Data Structures & Algorithms 2

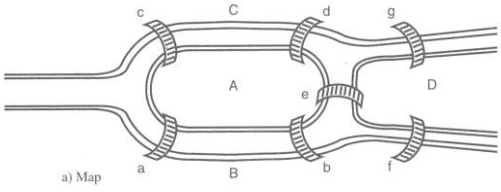
Topic 7 – Graphs

Graphs

- Graphs are one of the most versatile structures used in computer programming and are used to solve the more interesting problems
- There are two types of graphs:
  - Unweighted graphs where the links between nodes are equal
  - Weighted graphs where the links are each associated with an individual weight
- The shape of the graph is dictated by the problem you want to solve and represents some real-world situation

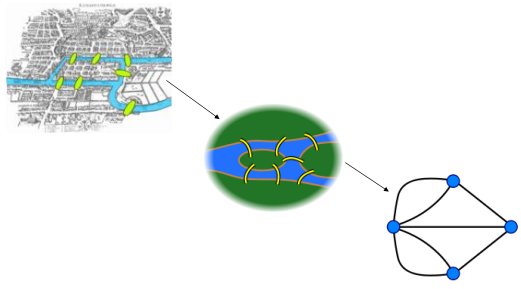
Real-world problems

- Graphs are used to represent real-world problem such as airline routes, electrical circuits and job scheduling
- For example, can you find a way to cross all seven bridges in Konigsberg only once?



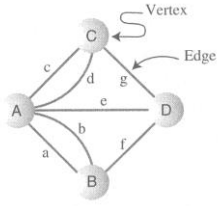
a) Map

Abstraction



Solution

- This problem inspired Leonhard Euler to invent graph theory during the 18th century
- The trick was to represent the bridges in the form of a graph



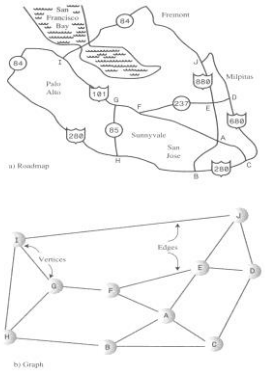
b) Graph

Konigsberg → Kaliningrad

- Birthplace of Christian Goldbach (1690)
- Home of Immanuel Kant (1724)
- Bridges used by Euler to develop graph theory (1736)
- Birthplace of David Hilbert (1862)
- Taken over by Russians – all 7 bridges destroyed (1944)

Graphs

- Nodes are called *vertices*
- Vertices are linked by *edges*
- Vertices are labeled in some way, often with letters
- Each edge has two vertices at its ends
- The graph represents which vertices are connected to each other

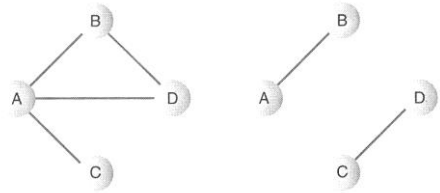


Example

Terminology

- Two vertices are adjacent if they are connected by an edge (e.g. G and F)
- A path is a sequence of edges (e.g. BAEJ and BCDJ are two paths that link the vertices B and J)
- A graph is said to be connected if there is at least one path from every vertex to every other vertex
- The algorithms we will consider all assume a connected graph

Connected Graph



Directed and Weighted

- A non-directed graph means you don't have to go in a particular direction, you can follow an edge in both directions
- Graphs are often used to model situations in which you can go in only one direction along an edge – like a one-way street
- These graphs are called *directed* and the allowed direction is shown as an arrowhead
- In some graphs edges are given a weight that represents factors such as the physical distance between two vertices or the cost / time taken to get from one vertex to another

Representation

- How do we represent graphs in a program?
- First of all we need a vertex class which stores the name of the vertex and whether it was visited or not

```
public Vertex(char label_in){    // constructor

      label = label_in;
      wasVisited = false;

}
```

Representing Edges

- In a binary tree, each node contains a reference to two child nodes
- However, graphs have a more free-form organization where they have an arbitrary number of edges
- We need a special data structure for representing connections between vertices
- Two commonly used methods are:
  - Adjacency matrix
  - Adjacency list

Adjacency Matrix

- The adjacency matrix is a two-dimensional array in which the elements indicate whether an edge is present between two vertices
- If a graph has N vertices then the adjacency matrix is an N x N array
- An edge between two vertices is represented by a 1 versus a 0 (you could also use a boolean)
- If the graph is non-directed then the data will be mirrored along the diagonal of the matrix

Adjacency Matrix

TABLE 13.1 Adjacency Matrix

	A	B	C	D
A	0	1	1	1
B	1	0	0	1
C	1	0	0	0
D	1	1	0	0

Adjacency List

- An adjacency list uses a linked list structure to represent edges
- There is a linked list for each vertices which contains a list of all the other vertices it is linked to
- The order in the linked list has no particular significance
- For a very sparse graph (few edges between vertices) you can speed up runtime by using an adjacency list (because you don't need to examine any 0 entries)
- However, the algorithms involving the adjacency matrix are simpler so we'll use this method

Adjacency List

TABLE 13.2 Adjacency Lists

Vertex	List Containing Adjacent Vertices
A	B—>C—>D
B	A—>D
C	A
D	A—>B

Adding Vertices & Edges

- Adding a new vertex is just like adding a new node

```
vertexList[nVerts++] = new Vertex('F');
```

- To add a new edge just update the adjacency matrix or list

```
adjMat[1][3] = 1;
adjMat[3][1] = 1;
```

Searching

- One of the most useful operations you can perform on your graph is searching
- For example, find all of the towns that can be visited by rail, leaving from Dublin
- There are two very well known approaches for searching a space
  - Depth-first search (DFS)
  - Breadth-first search (BFS)
- Depth-first search explores one possibility as far as it can and then backtracks when it meets a dead end
- Breadth-first search explores all possibilities of the same depth at the same time and spreads itself equally

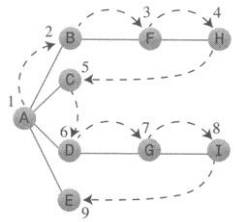
Depth-First search

- Depth-first search uses a stack to remember where it should go when it reaches a dead end
- Pick a starting point then push this vertex onto the stack and mark it so that you won't visit it again
- Next visit any adjacent vertex and start doing the same thing
- If there are no unvisited adjacent vertices then just pop a vertex off the stack and try again (backtrack)

Depth-First search

TABLE 13.3 Stack Contents During Depth-First Search

Event	Stack
Visit A	A
Visit B	AB
Visit F	ABF
Visit H	ABFH
Pop H	ABF
Pop F	AB
Pop B	A
Visit C	AC
Pop C	A
Visit D	AD
Visit G	ADG
Visit I	ADGI
Pop I	ADG
Pop G	AD
Pop D	A
Visit E	AE
Pop E	A
Pop A	
Done	



Analogy

- Depth-first search is like the "ball and wool" approach of finding the exit to a maze
- Any time you come up against a dead end you backtrack and mark the path you've been on so you won't try it again
- Eventually, you will have explored every possible path without retracing your steps
- Depth first search is used when you are trying to find a solution to a problem (like finding the exit to a maze!)

Implementing DFS

- The key to DFS is to be able to find the vertices that are unvisited and adjacent to a specified vertex
- Consult the adjacency matrix for this
- Go to the row for the specified vertex and you can pick out all the columns containing a one
- Check these adjacent vertices to see if they have been visited or not
- Use a stack structure to track the path you've followed so far and flag the vertices that have already been visited so you can avoid them
- Pop off the stack to backtrack

Breadth-First Search

- Depth-first search acts like it wants to get as far away from the starting point as quickly as possible
- Breadth-first search likes to stay as close as possible to the starting point
- It visits all possibilities at the same depth before going any deeper
- It ventures one vertex from the origin, then two, then three etc.
- Breadth-first search can be implemented using a queue instead of a stack

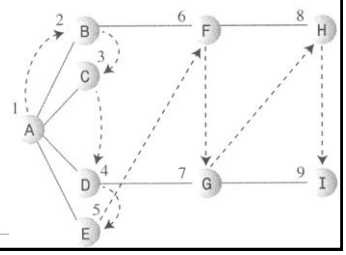
Breadth-First Search

- The rules are:
  - Visit the next unvisited vertex (if there is one) that's adjacent to the current vertex, mark it and add it to the back of the queue
  - Only when you have visited all adjacent vertices, remove a vertex from the queue and make it the current one

Breadth-First Search

TABLE 13.4 Queue Contents During Breadth-First Search

Event	Queue (Front to Rear)
Visit A	
Visit B	B
Visit C	BC
Visit D	BCD
Visit E	BCDE
Remove B	CDE
Visit F	CDEF
Remove C	DEF
Remove D	EF
Visit G	EFG
Remove E	FG
Remove F	G
Visit H	GH
Remove G	H
Visit I	HI
Remove H	I
Remove I	
Done	



Implementing BFS

- Again, check all the adjacent vertices using the adjacency matrix
- Insert them to the back of the queue
- Keep removing one item from the queue and inserting all its adjacent vertices to the back
- This way all the vertices at one particular depth will be explored before you move onto a vertex at the next depth

Analogy

- Breadth-first search is like the ripples spreading out on a pond
- The ripples cover all the search space and move further and further away from the origin at the same rate
- Breadth first search is used when you want to explore all of the possibilities at a given depth

BFS in Games

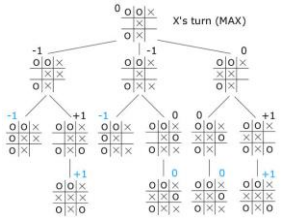
- In a typical board game you can choose one of several possible moves
- Each choice leads to further choices leading to an ever-expanding tree of possibilities
- We can represent each state of the board as a vertex and each choice as an edge leading to another board state
- A smart algorithm will search forward through the possibilities to see what could potentially happen based on each choice

Game trees

- Because you can't predict what move the person you're playing against will make, the trick is to look at every sequence of moves that *could* be made
- A simple chess playing algorithm might look at all the possible game states four moves ahead
- A depth first search to the endgame is not appropriate because we're not trying to find a full solution to the game – we cannot know how the opponent will choose to move
- Instead, we want to take into account all of the possibilities a certain number of moves ahead

Minimax algorithm

- Minimax is a decision rule used in decision theory, game theory and statistics for minimizing the possible loss for a worst case scenario
- We assume the opponent will make the best possible move and choose the option which has the least good best move



Artificial Intelligence

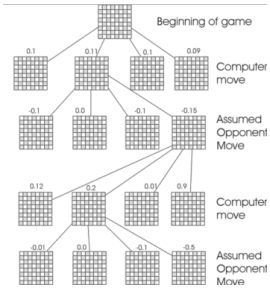
- If the computer knows all the possible game states four moves ahead then it can choose the move that stops you being able to make a good move
- By knowing what can happen up to four moves ahead, the computer opponent is far more informed that the human opponent
- Naïve users will usually only be thinking in terms of "if I make this move, then what moves can the computer make"
- This is only a breadth-first search of depth two

Othello

- Light green squares are possibilities for black to play
- Looking much more than four moves ahead could cause computer to hang due to combinatorial explosion!
- There are potentially trillions of different ways the game could pan out

Board Game Search Tree

- Each state of the board game can lead to several other states
- In chess the branching factor is about 35
- To play master-level chess requires searching about 8 moves ahead so about 2 x 10^{12} options must be examined



Deep Blue

- On May 11, 1997 the Deep Blue computer won a six-game match by two wins to one and three draws against world champion Garry Kasparov
- The computer was capable of evaluating 200 million positions per second
- It could search up to 40 moves ahead
- Kasparov said that he saw deep intelligence and creativity in the machine's moves and was certain that human players had intervened