Graphs

CPE 349

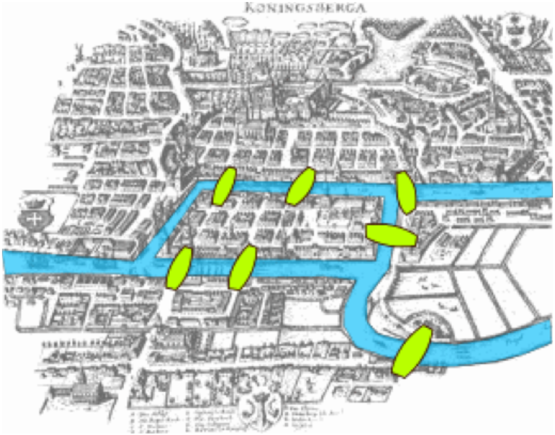
Theresa Migler-VonDollen

Graph Theory

- Father of graph theory: Leonhard Euler

- Swiss mathematician
- *Seven Bridges of Königsberg* 1736.

Seven Bridges of Königsberg



Is there a walk that traverses each bridge exactly once?

What is a graph?

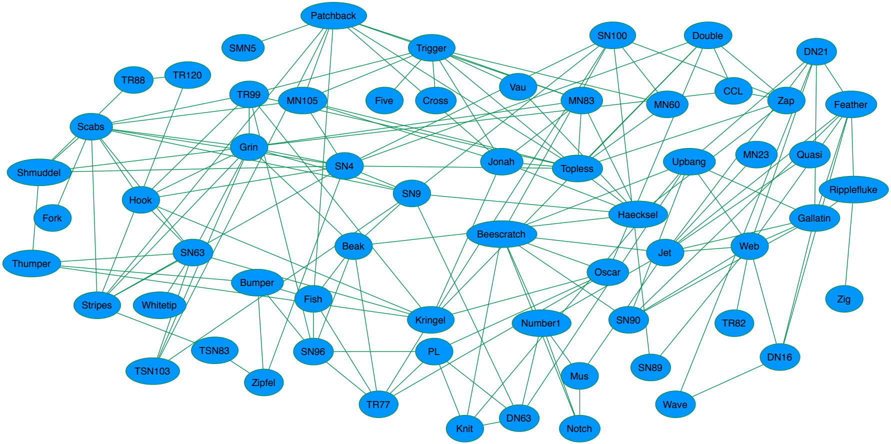
- Vertices and edges.
- Nodes and links.
- People and relationships.

Seven Bridges of Königsberg

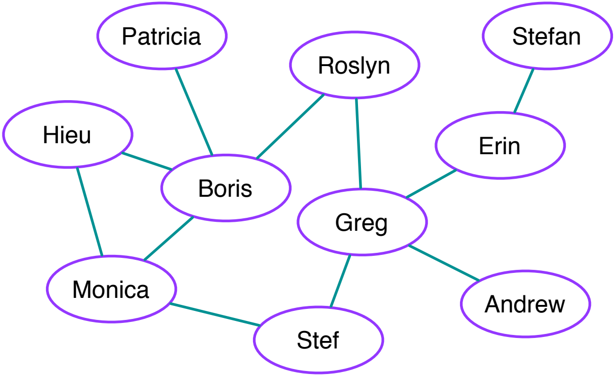
Theorem

There is a walk through a graph that traverses each edge exactly once if and only if the graph is connected and there are exactly two or zero vertices of odd degree.

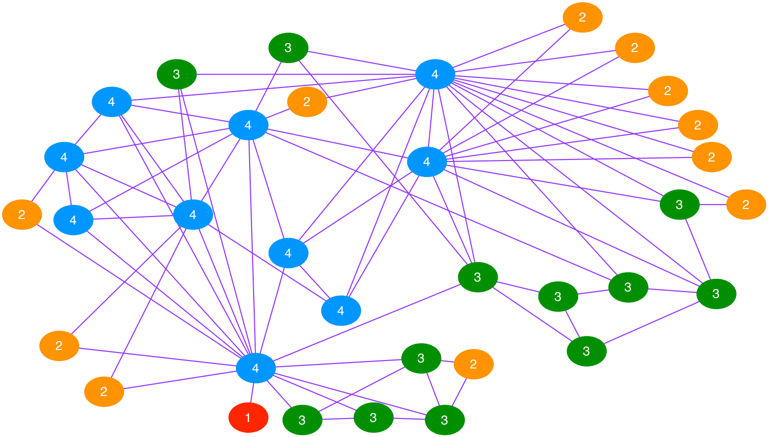
What is a graph? - Dolphin network



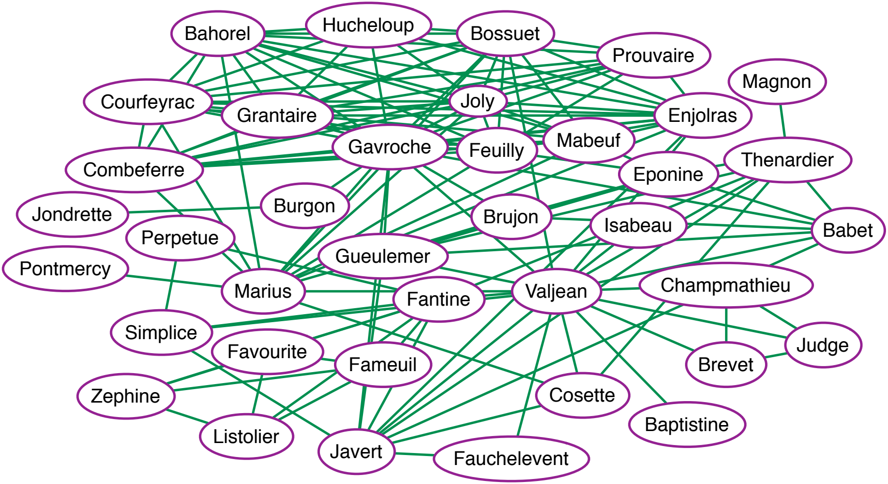
What is a graph? - Friends network



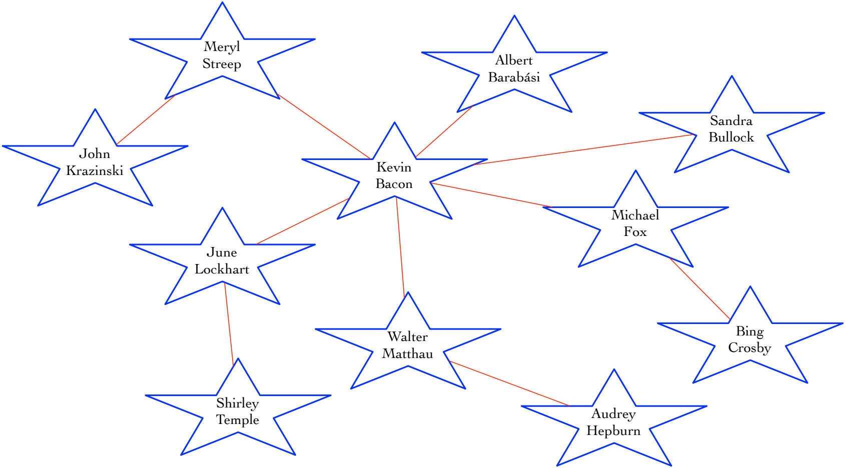
What is a graph? - Karate network



What is a graph? - Les Misérables network



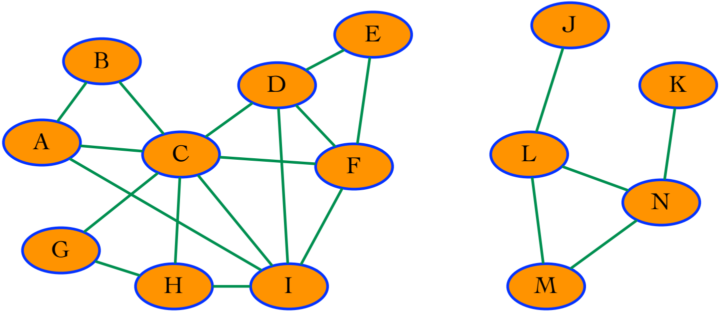
What is a graph? - Actors network



Types of networks

- Collaboration networks
- Who-talks-to-whom graphs
- Information linkage graphs
- Technological networks
- Biological networks

Graph Basics



Definition

A vertex A and a vertex B are *neighbors* if there is an edge, AB, between A and B.

D is neighbors with E, F, and C, but not B.

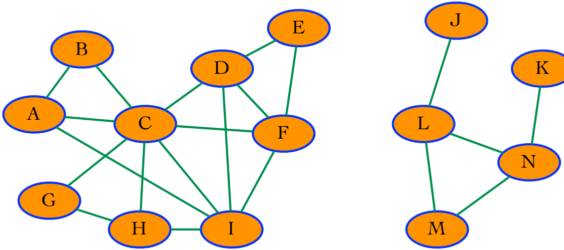
Basic Graph Representations

There are two basic ways to represent a graph, $G = (V, E)$, $V = \{v_1, v_2, \ldots v_n\}$:

1. An *adjacency matrix* is an $n \times n$ array where the (i, j) entry is:
   - $a_{ij} = 1$ if there is an edge from v_i to v_j.
   - $a_{ij} = 0$ otherwise.
2. An *adjacency list* is a set of n linked lists, one for each vertex.
   - The linked list for vertex v holds the names of all vertices, u, such that there is an edge from v to u.

- What is the size of each data structure?
- How long does it take to find a particular edge for each data structure?

Basic ways to describe a graph

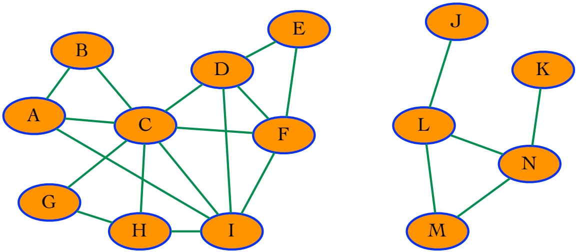


Definition

The *degree* of a vertex is the number of edges adjacent to it (or the number of neighbors).

C has degree 7. J has degree 1.

Graph Basics

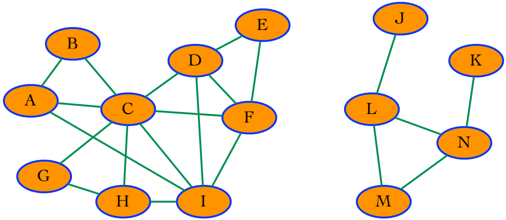


Definition

The *degree distribution* of a graph is the number of vertices of each degree.

$\{0, 2, 4, 4, 2, 1, 0, 1\}$ or $\{0, 1/7, 2/7, 2/7, 1/7, 1/14, 0, 1/14\}$

Graph Basics



Definition

A *path* between two vertices is a sequence of vertices with the property that each consecutive pair in the sequence is connected by an edge.

There are many paths connecting A and E.
One of these is A, C, D, E, another is A, B, C, G, H, I, F, E.
A, D, E is not a path connecting A and E.

Reachability

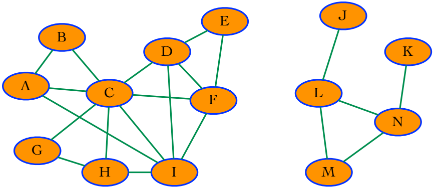
Definition

A vertex u is *reachable* from a vertex v if there is a path from v to u.

Reachability

Input: A graph, $G = (V, E)$, and a vertex, $v \in V$.
Goal: A list of all vertices reachable from v.

Which vertices are reachable from D?



Reachability

Reachability

Input: A graph, $G = (V, E)$, and a vertex, $v \in V$.
Goal: A list of all vertices reachable from v.

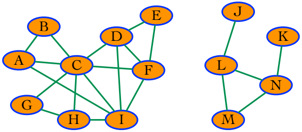
explore(G, v)

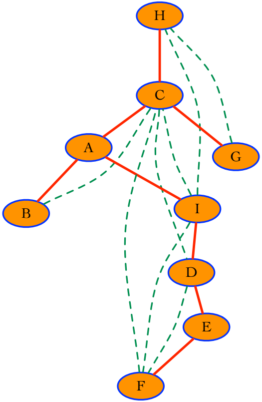
Input: A graph G and a vertex v.
Output: Vertices labeled "discovered" are vertices reachable from v.

```
discovered(v) = true.
for all neighbors of v, u do
    if discovered(u) = false then
        explore(G, u)
```

Reachability

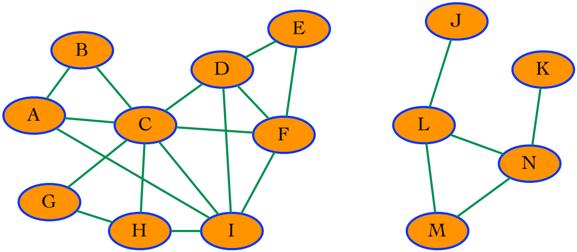
Example: explore(*Graph*, *H*)





- We call the red edges "tree edges".
- We call the dotted edges "back edges".

Graph Basics

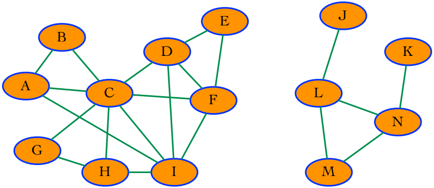


Definition

We say that a graph is *connected* if for each pair of vertices, there is a path between them.

The above graph is not connected.

Graph Basics



Definition

A *connected component* (or just *component*) of a graph is a subset of vertices such that every vertex in the subset has a path to every other vertex in the subset and the subset is not a part of some larger subset with the property that there is a path between every pair of vertices.

There are two components in the graph $A, B, C, D, E, F, G, H, I$ and J, K, L, M, N. Note that L, M, N is not a component.

Depth-First Search

What if we want to visit all connected components of a graph?

DFS(G)

Input: A graph $G = (V, E)$.
Output: A *forest* of connected components of G.

```
for all v ∈ V do
    discovered(v) = false
for all v ∈ V do
    if discovered(v) = false then
        explore(G, v)
```

- Is the algorithm correct?
- What is the running time?

Running Time for DFS

DFS(G)

Input: A graph $G = (V, E)$.
Output: A *forest* of connected components of G.

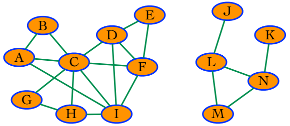
```
1: for all v ∈ V do
2:     discovered(v) = false
3: for all v ∈ V do
4:     if discovered(v) = false then
5:         explore(G, v)
```

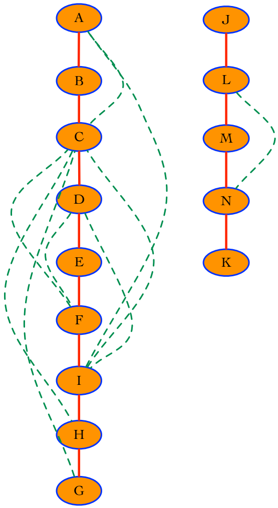
- Step 1 takes $|V|$ time.
- We call explore(G, v) $|V|$ times (once for each vertex).
- In explore, we examine all neighbors of a vertex, so we examine each edge (twice), $2|E|$.

The time complexity is $2|V| + 2|E| = O(|V| + |E|) = O(n + m)$

Depth-First Search

Example: DFS(*Graph*)





Depth-First Search - Versatile

- We could label each connected component by assigning a label each time explore is called in DFS.
- We could note when we visit and leave each vertex with pre- and post-orderings.

---

previsit(v)

1: pre[v]= clock
2: clock = clock + 1

---

postvisit(v)

1: post[v]= clock
2: clock = clock + 1

Depth-First Search

Consider the explore algorithm with pre- and postorderings.

---

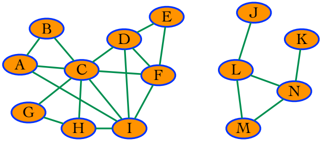
explore(G, v)

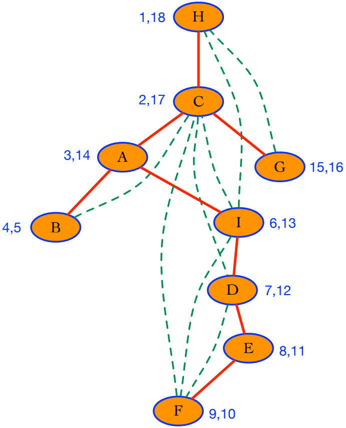
Input: A graph G and a vertex v.
Output: Vertices labeled "discovered" are vertices reachable from v.

discovered(v) =true.
previsit(v)
for all neighbors of v, u **do**
    if discovered(u) =false **then**
        explore(G, u)
postvisit(v)

Depth-First Search

Example: explore(G, H)



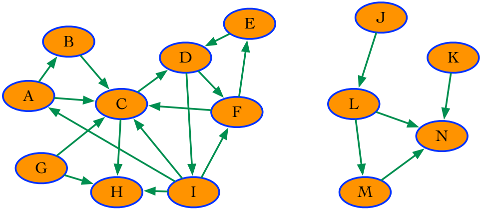


Directed Graphs

What if we want to imply one directional relationships?

- Family trees
- Sewage networks
- Food webs
- Webpage network
- Epidemiological networks...

Graph Basics



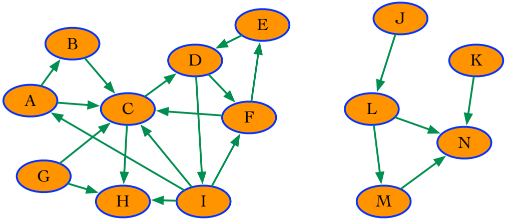
Here $(F, C) \in E$ but $(C, F) \notin E$.

Definition

The *indegree* of a vertex, v, in a directed graph is the number of edges directed into v.
The *outdegree* of a vertex, v, in a directed graph is the number of edges directed out of v.

The indegree of I is 1. The outdegree of I is 4.

Graph Basics



Definition

A *path* in a directed graph from a vertex x to a vertex y is a sequence of vertices with the property that each consecutive pair in the sequence is connected with an edge and all edges are directed in the same direction (out of x).

There is a path from G to E (G, C, D, F, E). There is not a path from H to D.

Depth-First Search in Directed Graphs

The algorithm runs with one small change to explore:

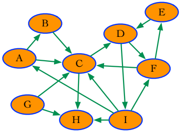
explore(G, v)

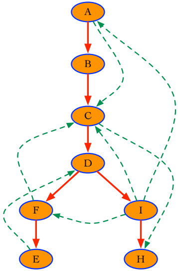
Input: A **directed** graph G and a vertex v.
Output: Vertices labeled "discovered" are vertices reachable from v.

discovered(v) $=$true.
for all **outgoing** neighbors of v, u **do**
    if discovered(u) $=$false **then**
        explore(G, u)

Depth-First Search in Directed Graphs

Example: explore(G, A):

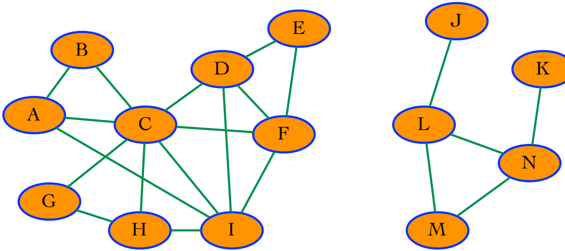




There are four types of edges:

- *Tree edges*
- *Forward edges* - Lead to a nonchild descendent.
- *Back edges* - Lead to an ancestor in the tree.
- *Cross edges* - Lead to neither descendant or ancestor.

Graph Basics

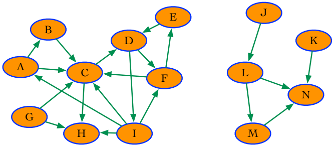


Definition

A *cycle* (in an undirected graph) is a path with at least 3 edges in which the first and last vertices are the same, but otherwise all vertices are distinct.

L, M, N is a cycle, so is A, C, F, I, and many more...

Graph Basics



Definition

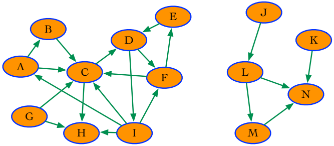
A *cycle* in a directed graph is a (directed) path with at least 2 edges in which the first and last vertices are the same, but otherwise all vertices are distinct.

A, B, C, D, I is a cycle. C, D, E, F, I is not a cycle.

Theorem

A directed graph has a cycle if and only if its DFS tree has a back edge.

Graph Basics

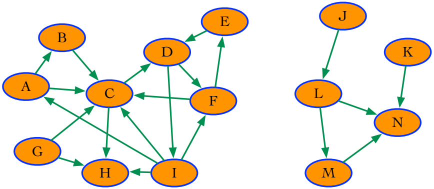


Definition

Two vertices, x and y, are *connected* in a directed graph if there is a path from x to y and y to x.

A and D are connected. L and M are not.

Graph Basics



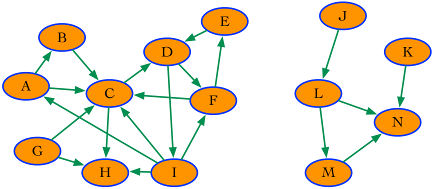
Definition

A directed graph, $G = (V, E)$ is *strongly connected* if for all pairs of vertices $u, v \in V$, u and v are connected.

Definition

The *strongly connected components* of a directed graph partition the graph into strongly connected subgraphs.

Graph Basics



Definition

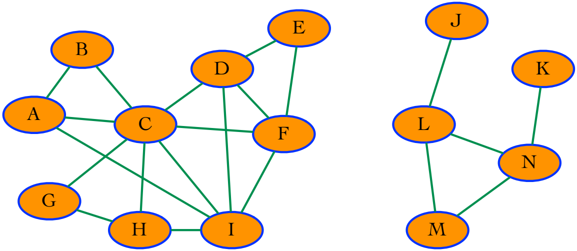
A *directed acyclic graph* or *DAG* is a directed graph with no cycles.

Theorem

Every directed graph is a DAG of its strongly connected components.

We can find such a decomposition in linear time...

Graph Basics



Definition

The *distance* between two vertices is the length of the shortest path connecting them.

The distance between A and F is 2.
By convention, the distance between H and K is ∞.

Calculating Distance

Calculating Distance in a Graph

Input: An undirected graph, $G = (V, E)$, and a vertex $v \in V$.
Goal: Return the distance from v to every other vertex in G.

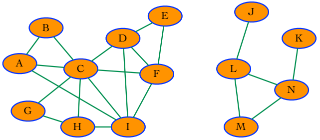
Breadth-First Search

BFS(G, A)

Input: An undirected graph, $G = (V, E)$, and a vertex, A.
Output: For all vertices, X, $dist(X)$ is set to be the distance from A to X.

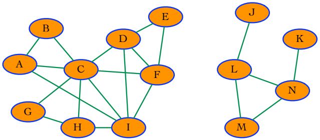
```
for all X ∈ V do
    dist(X) = ∞
dist(A) = 0
Q = [A] (a queue containing A)
while Q is not empty do
    X = dequeue(Q)
    for all edges (X, Y) ∈ E do
        if dist(Y) = ∞ then
            enqueue(Q, Y)
            dist(Y) = dist(X) + 1
```

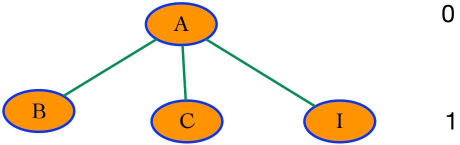
Breadth-First Search



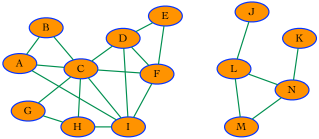
0

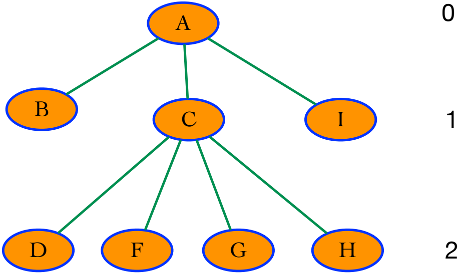
Breadth-First Search



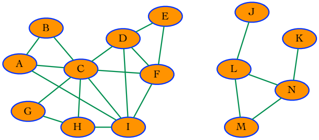


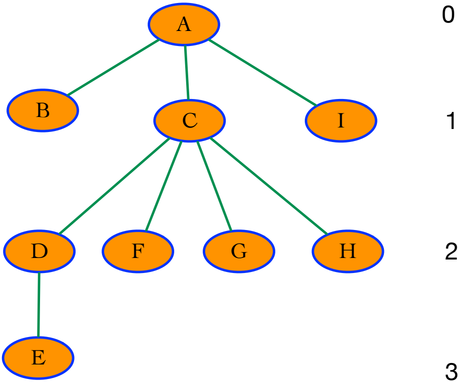
Breadth-First Search





Breadth-First Search





Running Time for BFS

BFS(G, A)

Input: A graph $G = (V, E)$ and a vertex A.
Output: For all vertices, X, reachable from A, $dist(X)$ is set to be the distance from A to X.

```
1:  for all X ∈ V do
2:      dist(X) = ∞
3:  dist(A) = 0
4:  Q = [A] (a queue containing A)
5:  while Q is not empty do
6:      X = dequeue(Q)
7:      for all edges (X, Y) ∈ E do
8:          if dist(Y) = ∞ then
9:              enqueue(Q, Y)
10:             dist(Y) = dist(X) + 1
```

- Step 1 takes $|V|$ time.
- Each vertex gets placed in the queue exactly once. $|V|$ time.
- In Step 7, we examine all neighbors of a vertex, so we examine each edge (twice), $2|E|$.

The time complexity is $2|V| + 2|E| = O(|V| + |E|) = O(n + m)$

Weighted Shortest Paths

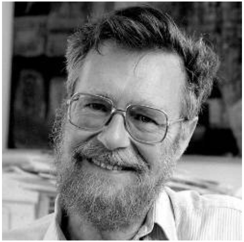
We used Breadth-First search to find shortest paths in graphs where the edges have unit length.
How can we handle the same problem in weighted graphs?

Shortest Paths in Weighted Graphs

Input: A graph, G, where each edge, e, has length, ℓ_e (a positive integer), and a vertex, v, in G.
Goal: Find shortest paths from v to every other vertex in the graph.

Any ideas?

Dijkstra's Algorithm



- Edsger W. Dijkstra (1930 - 2002) was a Dutch computer scientist.
- Received the Turing Award in 1972.
- Shaped computer science as we know it.
- Known for his algorithm for shortest paths, dining philosophers problem, and many others.

Dijkstra's Algorithm

Dijkstra(G, v)

Input: A graph, G, where each edge, e, has length, ℓ_e (a positive integer), and a vertex, v, in G.

Output: For all vertices, u, reachable from v, $dist(u)$ is set to the distance from v to u.

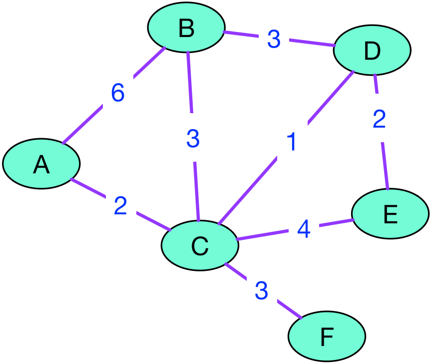
for all $u \in V$ **do**
    $dist(u) = \infty$, and $prev(u) = nil$
$dist(v) = 0$
$H = makequeue(V)$
while $H \neq \emptyset$ **do**
      $x = deletemin(H)$
    for all edges $(x, y) \in E$ **do**
        if $dist(y) > dist(x) + \ell_{(x,y)}$ **then**
            $dist(y) = dist(x) + \ell_{(x,y)}$
            $prev(y) = x$
            $decreasekey(H, y)$

Priority Queues

This data structure maintains a set of vertices with associated key values and supports the following operations:

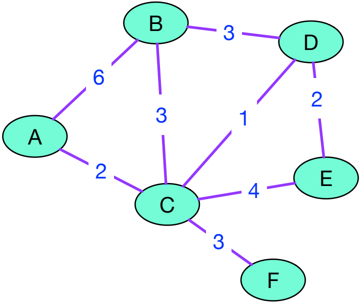
- *Insert* Add a new element to the set.
- *Decrease-key* Accomodate the decrease in key value of a particular element.
- *Delete-min* Return the element with the smallest key, and remove it from the set.
- *Make-queue* Build a priority queue out of the given elements, with the given key values.

Dijkstra's Algorithm



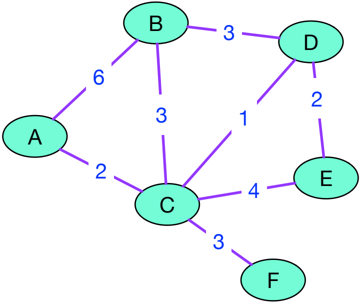
Dijkstra's Algorithm

A	B	C	D	E	F
0	∞	∞	∞	∞	∞



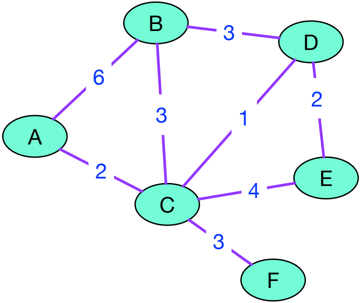
Dijkstra's Algorithm

A	B	C	D	E	F
0	6	2	∞	∞	∞



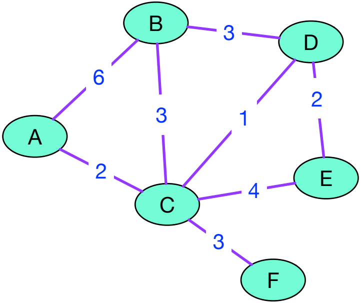
Dijkstra's Algorithm

A	B	C	D	E	F
0	5	2	3	6	5



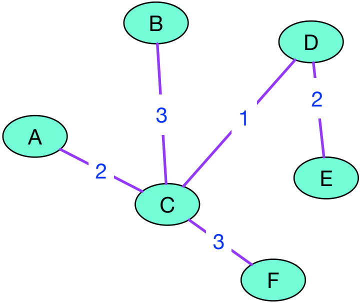
Dijkstra's Algorithm

A	B	C	D	E	F
0	5	2	3	5	5



Dijkstra's Algorithm

A	B	C	D	E	F
0	5	2	3	5	5



Traveling Salesperson Problem

Definition

A *Hamiltonian path* is a path in a graph that visits each vertex exactly once.
A *Hamiltonian cycle* is a Hamiltonian path that is a cycle.

Traveling Salesperson Problem

Input: A complete weighted graph.
Goal: Return a Hamiltonian cycle with smallest weight.

Traveling Salesperson Problem

Input: A list of cities and the distances between each pair of cities.
Goal: Return the shortest possible route that visits each city exactly once and returns to the origin city.

Traveling Salesperson Problem

- This is an NP-hard problem (will discuss this more later).
- This problem was first formulated (mathematically) in 1930.
  - It was first stated in a handbook for traveling salesmen in Germany in 1832.
- It has a number of applications:
  - School bus routes in a school district.
  - Farm distribution.
  - DNA sequencing.

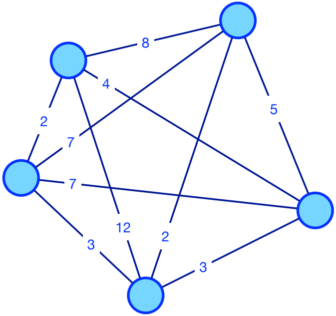
Traveling Salesperson Problem

Traveling Salesperson Problem

Input: A complete weighted graph.

Goal: Return a Hamiltonian cycle with smallest weight.

Example:

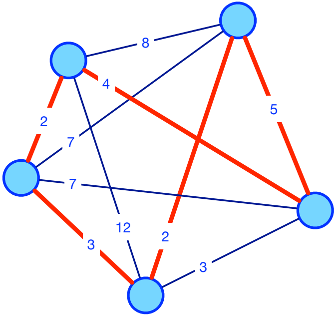


Traveling Salesperson Problem

Traveling Salesperson Problem

Input: A complete weighted graph.
Goal: Return a Hamiltonian cycle with smallest weight.

Example:



Traveling Salesperson Problem

Traveling Salesperson Problem

Input: A complete weighted graph.
Goal: Return a Hamiltonian cycle with smallest weight.

TSP - Brute Force

1. List all possible Hamiltonian cycles.
2. Calculate the weight of each cycle.
3. Choose the cycle with least weight.

This is certainly correct. But it is slow. How slow?

- If there are n vertices, the number of Hamiltonian cycles is $n!$.

Traveling Salesperson Problem

Traveling Salesperson Problem

Input: A complete weighted graph.
Goal: Return a Hamiltonian cycle with smallest weight.

TSP - Nearest Neighbor

1: Start at an arbitrary "home" vertex.
2: At each vertex, choose the nearest unvisited neighbor. In case of a tie, pick at random.
3: End at the home vertex.

Is this correct?

Traveling Salesperson Problem

Traveling Salesperson Problem

Input: A complete weighted graph.
Goal: Return a Hamiltonian cycle with smallest weight.

Can you think of any other algorithms?

- Correct AND
- Efficient

NO!

- This problem is NP-hard.

There is however a very efficient approximation algorithm, using the minimum spanning tree.

- We will create this approximation algorithm at the end of the quarter.

Proof Practice with Graphs

Theorem

Suppose G is a simple graph on n vertices. If G has $n-1$ edges and no cycles then G is connected.

Direct Proof: $P \Rightarrow Q$

Assume P

$\dots$

Therefore, Q.

Thus $P \Rightarrow Q$.

Proof Practice with Graphs

Theorem

Suppose G is a simple graph on n vertices. If G has $n-1$ edges and no cycles then G is connected.

Proof.

- Suppose G has no cycles and $n-1$ edges.
  - Because G has no cycles, G is a forest.
- Let k be the number of components (trees) of G.
  - Every component is a tree and therefore has one fewer edges than vertices.
- The number of edges in G is $n-k$, so $n-k=n-1$, $k=1$.
- G has exactly one component and therefore is connected.

□

Proof Practice with Graphs

Theorem

If T is a tree on 2 or more vertices, then T has at least one vertex of degree 1.

Contraposition: $P \Rightarrow Q$

Assume $\sim Q$

$\ldots$

Therefore, $\sim P$.

Therefore, $\sim Q \Rightarrow \sim P$ Thus $P \Rightarrow Q$.

Proof Practice with Graphs

Theorem

If T is a tree on 2 or more vertices, then T has at least one vertex of degree 1.

Proof.

- Suppose T has no vertices of degree 1.
- Starting at any vertex, v, follow a sequence of distinct edges until a vertex repeats.
  - This is possible because the degree of every vertex is at least two, so upon arriving at a vertex for the first time it is always possible to leave the vertex on another edge.
- When a vertex repeats for the first time, we have discovered a cycle.
  - Therefore T is not a tree.

□

Proof Practice with Graphs

Lemma

If there is a unique path between any two vertices, then G is a tree.

Contradiction: $P \Rightarrow Q$

Assume P and $\sim Q$.
$\ldots$
Therefore, something untrue such as Q AND $\sim Q$ or $0 = 1$.
Therefore, $\Rightarrow\Leftarrow$.
Thus $P \Rightarrow Q$.

Proof Practice with Graphs

Lemma

If there is a unique path between any two vertices, then G is a tree.

Proof.

Suppose that in the graph G, there is a unique path between any two vertices. For a contradiction, suppose that G is not a tree (suppose that G has a cycle).

- Any two vertices on the cycle are connected by at least two distinct paths.
- A contradiction.

□

Proof Practice with Graphs

Lemma

If G is a tree, then there is a unique path between any two vertices.

Contradiction: $P \Rightarrow Q$

Assume P and $\sim Q$.
$\ldots$
Therefore, something untrue such as Q AND $\sim Q$ or $0 = 1$.
Therefore, $\Rightarrow\Leftarrow$.
Thus $P \Rightarrow Q$.

Proof Practice with Graphs

Lemma

If G is a tree, then there is a unique path between any two vertices.

Proof.

Suppose G is a tree. For a contradiction, suppose there are two different paths from v to w: $v = v_1, v_2, \ldots, v_k = w$ and $v = w_1, w_2, \ldots, w_\ell = w$.

- Let i be the smallest integer such that $v_i \neq w_i$.
- Let j be the smallest integer greater than or equal to i such that $w_j = v_m$ for some m, which must be at least i. (Since $w_l = v_k$, such an m must exist.)

Then $v_{i-1}, v_i, \ldots, v_m = w_j, w_{j-1}, \ldots, w_{i-1} = v_{i-1}$ is a cycle in G.

- A contradiction.

□

Proof Practice with Graphs

Lemma

If G is a tree, then there is a unique path between any two vertices.

Lemma

If there is a unique path between any two vertices, then G is a tree.

Therefore, we get the following theorem:

Theorem

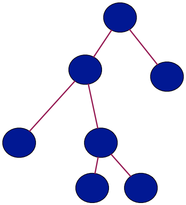
G is a tree if and only if there is a unique path between any two vertices.

Proof Practice with Graphs

Definition

A *full binary tree* is a tree where each vertex other than the leaves has two children.

Example:



Definition

A vertex is a *leaf* if it has no children; otherwise, it is an *internal vertex*.

Proof Practice with Graphs

Theorem

In a full binary tree, G, the number of leaves is exactly one more than the number of internal vertices.

Induction: $(\forall n \in \mathbb{N})$, $P(n)$ is true

1. (Base Case) Show that $P(1)$ is true.
2. (Inductive Hypothesis) Suppose, for all natural numbers, k, that $P(k)$ is true(or, for strong induction, suppose $P(1)$, $P(2)$,...,$P(k)$ are true).
3. (Inductive Step) Show that $P(k+1)$ is true.
4. (Conclusion) By steps 1 and 2 and the PMI, $P(n)$ is true for all $\mathbb{N}$.

Proof Practice with Graphs

Theorem

In a full binary tree, G, the number of leaves is exactly one more than the number of internal vertices.

Proof.

We will proceed by induction on the number of vertices in the tree.

- Let n be the number of vertices in the tree.
- For a tree with n vertices, let $\ell(n)$ be the number of leaves and let $i(n)$ be the number of internal vertices.
  - We want to show that $\ell(n) = 1 + i(n)$.

Proof Practice with Graphs

Theorem

In a full binary tree, G, the number of leaves is exactly one more than the number of internal vertices.

Proof (Cont.)

- **Base Case:** A tree with one vertex $n = 1$, has one leaf vertex and no internal vertices, so $\ell(1) = 1 + i(1)$.
- **Inductive Hypothesis:** Assume the statement is true for all trees with $n \leq k$ vertices.
- **Inductive Step:** Let T be a tree with $k + 1$ vertices.
- Let n_ℓ and n_r be the number of vertices in the left and right subtrees, respectively.
- Since $k + 1 = n_\ell + n_r + 1$, we know that $n_\ell \leq k$ and $n_r \leq k$
  - We can apply the inductive hypothesis to each of these subtrees: $\ell(n_\ell) = 1 + i(n_\ell)$ and $\ell(n_r) = 1 + i(n_r)$.

Proof Practice with Graphs

Theorem

In a full binary tree, G, the number of leaves is exactly one more than the number of internal vertices.

Proof (Cont.)

- The number of leaves of T is the sum of the number of leaves in each subtree: $\ell(k+1) = \ell(n_\ell) + \ell(n_r)$.
- Substituting the previous two equations in: $\ell(k+1) = 2 + i(n_r) + i(n_\ell)$.
- The number of internal vertices of T is the sum of the number of internal vertices of each subtree plus one (for the root of T): $i(k+1) = i(n_\ell) + i(n_r) + 1$.
- Substituting into the previous equation: $\ell(k+1) = i(k+1) + 1$.

□