

DATA TRU

Graphs (but not pie charts)

Final Project: Phase #3 due Tuesday, May 1
 Problem Set: Assignment #7 due Thursday, May 3

U - 1

DATA TRU A Familiar Place...

U - 2

DATA TRU Graph G : Formal Definition

- A **graph** G consists of two sets $G = \{V, E\}$
 - A set V of vertices, or nodes (**entities**)
 - A set $E \subseteq V \times V$ of edges (**relationships** between entities)
- A **subgraph**
 - Consists of a subset of a graph's vertices and a subset of its edges
- **Adjacent vertices**
 - Two vertices that are joined by an edge

U - 3

DATA TRU Paths and Cycles

- A **path** between two vertices
 - A sequence of edges that begins at the first vertex and ends at the other vertex
 - May pass through the same vertex more than once
- A **simple path**
 - A path that passes through a vertex only once
- A **cycle**
 - A path that begins and ends at the same vertex
- A **simple cycle**
 - A cycle that does not pass through a vertex more than once

U - 4

Connected and Complete

- A **connected** graph
 - A graph that has a path between each pair of distinct vertices
- A **disconnected** graph
 - A graph that has at least one pair of vertices without a path between them
- A **complete** graph
 - A graph that has an edge between each pair of distinct vertices

U - 5

Directed Graphs and DAGs

- **Directed** graph
 - Each edge is a directed edge, or an **arc**
 - Can have two arcs between a pair of vertices, one in each direction
 - Vertex y is adjacent to vertex x iff there is a directed edge from x to y
- **Directed path**
 - A sequence of directed edges between two vertices
- **Directed Acyclic Graph (DAG)**
 - Directed graph that has no cycles

How few edges can you remove to make the graph a DAG?

U - 6

Weighted and Multiple Edges

- **Weighted graph**
 - A graph whose edges have numeric labels
 - Usually label correspond to the "cost" of the relationship represented by the edge
- **Multigraph**
 - Not a graph
 - Allows multiple edges between vertices
 - Multiple edges indicate multiple relations between vertices

U - 7

It's Déjà Vu All Over Again

A **tree** is a graph in which there is exactly one simple path connecting any two nodes

How many edges does a tree with n nodes have?

U - 8

DATA TRU **Graphs as ADTs: Operations**

- Create an empty graph
- Determine whether a graph is **empty**
- Determine the number of **vertices** in a graph
- Determine the number of **edges** in a graph
- Determine whether an **edge exists** between two given vertices; for weighted graphs, return **weight** value
- Insert a vertex** in a graph whose vertices have distinct search keys that differ from the new vertex's search key
- Insert an edge** between two given vertices in a graph
- Delete a vertex** from a graph and any edges between the vertex and other vertices
- Delete the edge** between two given vertices in a graph
- Retrieve** from a graph the vertex that contains a given search key

U - 9

DATA TRU **public interface Graph {**

```

public Graph() // Create an empty graph
public boolean isEmpty() // returns true iff a graph is empty
public int numVertices() // returns the number of vertices in a graph
public int numEdges() // returns the number of edges in a graph
public boolean isEdge(Vertex v1, Vertex v2) // returns true iff an edge exists
between two given vertices; for weighted graphs, return weight value

public boolean addVertex(Vertex v)
// Insert a vertex in a graph, return true if successful

public boolean addEdge(Vertex v1, Vertex v2)
// Insert an edge between two given vertices in a graph

public boolean addEdge(Vertex v1, Vertex v2, int weight)
// Insert a weighted edge between two given vertices in a graph

public int getEdgeWeight(Vertex v1, Vertex v2)
// returns the weight of the edge between the two vertices

public boolean deleteVertex(Vertex v)
// Deletes a vertex from a graph and any edges between the vertex and
other vertices, returns true if successful

public boolean deleteEdge(Vertex v1, Vertex v2)
// Deletes the edge between two given vertices in a graph

public Vertex getAdjacentVertices(String key)
// Retrieve from a graph the vertex that contains a given search key17-10
    
```

DATA TRU **Implementing Graphs: Adjacency Matrix**

- Adjacency matrix for a graph with n vertices numbered $0, 1, \dots, n - 1$
 - An $n \times n$ array matrix such that $matrix[i][j] =$
 - 1 (true) iff there is an edge from vertex i to vertex j
 - 0 (false) iff there is no edge from vertex i to vertex j

	a	b	c	d
a	0	0	1	1
b	0	0	1	0
c	1	1	0	0
d	0	0	1	0

What can you deduce about the matrix of an undirected graph? U - 11

DATA TRU **Adjacency Weighted Matrix**

- Adjacency matrix for a weighted graph with n vertices numbered $0, 1, \dots, n - 1$
 - An $n \times n$ array matrix such that $matrix[i][j] =$
 - The weight label of the edge from vertex i to vertex j iff there is an edge from i to j
 - ∞ iff there is no edge from vertex i to vertex j

	a	b	c	d
a	0	∞	4	5
b	∞	0	8	∞
c	4	8	0	2
d	7	∞	2	0

U - 12

Implementing Graphs: Adjacency Lists

- An adjacency list for a graph with n vertices numbered $0, 1, \dots, n - 1$
 - Consists of n linked lists
 - The i^{th} linked list has a list entry for vertex j iff the graph contains an edge from vertex i to vertex j

U - 13

Weighted Adjacency Lists

- Adjacency list for a weighted undirected graph
 - Each list entry contains the edge label
 - Treats each edge as if it were two directed edges in opposite directions

U - 14

Adjacency Matrix vs Adjacency Lists

- Adjacency matrix compared with adjacency list
 - Two common operations on graphs
 - $isEdge(v, w)$
Determine whether there is an edge from vertex v to vertex w
 - $getAdjacentVertices(v)$
Return list of all vertices adjacent to a given vertex v
 - Adjacency matrix
 - Supports $isEdge(v, w)$ more efficiently
 - Adjacency list
 - Supports $getAdjacentVertices(v)$ more efficiently
 - Often requires less space than an adjacency matrix

U - 15

Graph Traversals

- A graph-traversal algorithm
 - Visits all the vertices that it can reach
 - Visits all vertices of the graph iff the graph is connected (effectively computing connected components)
 - Connected component
 - The subset of vertices visited during a traversal that begins at a given vertex
 - Could loop indefinitely if a graph contains a loop
 - To prevent this, the algorithm must
 - Mark each vertex during a visit, and
 - Never visit a vertex more than once

Does it remind you of another algorithm we have seen? U - 16

DATA TRU DepthFirstSearch(*v*)

```
// traverses a graph starting at
// vertex v using DFS
Create a new stack S
Push v onto S
Mark v as visited
While (S is not empty)
  If there are no unvisited vertices
  adjacent to the vertex on top of S:
    Pop S
  Else:
    Select an unvisited vertex u adjacent
    to the vertex w on top of S
    Push u onto S
    Mark u as visited
```

Draw the DFS tree
And the contents of the stack

U - 17

DATA TRU BreadthFirstSearch(*v*)

```
// traverses a graph starting at
// vertex v using BFS
Create a new queue Q
enqueue v onto Q
Mark v as visited
While (Q is not empty)
  dequeue a vertex w from Q
  For each unvisited vertex u
  adjacent to w:
    Mark u as visited
    enqueue u onto Q
```

Draw the BFS tree
And the contents of the queue

U - 18

DATA TRU Dependency Graph

- A directed graph, preferably a DAG
- Usually reflect dependencies or requirements
- I.e., Assembly lines, Supply lines, Organizational charts, ...
- Understanding dependencies requires "topological sorting"

U - 19

DATA TRU Applications of Graphs: Topological Sorting

- **Topological order**
 - A list of vertices in a DAG such that vertex *x* precedes vertex *y* iff there is a directed edge from *x* to *y* in the graph
 - There may be several topological orders in a given graph
- **Topological sorting**
 - Arranging the vertices into a topological order

U - 20

Topological Sorting Algorithms

- Select a vertex v that has **no successor**
- Remove v from the graph (along with all edges that lead to it),
- Add v to the beginning of a list of vertices L
- Repeat previous steps
- When the graph is empty, L 's vertices will be in topological order

U - 21

Another Topological Sorting Algorithm

- Select a vertex v that has **no predecessor**
- Remove v from the graph (along with all edges that lead to it),
- Add v to the end of a list of vertices L
- Repeat previous steps
- When the graph is empty, L 's vertices will be in topological order

U - 22

Spanning Trees

- A **tree** is an undirected connected graph without cycles
- A **spanning tree** of a connected undirected graph G
 - A subgraph of G that contains all of G 's vertices and enough of its edges to form a tree

U - 23

Spanning Trees

- To **obtain** a spanning tree from a connected undirected graph with cycles
 - Remove edges (maintaining connectedness) until there are no cycles
- You can **determine** whether a connected graph contains a cycle by counting its vertices and edges
 - A **connected** undirected graph that has n vertices must have at least $n - 1$ edges
 - If it has exactly $n - 1$ edges, it cannot contain a cycle
 - If it has more than $n - 1$ edges, it must contain at least one cycle

How many spanning trees does a complete 4-node graph have? U - 24

DATA TRU DFS and BFS Spanning Tree Algorithms

- Traverse the graph using DFS/BFS and **mark the edges** that you follow
- After the traversal ends, the graph's vertices and marked edges form the spanning tree

U - 25

DATA TRU Minimum Spanning Tree

A spanning tree for which the sum of its edge weights is **minimal**

Formulated by Czech engineer **Boruvka** in 1923

He wanted to find the cheapest way to bring electricity to the Bohemian countryside

U - 26

DATA TRU Minimum Spanning Tree

U - 27

DATA TRU The "Snowball" MST Algorithm

Discovered by **Prim**

- Finds an MST beginning at any vertex
- **Strategy**
 - Find the least-cost edge (v, u) from a visited vertex v to some unvisited vertex u
 - Mark u as visited
 - Add vertex u and edge (v, u) to the MST
 - Repeat the above steps until there are no more unvisited vertices

U - 28

"Parallel Choices" MST Algorithm

Described by **Boruvka**

- Finds an MST beginning at **every** vertex
- **Strategy**
 - For every vertex v select its least-cost edge (v, u) to the remaining vertices
 - Merge connected vertices into a "supervertex"
 - Repeat the above steps until one vertex remains

U - 29

Shortest Paths

- What is the shortest way to go from the Science Center to Tower?
- **Shortest path** between two vertices in a **weighted** graph is the path that has the smallest sum of its edge weights among all the possible paths

U - 30

Dijkstra's Shortest Path Algorithm

- Determines the shortest paths between a given origin o and all other vertices, using:
 - A set `vertexSet` of selected vertices
 - An array `pathWeight[]`, where `pathWeight[v]` is the weight of the shortest path from vertex o to vertex v that passes through vertices in `vertexSet`

U - 31

Circuits (and evening strolls in Königsberg)

- A circuit is a special cycle that passes through every vertex (or edge) in a (multi)graph exactly once
- In Königsberg people wondered if it were possible to go for a walk in the town and cross each bridge **once**
 - Euler circuit: A circuit that begins at a vertex v , passes through **every edge** exactly once, and terminates back at v
 - Euler proved that it exists if and only if ...

U - 32

DATA TRU Drawing Without Lifting Your Pencil

- Which of the following sketches can you trace without lifting your pencil from the paper?

U - 33

DATA TRU Hamiltonian Circuit and Path

- Begin at some vertex v , pass through every vertex exactly once, and terminate back at v (a circuit)

U - 34

DATA TRU Traveling Salesperson Problem

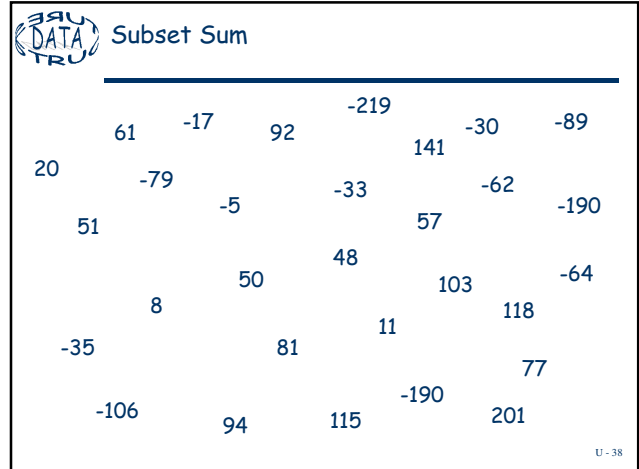
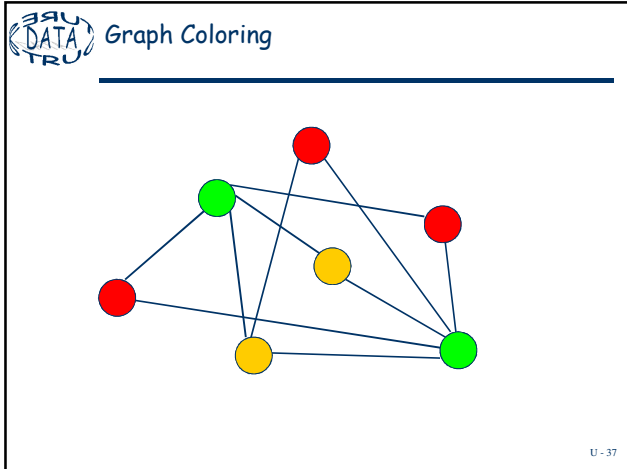
- Of all the Hamiltonian paths, find the cheapest one


U - 35

DATA TRU Complexity: Easy vs Difficult Problems

- Complexity is measured in terms of time required to find a solution
- Easy Problems (can be done in $O(n^k)$, k small)
 - Sorting
 - Inserting, Deleting, Searching
 - BFS and DFS
 - Topological sort
 - Eulerian circuit
- Difficult Problems (seem to require $O(2^n)$)
 - Longest simple cycle problem
 - Subset sum problem
 - The traveling salesperson problem
 - Graph coloring problem

U - 36



 Are They Really That Different?

- Find the *shortest* simple path between two vertices in a graph
- Find the *longest* simple path between two vertices in a graph
- Find a cycle that traverses each *edge* of a graph exactly once (a Euler tour)
- Find a simple cycle that contains every *vertex* in a graph (a Hamiltonian cycle)

U - 39