Graphs

The ultimate data structure
Definition of graph

- Non-linear data structure consisting of nodes & links between them (like trees in this sense)
- Unlike trees, graph nodes may be completely unordered, or may be linked in any order suited to the particular application
Undirected graphs

- Simplest form of graph
- Set of nodes (*vertices* -- one node is a *vertex*) and set of links between them (*edges*)
- Either or both sets may be empty
- Each edge connects two vertices; as the name implies, neither vertex is considered to be a point of origin or a point of destination
- An edge may also connect a vertex to itself
This graph has 5 vertices and 4 edges; a graph is defined by which vertices are connected, and which edges connect them.

This is the same graph, even though its shape is different.
Application: an undirected state graph

- Each vertex represents the state of some object or situation
- Each edge represents the possibility of moving directly from one state to another
- There exists a path from one state to another if there is an edge connecting each vertex and any vertices between them
Undirected state graph example
Bipartite Graphs

- A simple graph $G$ is called bipartite if its vertex set $V$ can be partitioned into 2 disjoint non-empty sets $V_1$ and $V_2$ such that:
  - every edge connects a vertex in $V_1$ with a vertex in $V_2$
  - no edge connects 2 vertices in $V_1$ or in $V_2$
Examples

Graph at left is not bipartite; to divide into 2 sets, one set must include 2 vertices and to be bipartite, those vertices must not be connected. But every vertex in this graph is connected to 2 others.

This graph, on the other hand, is bipartite; the two sets are $V_1 = \{v_1, v_3, v_5\}$ and $V_2 = \{v_2, v_4, v_6\}$. Note that the definition doesn’t say a vertex in one set can’t connect to more than one vertex in the other - only that each in one must connect to one in the other, and no vertex in a set can connect to a vertex in the same set.
Examples: are they bipartite?
Directed graphs

- Each edge has an orientation, or direction
- One vertex connected by the edge is the source, while the other is the target
- Directed graphs are diagrammed like undirected graphs, except the edges are represented by arrows instead of lines
Directed graph example

Each arrow represents the direction of the edge; to get from point a to point f, you must pass through c, d and b (in that order), but to get from f to a requires only going through b (by the shortest route)
Directed graph application examples

- Route diagrams for airline routes
- Course pre- and co-requisites
- Program flow charts
- Programming language syntax diagrams
Representing Graphs

- Adjacency list: specifies vertices adjacent with each vertex in a simple graph
  - can be used for directed graph as well - list terminal vertices adjacent from each vertex

<table>
<thead>
<tr>
<th>Vertex</th>
<th>Adjacent vertices</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>c,d</td>
</tr>
<tr>
<td>b</td>
<td>e</td>
</tr>
<tr>
<td>c</td>
<td>a,d</td>
</tr>
<tr>
<td>d</td>
<td>a,c,e</td>
</tr>
<tr>
<td>e</td>
<td>d,b</td>
</tr>
</tbody>
</table>
Adjacency matrix

- Square grid of true/false values that represent the edges of a graph
- If the intersection of a row and column has a true value in it, there exists an edge between the vertices represented by the two indexes
- Since this is a directed graph, the value will be true only if there is an edge from the vertex indicated by the row index to the vertex indicated by the column index
Adjacency matrix

- An adjacency matrix can be stored in a two-dimensional array of bools or ints
- Each vertex is represented by one row and one column
- For a graph with four vertices, the following declaration could be used:
  ```java
  boolean [][] aMatrix = new boolean [4][4];
  ```
The graph above could be represented by the adjacency matrix on the right.

<table>
<thead>
<tr>
<th>row</th>
<th>[0]</th>
<th>[1]</th>
<th>[2]</th>
<th>[3]</th>
</tr>
</thead>
<tbody>
<tr>
<td>[0]</td>
<td>f</td>
<td>t</td>
<td>f</td>
<td>f</td>
</tr>
<tr>
<td>[1]</td>
<td>f</td>
<td>f</td>
<td>f</td>
<td>t</td>
</tr>
<tr>
<td>[2]</td>
<td>f</td>
<td>f</td>
<td>f</td>
<td>f</td>
</tr>
<tr>
<td>[3]</td>
<td>f</td>
<td>t</td>
<td>t</td>
<td>t</td>
</tr>
</tbody>
</table>
Representing Graphs: mathematical POV

- Adjacency matrix: $n \times n$ zero-one matrix with 1 representing adjacency, 0 representing non-adjacency
- Adjacency matrix of a graph is based on chosen ordering of vertices; for a graph with $n$ vertices, there are $n!$ different adjacency matrices
- Adjacency matrix of graph with few edges is a sparse matrix - many 0s, few 1s
Simple graph adjacency matrix

With ordering a,b,c,d,e:

\[
\begin{array}{ccccc}
0 & 0 & 1 & 1 & 0 \\
0 & 0 & 0 & 0 & 1 \\
1 & 0 & 0 & 1 & 0 \\
1 & 0 & 1 & 0 & 1 \\
0 & 1 & 0 & 1 & 0 \\
\end{array}
\]
Adjacency matrices and digraphs

- For directed graphs, the adjacency matrix has a 1 in its (i,j)th position if there is an edge from \( v_i \) to \( v_j \)
- Directed multigraphs can be represented by placing the number of edges in in position \( i,j \) that exist from \( v_i \) to \( v_j \)
- Adjacency matrices for digraphs are not necessarily symmetric
Edge lists

- Linked lists are used to represent vertices
- Each vertex is a linked list; nodes in each list contain numbers indicating the target vertices for that particular source vertex
- In other words, if list A contains a link to vertex B, there is an edge from A to B
Edge lists

- For n vertices, there must be n lists; often the lists are stored as an array of list head references
- A vertex that is not a source would be represented in the array by a NULL pointer
The graph above could be represented by the array of edge lists on the right.
Incidence Matrices

- In an adjacency matrix, both rows and columns represent vertices.
- In an incidence matrix, rows represent vertices while columns represent edges.
  - A 1 in any matrix position means the edge in that column is incident with the vertex in that row.
  - Multiple edges are represented by identical columns depicting the edges.
Incidence Matrix Example

<table>
<thead>
<tr>
<th></th>
<th>e1</th>
<th>e2</th>
<th>e3</th>
<th>e4</th>
<th>e5</th>
<th>e6</th>
<th>e7</th>
<th>e8</th>
<th>e9</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>b</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>c</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>d</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>e</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
Choosing an implementation

- Adjacency matrix is easier to implement & use than any of the edge-based solutions
- Frequency of certain operations may dictate another solution
Choosing an implementation

- Both adding and removing edges and checking for the presence of a particular edge are $O(N)$ operations in the worst case for edge lists; may be as low as $O(\log N)$ for edge sets, depending on representation.
- Edge lists are very efficient for operations that execute one time for each edge with a particular vertex.
Disadvantages of adjacency matrix

- Operations on all edges for a vertex require stepping through entire row (whether or not an entry is an edge) -- this is $O(N)$
- If each vertex has very few edges, an adjacency matrix is mostly wasted space
Graph traversals

- Traversal: visiting all nodes in a data structure
- Can’t apply tree traversal methods to graphs, since nodes in a graph don’t have the same child/parent relationships
- Graph traversal methods utilize subsidiary data structures (queues and stacks) to keep track of vertices to be visited
Graph traversals

- Purpose of traversal algorithm: starting at a given vertex, process the information at that vertex, then move along an edge to process a neighbor
- When traversal is complete, all vertices that can be reached from the start vertex (but not necessarily all vertices in the graph) have been visited
Depth-first search

- After processing the initial vertex, a depth-first search moves along an edge to one of the vertex’s neighbors
- Once the second vertex is processed, the search moves along an edge to one of its neighbors
- The process continues to move forward as long as vertices can be reached in this manner
Depth-first search

- The search always proceeds forward until no further forward moves can be made.
- Once forward motion is no longer possible at the current node, the search backs up to the previous vertex and visits any previously unvisited neighbors.
- Eventually the search backs up to the original starting node, having visited all nodes that were reachable from this point.
Depth-first search example

The diagram illustrates a depth-first search example.
Breadth-first search

- Uses queue to keep track of vertices which might still have unprocessed neighbors
- Initial vertex is processed, marked, and placed in queue
- Repeats the following until queue is empty:
  - remove vertex from front of queue
  - for each unmarked neighbor of the current vertex, mark the neighbor, then place the neighbor on the queue
Graph Isomorphism

- The simple graphs $G_1=(V_1,E_1)$ and $G_2=(V_2,E_2)$ are isomorphic if there is a one-to-one and onto function $f$ from $V_1$ to $V_2$ with vertices $a$ and $b$ adjacent in $G_1$ if and only if $f(a)$ and $f(b)$ are adjacent in $G_2$ for all vertices $a$ and $b$ in $G_1$.
- Function $f$ is called an isomorphism.
Determining Isomorphism

- The number of vertices, number of edges, and degrees of the vertices are invariants under isomorphism.
- If any of these quantities differ in two simple graphs, the graphs are not isomorphic.
- If these quantities are the same, the graphs may or may not be isomorphic.
Determining Isomorphism

• To show that a function \( f \) from the vertex set of graph \( G \) to the vertex set of graph \( H \) is an isomorphism, we need to show that \( f \) preserves edges.

• Adjacency matrices can help with this: we can show that the adjacency matrix of \( G \) is the same as the adjacency matrix of \( H \) when rows and columns are arranged according to \( f \).
Example

Consider the two graphs below:

- Both have six vertices and seven edges;
- Both have four vertices of degree 2 and two vertices of degree 3;
- The subgraphs of $G$ and $H$ consisting of vertices of degree 2 and the edges connecting them are isomorphic.
• Need to define a function $f$ and then determine whether or not it is an isomorphism:
  
  – $\text{deg}(u_1)$ in $G$ is 2; $u_1$ is not adjacent to any other degree 2 vertex, so the image of $u_1$ must be either $v_4$ or $v_6$ in $H$
  
  – can arbitrarily set $f(u_1) = v_6$ (if that doesn’t work, we can try $v_4$)
Example continued

• Continuing definition of $f$:
  – Since $u_2$ is adjacent to $u_1$, the possible images of $u_2$ are $v_3$ and $v_5$
  – We set $f(u_2) = v_3$ (again, arbitrarily)

• Continuing in this fashion, using adjacency and degree of vertices as guides, we derive $f$ for all vertices in $G$, as shown on next slide
Example continued

Next, we set up adjacency matrices for $G$ and $H$, arranging the matrix for $H$ so that the images of $G$’s vertices line up with their corresponding vertices …

\[
\begin{align*}
  f(u_1) &= v_6 \\
  f(u_2) &= v_3 \\
  f(u_3) &= v_4 \\
  f(u_4) &= v_5 \\
  f(u_5) &= v_1 \\
  f(u_6) &= v_2
\end{align*}
\]
Example continued
Example concluded

- Since $AG = AH$, it follows that $f$ preserves edges, and we conclude that $f$ is an isomorphism; thus $G$ and $H$ are isomorphic.
- If $f$ were not an isomorphism, we would not have proved that $G$ and $H$ are not isomorphic, since another function might show isomorphism.
Paths

• In an undirected graph, a path of length $n$ from $u$ to $v$, where $n$ is a positive integer, is a sequence of edges $e_1, \ldots, e_n$ of the graph such that $f(e_1) = \{x_0, x_1\}$, $f(e_2) = \{x_1, x_2\}$, $\ldots$, $f(e_n) = \{x_{n-1}, x_n\}$ where $x_0 = u$ and $x_n = v$

• In a simple graph, we denote this path by its vertex sequence
Paths

- Circuit: a path that begins and ends at the same vertex (i.e., $u=v$)
- The path or circuit is said to pass through or traverse the vertices $x_1, x_2, ..., x_{n-1}$
- Simple path or circuit: path or circuit that does not contain the same edge more than once
Example 1

In the simple graph at the left, the path: \( u_1, u_2, u_4, u_5 \) is a simple path of length 3 since \( \{u_1,u_2\}, \{u_2,u_4\}, \{u_4,u_5\} \) all exist as edges.
Example 1

The path: u1, u2, u5, u4 is not a path, because no edge exists between u2 and u5.

The path: u1, u2, u6, u5, u1 is a circuit of length 4.
Paths and Isomorphism

- The existence of a simple circuit of length $k$, where $k > 2$, is a useful isomorphic invariant for simple graphs.
- If one graph has such a circuit and the other does not, the graphs are not isomorphic.
Euler circuits & paths

• An Euler circuit of a graph $G$ is a simple circuit that contains every edge of $G$

• An Euler path of graph $G$ is a simple path containing every edge of $G$
Example 1

There are several Euler circuits in this graph, including \((a, b, e, c, d, e, a)\) and \((d, c, e, b, a, e, d)\), for example. Any path that doesn’t start at \(e\) could be an Euler circuit.

This graph does not contain an Euler circuit, but does contain Euler paths. \((b, e, d, c, a, b, d, a)\) and \((a, c, d, e, b, d, a, b)\) are examples.
Example 1

This graph contains neither an Euler circuit nor an Euler path. Starting from any of the “outside” vertices, as in the graph above, you can create a circuit, but you will have to skip at least 3 edges. Starting from e, you would have to skip even more edges to make a circuit.

In seeking an Euler path starting from any “outside” vertex, you can go up or down the diagonal and all the way around, but you would have to reuse one of the “inside” edges to get to one of the others inside, and then have to reuse that one to get to the last one.
Necessary and Sufficient Conditions for Euler Circuits

• If a graph has an Euler circuit, every vertex must have an even degree
  – Starting from any vertex, you “touch” that vertex once; returning to it, you “touch” it one more time - so the initial vertex has degree 2
  – Each remaining vertex is “touched” twice each time you pass through it (by the incoming and outgoing edges), so each must have even degree
An algorithm for constructing Euler circuits

- **A**) Make an initial circuit by starting at a chosen vertex, making a path that returns to this vertex.
- **B**) Remove those edges already used, leaving a subgraph. While edges remain in the subgraph,
  - create a circuit, as in step A, starting from a vertex that was incident with an edge in the circuit from step A,
  - follow step B with the new circuit.
- The algorithm concludes when no edges are left (meaning we have an Euler circuit) or no more circuits can be constructed from the remaining edges (meaning we don’t).
Example 2: applying algorithm

Starting from vertex a, create the following circuit: a, e, b, a

Removing the edges that were used in the first step, we have the following subgraph

Starting from vertex e, we can easily construct a circuit, for example: e, c, d, e

Since no edges remain, we know the graph has an Euler circuit
Example 3: applying algorithm

Starting from a, create a circuit, for example: a, e, d, a

Removing the edges used, we get the subgraph:

Starting again from e, we can create the circuit e, c, b, e, which leaves behind the subgraphs below:

Since neither of these can form a circuit without reusing the edge, we can conclude there isn’t an Euler circuit in the graph.
Finding Euler paths

- A graph has an Euler path but not an Euler circuit if and only if it has exactly two vertices of odd degree.

The graph at left has two vertices (a and b) of odd degree; thus, it contains an Euler path (but not an Euler circuit).
Hamilton Paths & Circuits

- Hamilton path: a path that passes through every vertex in a graph exactly once
- Hamilton circuit: a circuit that passes through every vertex in a graph exactly once
Hamilton Paths & Circuits

• Although there is no simple algorithm to determine the existence of a Hamilton circuit or path, several theorems are known to give sufficient conditions for their existence, including the following:
  – If G is a connected simple graph with n vertices where $n \geq 3$, then G has a Hamilton circuit if the degree of each vertex is at least $n/2$
Example 4

Using the theorem on the previous slide, we can observe that there are 5 vertices, so $n \geq 3$; furthermore, each vertex has at least $\text{deg}(2)$: therefore, a Hamilton circuit may exist. One such circuit is: $a, c, d, e, b, a$

Note that the theorem only gives a sufficient condition for Hamilton circuits. The graph at left satisfies the theorem, but it is impossible to construct a circuit from any vertex without passing through vertex d more than once.
Determining existence of a path

• Both breadth-first and depth-first searches are suitable tools for helping determine existence of a path between two nodes
• To find a path from vertex $u$ to vertex $v$, use $u$ as the start node and run a search – if $v$ is visited, then a path exists
• A more involved problem is determining the best (shortest) path from one node to another
Finding the shortest path

• It is important to keep in mind that the picture of a graph is not the graph itself – so “shortest” is not simply a matter of physical distance

• In fact, it might be more appropriate to think in terms of the cheapest path – that is, the one that is least costly in terms of whatever resource we’re expending

• In order to find the shortest/cheapest path, we must have some way of measuring the cost (aka the weight) of each edge we must traverse
Weighted edges

• One way to add a cost dimension to a graph is by adding a weight feature to each edge
• A weight is simply a non-negative integer associated with the edge – we can think of this number as a distance or cost measure
• We determine the weight of a path by taking the sum of the weights of all the edges involved
• The shortest path between two vertices can then be defined as the path with the smallest weight
Edsgar Dijkstra and the shortest distance algorithm

• Dijkstra’s algorithm is used to determine the shortest distance from a start node to every vertex in the graph

• We start with an int array called distance with one element for every vertex – goal is to fill the array so that for every vertex v, the value of distance[v] is the weight of the shortest path from a specified start vertex to v
Steps in the algorithm

- Initialize all elements in distance array to infinity (or some approximation thereof)
- Initialize distance[start] to 0
  - shortest distance between a vertex and itself is: don’t move
- At this point, our distance array tells us that it costs us nothing to stay put, and that we can’t get from here to anywhere else
- Or, another way to state it: staying put is, thus far, the only permitted path
- Now we initialize an empty set of vertices(allowedVertices); we will fill this set with vertices that can be reached from start
- As the number of paths increases, the set grows
Allowing further paths

• The algorithm works by gradually loosening the restriction over which vertices can be visited
• As more vertices are allowed, the distance array is revised to reflect courses that pass through permitted vertices
• When the algorithm ends, we have allowed all vertices and the distance values are all correct
The big loop

- We now have an empty set of allowed vertices, and a distance array that reflects this empty state
- The next step is a loop, shown in pseudocode on the next slide
The big loop

for (size=1; size<=n; size++)

1. next = closest vertex to start that is not already in the allowedVertices set
2. add next to allowedVertices
3. revise distance array so that next appears on permitted paths

Each step is described in detail in the next several slides
Choosing the next vertex

- This step determines which of the (as yet) unallowed vertices (those which are not in the set) is closest to the start vertex.
- We choose the unallowed vertex that has the smallest value in the distance array; if several have the minimum value, pick one arbitrarily.
Add next to set & revise distance

- The mechanism for adding the next vertex to the allowedVertices set is implementation-dependent; suffice it to say we just do it.
- To revise the distance array, we must examine each array element and change it if:
  - the vertex it represents is not in the allowedVertices set.
  - the addition of the next node results in a shorter path to the unallowed node.
  - we don’t consider nodes that are already allowed because, earlier in the algorithm, we already chose them because their distance from start was less than next’s.
Pseudocode for revision of distance

for (v=0; v<distance.length; v++)
{
    if(!allowedVertices.contains(v) && an edge exists (next, v))
    {
        sum = distance[next] + weight of (next, v)
        if (sum < distance[v])
            distance[v] = sum
    }
}
The weight vs. the path

• Thus far, we have determined the weight of the shortest path from start to any other vertex.
• We need to keep track of an additional piece of information to glean the path from the weight: we need to know the predecessor of each node in the path.
Predecessor array

- We use another array to keep track of which vertex was next when distance[v] was changed.
- In other words:
  - when distance[v] = sum, then
  - predecessor[v] = next
- At the conclusion of the algorithm, distance[v] is either the weight of the shortest path from start to v, or, if no path exists, is infinity.
Printing a path

// the code below prints the path in reverse –
// that is \( v \) – start, not start – \( v \)

pathV = v;
System.out.println(pathV);
while (pathV != start) {
    pathV = predecessor[pathV];
    System.out.println(pathV);
}