no cycles in e, at least one edge (x, y) on the cycle is not present in e
- If we remove (x, y) from t_1, we have a new spanning tree t_2 of which e is a subset
- We have **not** shown that t_2 is still an MST, though

Proof of Kruskal...

- To prove that t_2 is an MST, we have to show that the cost of (x, y) (the edge we removed from t) is at least the cost of (u, v) (the new edge we added to t_2)
- Since the cost of t_2 cannot be smaller than the cost of t, this would imply that t_2 is an MST
- Since (u, v) is the smallest unprocessed edge, we just have to show that (x, y) is also unprocessed
- We know that (x, y) is not in e, and e is a subset of the processed edges

Proof of Kruskal...

- Why can't (x, y) exist among the remainder of the processed edges? (i.e. why can't it be one of the edges we rejected so far?)
- Remember from I1 that e is a subset of t, so e together with (x, y) would still be a subset of t (i.e. including (x, y) with e would still have no cycles)
- If this was the case, we would have added (x, y) to e when we processed (x, y). But, we know that (x, y) does not exist in e!
- Therefore, (x, y) cannot be a processed edge
- When the loop terminates, e **is** an MST from I1 and the fact that we have $v - 1$ edges