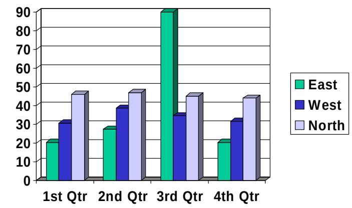
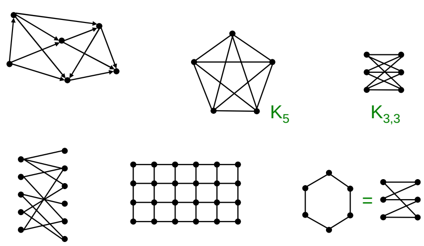
Graphs and Graph Algorithms

This is not a Graph



...not the kind we mean, anyway

These are Graphs



Applications of Graphs

- Communication networks
- Routing and shortest path problems
- Commodity distribution (flow)
- Traffic control
- Resource allocation
- Geometric modeling
- ...

Graph Definitions

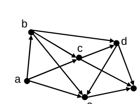
A *directed graph* (or *digraph*) is a pair (V,E) where
- V is a set
- E is a set of ordered pairs (u,v), where $u,v \in V$
- usually require $u \neq v$ (no self-loops)

An element of V is called a *vertex* (pl. *vertices*) or *node*
An element of E is called an *edge* or *arc*

|V| = size of V, often denoted n
|E| = size of E, often denoted m

Graph Definitions

Example:



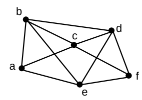
V = {a,b,c,d,e,f}
E = {(a,b), (a,c), (a,e), (b,c), (b,d), (b,e), (c,d), (c,f), (d,e), (d,f), (e,f)}

|V| = 6, |E| = 11

Graph Definitions

An *undirected graph* is just like a directed graph, except the edges are *unordered pairs* (*sets*) {u,v}

Example:



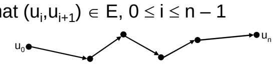
V = {a,b,c,d,e,f}
E = {{a,b}, {a,c}, {a,e}, {b,c}, {b,d}, {b,e}, {c,d}, {c,f}, {d,e}, {d,f}, {e,f}}

More Graph Definitions

- the vertices u and v are called the *source* and *sink* of the directed edge (u,v), respectively
- u and v are called the *endpoints* of (u,v)
- two vertices are *adjacent* if they are connected by an edge
- the *outdegree* of a vertex u in a directed graph is the number of edges of which u is the source
- the *indegree* of a vertex v is in a directed graph is the number of edges of which v is the sink
- the *degree* of a vertex u in an undirected graph is the number of edges of which u is an endpoint

More Graph Definitions

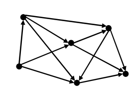
- a *path* is a sequence $u_0,u_1,u_2,\ldots,u_n$ of vertices such that $(u_i,u_{i+1}) \in E$, $0 \leq i \leq n-1$



- the *length* of the path is the number of edges in it (in this example, n – 1)
- a path is *simple* if it does not repeat any vertices
- a *cycle* is a path $u_0,u_1,u_2,\ldots,u_n$ such that $u_0 = u_n$
- a cycle is *simple* if it does not repeat any vertices except the first and last
- a graph is *acyclic* if it has no cycles
- a directed acyclic graph is called a *dag*

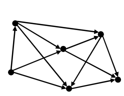
More Graph Definitions

Q) Is this a dag?



More Graph Definitions

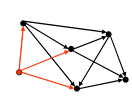
Q) Is this a dag?



A) yes – if and only if you can iteratively eliminate vertices of indegree 0 and get all the way through the graph

More Graph Definitions

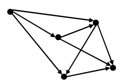
Q) Is this a dag?



A) yes – if and only if you can iteratively eliminate vertices of indegree 0 and get all the way through the graph

More Graph Definitions

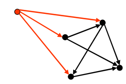
Q) Is this a dag?



A) yes – if and only if you can iteratively eliminate vertices of indegree 0 and get all the way through the graph

More Graph Definitions

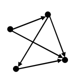
Q) Is this a dag?



A) yes – if and only if you can iteratively eliminate vertices of indegree 0 and get all the way through the graph

More Graph Definitions

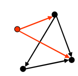
Q) Is this a dag?



A) yes – if and only if you can iteratively eliminate vertices of indegree 0 and get all the way through the graph

More Graph Definitions

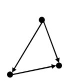
Q) Is this a dag?



A) yes – if and only if you can iteratively eliminate vertices of indegree 0 and get all the way through the graph

More Graph Definitions

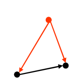
Q) Is this a dag?



A) yes – if and only if you can iteratively eliminate vertices of indegree 0 and get all the way through the graph

More Graph Definitions

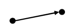
Q) Is this a dag?



A) yes – if and only if you can iteratively eliminate vertices of indegree 0 and get all the way through the graph

More Graph Definitions

Q) Is this a dag?



A) yes – if and only if you can iteratively eliminate vertices of indegree 0 and get all the way through the graph

More Graph Definitions

Q) Is this a dag?

A) yes – if and only if you can iteratively eliminate vertices of indegree 0 and get all the way through the graph

More Graph Definitions

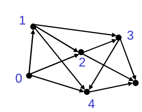
Q) Is this a dag?

A) yes – if and only if you can iteratively eliminate vertices of indegree 0 and get all the way through the graph

Topological Sort

Just computed a *topological sort* of the dag

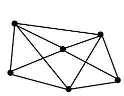
- a numbering of the vertices such that all edges go from lower- to higher-numbered vertices



Useful in job scheduling with precedence constraints

Graph Coloring

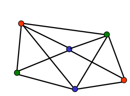
A *coloring* of an undirected graph is an assignment of a color to each node such that no two adjacent vertices get the same color



Q) How many colors are needed to color this graph?

Graph Coloring

A *coloring* of an undirected graph is an assignment of a color to each node such that no two adjacent vertices get the same color

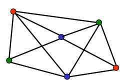


Q) How many colors are needed to color this graph?

A) 3

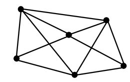
An Application of Coloring

- Vertices are jobs
- Edge (u,v) is present if jobs u and v each require access to the same shared resource, thus cannot execute simultaneously
- Colors are time slots to schedule the jobs
- Minimum number of colors needed to color the graph = minimum number of time slots required



Planarity

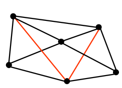
A graph is *planar* if it can be embedded in the plane with no edges crossing



Q) Is this graph planar?

Planarity

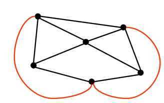
A graph is *planar* if it can be embedded in the plane with no edges crossing



Q) Is this graph planar?
A) yes

Planarity

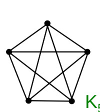
A graph is *planar* if it can be embedded in the plane with no edges crossing



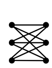
Q) Is this graph planar?
A) yes

Planarity

Kuratowski's Theorem



K_5



$K_{3,3}$

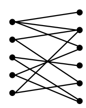
A graph is planar if and only if it does not contain a copy of K_5 or $K_{3,3}$ (possibly with other nodes along the edges shown)

The Four-Color Theorem

Every planar graph is 4-colorable
(Appel & Haken, 1976)

Bipartite Graphs

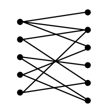
A directed or undirected graph is *bipartite* if the vertices can be partitioned into two sets such that all edges go between the two sets



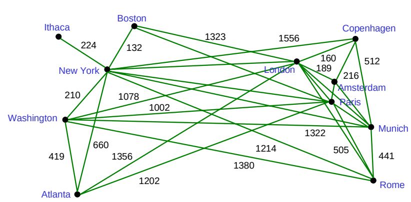
Bipartite Graphs

The following are equivalent:

- G is bipartite
- G is 2-colorable
- G has no cycles of odd length

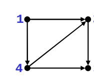


Traveling Salesperson

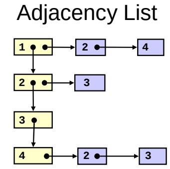


Find a path of minimum distance that visits every city

Representations of Graphs



Adjacency List



Adjacency Matrix

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

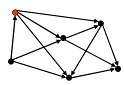
Graph Algorithms

- Search
  - depth-first search
  - breadth-first search
- Shortest paths
  - Dijkstra's algorithm
- Minimum spanning trees
  - Prim's algorithm
  - Kruskal's algorithm

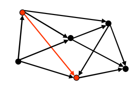
Depth-First Search

- Follow edges depth-first starting from an arbitrary vertex r, using a stack to remember where you came from
- When you encounter a vertex previously visited, or there are no outgoing edges, retreat and try another path
- Eventually visit all vertices reachable from r
- If there are still unvisited vertices, repeat
- O(m) time

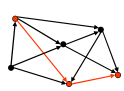
Depth-First Search



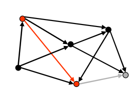
Depth-First Search



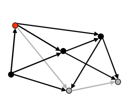
Depth-First Search



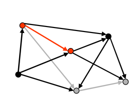
Depth-First Search



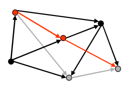
Depth-First Search



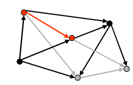
Depth-First Search



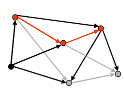
Depth-First Search



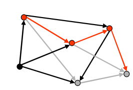
Depth-First Search



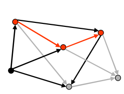
Depth-First Search



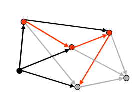
Depth-First Search



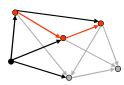
Depth-First Search



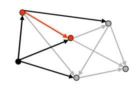
Depth-First Search



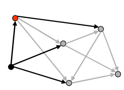
Depth-First Search



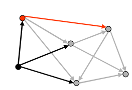
Depth-First Search



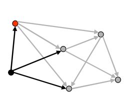
Depth-First Search



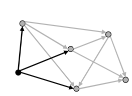
Depth-First Search



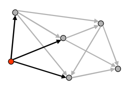
Depth-First Search



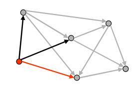
Depth-First Search



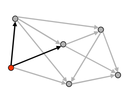
Depth-First Search



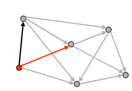
Depth-First Search



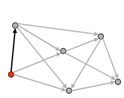
Depth-First Search



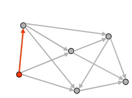
Depth-First Search



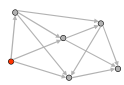
Depth-First Search



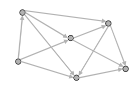
Depth-First Search



Depth-First Search



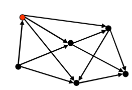
Depth-First Search



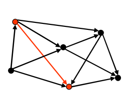
Breadth-First Search

- Same, except use a queue instead of a stack to determine which edge to explore next

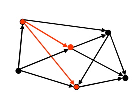
Breadth-First Search



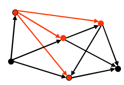
Breadth-First Search



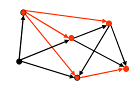
Breadth-First Search



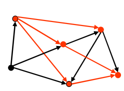
Breadth-First Search



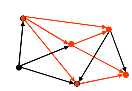
Breadth-First Search



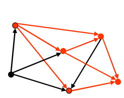
Breadth-First Search



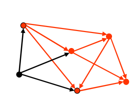
Breadth-First Search



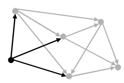
Breadth-First Search



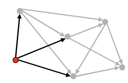
Breadth-First Search



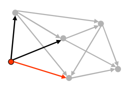
Breadth-First Search



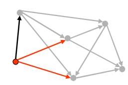
Breadth-First Search



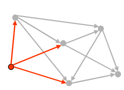
Breadth-First Search



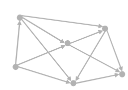
Breadth-First Search



Breadth-First Search



Breadth-First Search

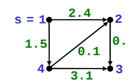


Shortest Paths

Suppose you have a USAir route map with intercity distances. You want to know the shortest distance from Ithaca to every city served by USAir.

This is known as the *single-source shortest path problem*.

Shortest Paths



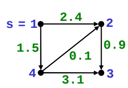
Digraph with edge weights

	1	2	3	4
1	0	2.4	∞	1.5
2	∞	0	0.9	∞
3	∞	∞	0	∞
4	∞	0.1	3.1	0

Corresponding matrix

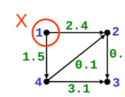
Single-source shortest path problem: Given a graph with edge weights w(u,v) and a designated vertex s, find the shortest path from s to every other vertex (length of a path = sum of edge weights)

Shortest Paths



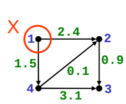
- Let d(s,u) denote the distance (length of shortest path) from s to u. In this example,
  - d(1,1) = 0
  - d(1,2) = 1.6
  - d(1,3) = 2.5
  - d(1,4) = 1.5

Dijkstra's Algorithm



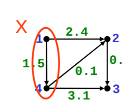
- Let X = {s}
  - X is the set of nodes for which we have already determined the shortest path
- For each node u ∉ X, define D(u) = w(s,u)
  - D(2) = 2.4
  - D(3) = ∞
  - D(4) = 1.5

Dijkstra's Algorithm



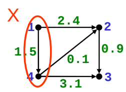
- Find u ∉ X such that D(u) is minimum, add it to X
  - at that point, d(s,u) = D(u)
- For each node v ∉ X such that (u,v) ∈ E, if D(u) + w(u,v) < D(v), set D(v) = D(u) + w(u,v)
  - D(2) = 2.4
  - D(3) = ∞
  - D(4) = 1.5

Dijkstra's Algorithm



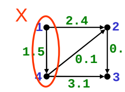
- Find u ∉ X such that D(u) is minimum, add it to X
  - at that point, d(s,u) = D(u) u = 4
- For each node v ∉ X such that (u,v) ∈ E, if D(u) + w(u,v) < D(v), set D(v) = D(u) + w(u,v)
  - D(2) = 2.4
  - D(3) = ∞
  - D(4) = 1.5 = d(1,4)

Dijkstra's Algorithm



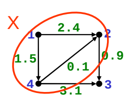
- Find u ∉ X such that D(u) is minimum, add it to X
  - at that point, d(s,u) = D(u) u = 4
- For each node v ∉ X such that (u,v) ∈ E, if D(u) + w(u,v) < D(v), set D(v) = D(u) + w(u,v)
  - D(2) = ~~2.4~~ 1.6
  - D(3) = ~~∞~~ 4.6
  - D(4) = 1.5 = d(1,4)

Dijkstra's Algorithm



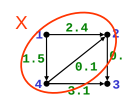
- Find u ∉ X such that D(u) is minimum, add it to X
  - at that point, d(s,u) = D(u)
- For each node v ∉ X such that (u,v) ∈ E, if D(u) + w(u,v) < D(v), set D(v) = D(u) + w(u,v)
  - D(2) = ~~2.4~~ 1.6
  - D(3) = ~~∞~~ 4.6
  - D(4) = 1.5 = d(1,4)

Dijkstra's Algorithm



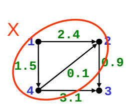
- Find u ∉ X such that D(u) is minimum, add it to X
  - at that point, d(s,u) = D(u) u = 2
- For each node v ∉ X such that (u,v) ∈ E, if D(u) + w(u,v) < D(v), set D(v) = D(u) + w(u,v)
  - D(2) = ~~2.4~~ 1.6 = d(1,2)
  - D(3) = ~~∞~~ 4.6
  - D(4) = 1.5 = d(1,4)

Dijkstra's Algorithm



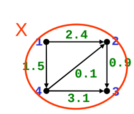
- Find u ∉ X such that D(u) is minimum, add it to X
  - at that point, d(s,u) = D(u) u = 2
- For each node v ∉ X such that (u,v) ∈ E, if D(u) + w(u,v) < D(v), set D(v) = D(u) + w(u,v)
  - D(2) = ~~2.4~~ 1.6 = d(1,2)
  - D(3) = ~~∞~~ ~~4.6~~ 2.5
  - D(4) = 1.5 = d(1,4)

Dijkstra's Algorithm



- Find u ∉ X such that D(u) is minimum, add it to X
  - at that point, d(s,u) = D(u)
- For each node v ∉ X such that (u,v) ∈ E, if D(u) + w(u,v) < D(v), set D(v) = D(u) + w(u,v)
  - D(2) = ~~2.4~~ 1.6 = d(1,2)
  - D(3) = ~~∞~~ ~~4.6~~ 2.5
  - D(4) = 1.5 = d(1,4)

Dijkstra's Algorithm



- Find u ∉ X such that D(u) is minimum, add it to X
  - at that point, d(s,u) = D(u) u = 3
- For each node v ∉ X such that (u,v) ∈ E, if D(u) + w(u,v) < D(v), set D(v) = D(u) + w(u,v)
  - D(2) = ~~2.4~~ 1.6 = d(1,2)
  - D(3) = ~~∞~~ ~~4.6~~ 2.5 = d(1,3)
  - D(4) = 1.5 = d(1,4)

Dijkstra's Algorithm

Proof of correctness – show that the following are invariants of the loop:

- For u $\in$ X, D(u) = d(s,u)
- For u $\in$ X and v $\notin$ X, d(s,u) $\leq$ d(s,v)
- For all u, D(u) is the length of the shortest path from s to u such that all nodes on the path (except possibly u) are in X

Implementation:

- Use a priority queue for the nodes not yet taken – priority is D(u)

Complexity

- Every edge is examined once when its source is taken into X
- A vertex may be placed in the priority queue multiple times, but at most once for each incoming edge
- Number of insertions and deletions into priority queue = m + 1, where m = |E|
- Total complexity = O(m log m)

Conclusion

- There are faster but much more complicated algorithms for single-source, shortest-path problem that run in time O(n log n + m) using something called *Fibonacci heaps*
- It is important that all edge weights be nonnegative – Dijkstra's algorithm does not work otherwise, we need a more complicated algorithm called *Warshall's algorithm*
- Learn about this and more in CS482