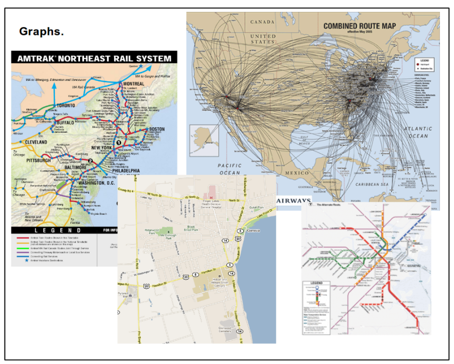
Graphs and
Graph Algorithms

Graphs

For storing relationships between pairs of things.

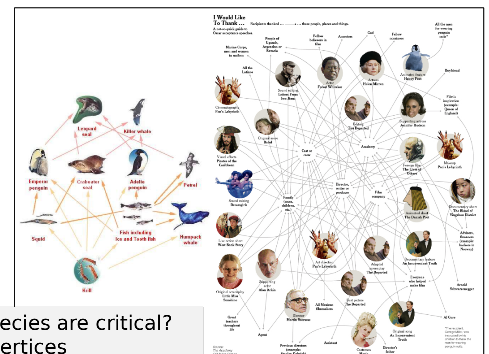
Graphs are extremely important because lots of things can be modeled as graphs, and lots of problems reduce to common graph algorithms.

Graphs



- driving directions (how to travel) from A to B → shortest (distance, time) path from A to B
- can you get from A to B? → reachability

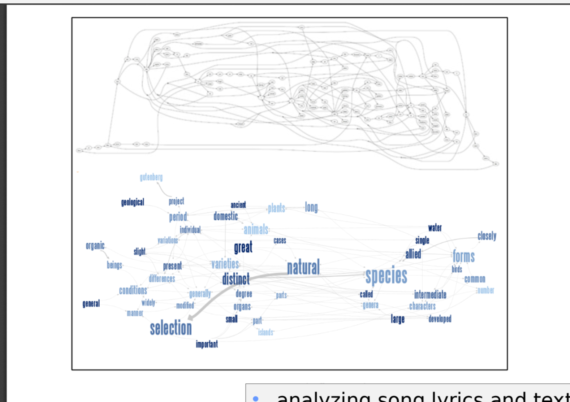
Graphs



- what species are critical? → cut vertices
- what species would be affected by the removal of another? → reachability

- most commonly thanked? → high indegree

Graphs



- analyzing song lyrics and text – connecting consecutive words

Graphs

Formally, a graph G consists of a set of vertices
$V = \{ v_1, v_2, v_3, \dots \}$
and a set of edges that connect pairs of vertices
$E = \{ (u,v) \mid u, v \text{ in } V \}$.

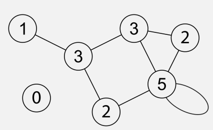
← vertices represent the things

← edges represent the relationships

n is often used to denote the number of vertices (|V|).
m is often used to denote the number of edges (|E|).

Some Graph Terminology

- the vertices u, v of an edge (u,v) are the *endpoints* of the edge
  - an edge is *incident* on its endpoints
- the *degree* of a vertex is the number of incident edges
  - for directed graphs, the *indegree* is the number of incoming edges and the *outdegree* is the number of outgoing edges



an undirected graph with each vertex labeled with its degree

- a *path* is a route from one vertex to another, following edges (in the proper direction, if the edges are directed)
- a *cycle* is a path that starts and ends at the same vertex

Flavors of Graphs

- undirected vs directed
  - does having edge (u,v) imply that edge (v,u) also exists?
  - a *mixed* graph has both directed and undirected edges
- connected vs not connected
  - is there a path between every pair of vertices?
  - minimum number of edges in a connected graph is n-1
- simple vs not simple (self loops, multiedges)
  - a *self loop* is an edge (v,v)
  - *multiedges* occur when there are multiple edges between a pair of vertices (u,v)
  - maximum number of edges in a simple graph is n(n-1)/2 (undirected) or n(n-1) (directed) = $\Theta(n^2)$
- sparse vs dense
  - typically “sparse” means O(n) edges while “dense” means $O(n^2)$

Flavors of Graphs

- cyclic vs acyclic
  - an undirected acyclic graph is a tree
  - a tree has exactly n-1 edges
- weighted vs unweighted
  - associate a value (*weight* or *cost*) with each edge
  - (less common) associate a value with each vertex

Flavors of Graphs

- labeled vs unlabeled
  - do vertices have unique labels to distinguish them from one another?
- embedded vs topological
  - do the vertices and edges have geometric positions, or are elements of the graph structure (such as edges or edge weights) derived from the geometry?
    - e.g. TSP over points in the plane or grids of points where edges connect neighboring points
  - an embedding also means there is a particular order to the edges incident on each vertex
- implicit vs explicit
  - is the graph built only as used, or fully constructed in advance?
  - typically don't even create nodes and edges for implicit graphs – have function to compute incident edges

The Importance of the Flavor

Particular properties of the graph can affect –

- the choice of implementation for the Graph ADT
- the applicable algorithms
  - some are only meaningful for certain kinds of graphs
  - some exploit certain properties of the graph to achieve greater efficiency

Graph ADT

- not generally provided as a data structure unless you are working with a specialized data structures or graph library
  - e.g. Java Collections does not include Graph
  - graph libraries often include graph algorithms as well as the data structure

Graph ADT

What kinds of operations do we need?

- access graph structure
- modify graph structure – insert, remove

Graph ADT

- numVertices(), numEdges() – get the number of vertices/edges in the graph
- vertices(), edges() – get an iterator of the vertices/edges
- aVertex() – get a vertex of the graph
- degree(v) – get the degree of vertex v
- adjacentVertices(v) – get an iterator of the vertices adjacent to v
- incidentEdges(v) – get an iterator of the edges incident on v
- endVertices(e) – get the two end vertices of an edge
- opposite(v,e) – get the end vertex of e that isn't v
- areAdjacent(v,w) – are vertices v,w adjacent to each other? (i.e. there is an edge connecting them)

Graph ADT (for directed graphs)

- directedEdges(), undirectedEdges() – get iterator of directed/undirected edges
- destination(e), source(e) – get the destination/source of edge e
- isDirected(e) – is edge e directed?
- inDegree(v), outDegree(v) – get the in-degree/out-degree of vertex v
- inIncidentEdges(v), outIncidentEdges(v) – get iterator of the incoming/outgoing edges of v
- isAdjacentVertices(v), outAdjacentVertices(v) – get iterator of vertices adjacent to v along incoming/outgoing edges of v

Graph ADT (for modifying the structure)

- insertEdge(v,w,o) – insert undirected edge connecting vertices v, w, storing object o with the edge
- insertDirectedEdge(v,w,o) – insert directed edge from vertex v to vertex w, storing object o with the edge
- insertVertex(o) – insert a new isolated vertex, storing the object o with the vertex
- removeVertex(v) – remove vertex v and all of its incident edges
- removeEdge(e) – remove edge e (no vertices are removed, even if the removal creates an isolated vertex)
- makeUndirected(e) – make edge e undirected
- reverseDirection(e) – reverse the direction of the undirected edge e
- setDirectionFrom(e,v), setDirectionTo(e,v) – make edge e directed away from/towards vertex v

Implementing the Graph ADT

What information do we need to capture?

- structural information – edges connecting vertices
- data – vertex labels, edge/vertex weights, …
  - i.e. an object *o* associated with each Vertex and Edge

Building blocks –

- lookup is fast in arrays and if the info needed is stored directly; searching or computing is slow
  - e.g. storing a vertex's degree is faster than counting its incident edges
- storing info takes space and requires updates (slow) when the graph changes

Standard Implementations

Adjacency matrix –

- a 2D array M where M[i][j] = 1 if edge (i,j) exists and 0 otherwise

Adjacency list –

- each vertex stores a list of incident edges

Implementing Graph ADT –

- how is vertex and edge info stored? (the objects *o*)
- how do we keep track of all of the vertices? edges?
- for adjacency matrix, how do we manage going from a Vertex to the corresponding index?

Graph ADT Implementations

<u>adjacency matrix</u>

graph stores

- a list of vertices
- a list of edges
- **2D array, indexed by vertex key**

vertex stores

- the associated object
- degree of the vertex
- **distinct integer key in the range 0..n-1**

edge stores

- the associated object
- endpoint vertices

array stores

- **A[i][j] holds the edge from vertex with index i to vertex with index j (`null` if no edge)**

<u>adjacency list</u>

graph stores

- a list of vertices
- a list of edges

vertex stores

- the associated object
- degree of the vertex
- **list of incident edges**

edge stores

- the associated object
- endpoint vertices

Click to add Title

For a graph G, let n be the number of vertices, m be the number of edges, and deg(v) be the degree of vertex v.

Give the running time for the following operations on G.

	adjacency matrix	adjacency list
is edge (u,v) in G?	O(1)	O(min(deg(u),deg(v)))
get the vertices adjacent to v (i.e. those vertices u for which edge (u,v) is in G)	O(n)	O(deg(v))
insert a new vertex	O(n) best case O(n^2) worst case	O(1)

access `array[u.key][v.key]`

must scan through entire row (col) of array

best case doesn't need growing, but must initialize row and col for new vertex

search through one vertex's adjacency list – pick the one with smaller degree

search v's adjacency list

add to (unordered) list of vertices

	adjacency list	adjacency matrix
numVertices(), numEdges()		
vertices(), edges()		
aVertex()		
degree(v)		
adjacentVertices(v)		
incidentEdges(v)		
endVertices(e)		
opposite(v,e)		
areAdjacent(v,w)		
insertEdge(v,w,o)		
insertVertex(o)		
removeVertex(v)		
removeEdge(e)		
space		

Adjacency Matrix Implementation

graph stores

- a list of vertices
- a list of edges
- 2D array, indexed by vertex key

doubly-linked list allows for O(1) removal given reference to list node

vertex stores

- the associated object
- degree of the vertex
- **reference to the vertex's location in the list of vertices**
- distinct integer key in the range 0..n-1

edge stores

- the associated object
- endpoint vertices
- **reference to the edge's location in the list of edges**

array stores

- A[i][j] holds the edge from vertex with index i to vertex with index j (`null` if no edge)

Adjacency List Implementation

graph stores

- a list of vertices
- a list of edges

doubly-linked list allows for O(1) removal given reference to list node

vertex stores

- the associated object
- degree of the vertex
- **reference to the vertex's location in the list of vertices**
- list of incident edges

doubly-linked list allows for O(1) removal given reference to list node

edge stores

- the associated object
- endpoint vertices
- **reference to the edge's location in the list of edges**
- **references to the edge's location in the incidence lists for its endpoint vertices**

	adjacency list	adjacency matrix
numVertices(), numEdges()	O(1)	O(1)
vertices(), edges()	O(1) per element	O(1) per element
aVertex()	O(1)	O(1)
degree(v)	O(1)	O(1)
adjacentVertices(v)	O(1) per element	O(n) – to scan row/column of array
incidentEdges(v)	O(1) per element	O(n) – to scan row/column of array
endVertices(e)	O(1)	O(1)
opposite(v,e)	O(1)	O(1)
areAdjacent(v,w)	O(min(deg(v,w))) – search list for vertex with smaller degree	O(1)
insertEdge(v,w,o)	O(1)	O(1)
insertVertex(o)	O(1)	O(n) – to initialize row/col of array $O(n^2)$ – if array needs to grow
removeVertex(v)	O(deg(v)) – to remove each incident edge	O(1) – with clever bookkeeping (and wasted space) $O(n^2)$ – shifting in array
removeEdge(e)	O(1)	O(1)
space	O(n+m)	$O(n^2)$

Comparison

Adjacency matrix –

- **very time-efficient for `isAdjacent` – O(1)**
- very space-inefficient for sparse graphs
- time-inefficient for traversing edges incident on a vertex – O(n)
- time-inefficient for insert/remove vertex

Adjacency list –

- **space-efficient except for the most dense graphs**
- **time-efficient for traversing edges incident on a vertex – O(deg)**
- `isAdjacent` is O(deg) rather than O(1)