Chapter 15
Graphs and Paths

You know about trees. They have a rigid structure of each node have a single node that points to it (or none, in the case of the root). Sometimes life isn’t so structured.

Examples:

- I need to fly to Tokyo. I want to find the cheapest way to fly there. How could I think about the data?
- Everybody in the class has candy bars they love and some they don’t. I have a set of candy bars in this bag. I want to distribute so everybody gets a kind they like. How do I represent the data?
- I want to travel every bike path in Logan (assuming they had any ☺) but I don’t want to ever travel on the same path twice. Can I begin at my home, visit every path, and then return?

The data representation in all of these cases is the graph!

Definitions

- **Graph** – set of vertices and set of edges.
  - More formally, a graph \( G = (V,E) \) is an ordered pair of finite sets \( V \) and \( E \).
    - The set \( V \) is the set of vertices.
    - The set \( E \) is the set of edges.
    - Sometimes the edges have weight (termed a weighted graph)
    - Sometimes the edges have direction (I can go from A to B but not B to A).
      A graph with directed edges is termed a digraph or directed graph.
      Otherwise, we call in “undirected”.
- **Successor** – the follows relationship in a directed graph.
- **Degree** – the number of edges incident to or touching a vertex.
  - **In-degree** – the number of edges coming into a vertex (digraph only).
  - **Out-degree** – the number of edges going out of a vertex (digraph only).
- **Predecessor** – the precedes relationship in a directed graph.
- Make a graph of five people around you now. The people become the nodes. Let edges be “is born in the same year as you”.
  - Is this directed or undirected?
  - Is in connected? – a connected graph is which you can get from a node to every other node following arcs.
  - What if you are all born in the same year? Then you have an edge between every pair of nodes. This is termed “complete” – as there are no missing edges.
  - For the same five people, let the edges be “A likes B.” Draw the graph.
  - Is this directed or undirected? Why?
  - What are the characteristics of the node with the largest in-degree? (most liked?)
What are the characteristics of the node with the largest out-degree? (most liking?)

**Representation of Graphs**

- How do we represent a graph?
- For the discussion below, consider the graph shown in Figure 1.

![Figure 1 A Directed Graph](image)

For the purposes of discussion, let’s assume this graph is the graph of people where the arcs represent “likes.”

**Adjacency Matrix**

- When two nodes are connected by an edge, we say the nodes are adjacent. So an adjacency matrix stores the edges.
- An *adjacency matrix* for an *n*-vertex graph \( G = (V,E) \) is an \( n \times n \) Boolean matrix \( A \). \( A[i][j] \) indicating if there is an edge between \( i \) and \( j \).
- Elements of the array can be Boolean (to indicate edge existence) or can be the weight of the edge (where 9999 or -9999 represents no edge at all)
- Common operations:
  - Determine whether there is an edge from vertex \( i \) to vertex \( j \). (In our case, does \( i \) like \( j \)?)
  - Find all vertices from a given vertex \( i \). (In our case, who does \( i \) like?)
- For the graph in Figure 1, the adjacency matrix is shown in Figure 2.
Together, let’s develop the Graph class. What are the variables? Are they public or private? What helper structures do you need?

Given the structure we developed on the board, at your seats: Write the following:
- isEdge(i,j) - returns true if there is an edge directly from i to j
- printSucc(i) – prints the names of all nodes that are successors to i
- countPred(i) – counts the number of predecessors of node i

What are the advantages and disadvantages of this storage choice? What operations can you do easily? Which ones are slower? What about space?

This adjacency matrix representation has some disadvantages, right? So we look for alternatives?

**Adjacency List**

- For a sparse graph, a better solution (as far as representation) is an *adjacency list*.
- Each node points to a linked list of successors
- Note there are two types of structures (or C++ classes) – nodes and edges.
- For the graph in Figure 1, the adjacency list is shown in Figure 3.
Together, let’s develop the Graph class. What are the variables? Are they public or private? What helper structures do you need?

We might get something like:

class EdgeNode
{
    public:
    int ToNode; // Node id of target of edge
    int FromNode; // Node id of source of edge
    int Distance; // Cost of Edge (for a weighted graph only)
    EdgeNode *Next; // Next edge in linked list for adjacency list representation

    // Constructor – note the use of default parameters.
    // Note the strange assignments (Distance(d)) – they aren’t my favorite either, but texts seem to like them
    EdgeNode( int F = -1, int T=-1, int d=0, EdgeNode *N = NULL):
    Distance(d),ToNode( T ), FromNode(F), Next( N ){};
}; // EdgeNode

class GraphNode
{
    public:
    int nodeID; // ID of node; we need something that makes it easy to find the node in the array
    string Name; // Node name
    EdgeNode *Adj; // Adjacency list
    bool visited; // true if visited already

    GraphNode( int E = -1, string name=" ", EdgeNode *A = NULL):
    Element(E), Name(name), Adj(A){ visited = false;}
}; // GraphNode

class Graph
{
    protected:
    GraphNode *G; // Array of nodes of graph – will be given space once the size is known
    int nodeCt; // Size of G

    public:
    Graph(int size ) { G = new GraphNode[size]; nodeCt = size;}; // create node array
    void PrintGraph(string label,ostream & fout);
    void BuildGraph(istream & inf);
}; // Graph
• How do I represent an undirected graph using this method?  (an edge is placed twice, once for \( A \rightarrow B \) and once for \( B \rightarrow A \))

Efficiency Considerations

• There is a difference in efficiency depending on implementation.
• With your neighbor, decide which is the best (most efficient) representation to use if you know you will need to perform lots of the following:
  o Is there an edge? – (better with matrix).
  o Find all predecessors – (better with matrix).
  o Find all successors – (better with adjacency list).
  o Which is better for space if there are few edges (adjacency list)
• At your seats:  Let’s do the same functions as before, but with the adjacency list representation.
  o \( \text{isEdge}(i,j) \) - returns true if there is an edge directly from \( i \) to \( j \)
  o \( \text{printSucc}(i) \) – prints the names of all nodes that are successors to \( i \)
  o \( \text{countPred}(i) \) – counts the number of predecessors of node \( i \)

More definitions:

• Sometimes we aren’t interested in the whole graph. Suppose the graph below represents one way streets in a town and the nodes represent hotels. Maybe I only care about hotels 0,1,3,6.  I could take those nodes and the edges between them and talk about the subgraph.

```
\begin{tikzpicture}
\node (0) at (0,0) {0};
\node (1) at (1,1) {1};
\node (2) at (-1,1) {2};
\node (3) at (0,2) {3};
\node (4) at (2,2) {4};
\node (5) at (1,0) {5};
\node (6) at (0,-1) {6};

\draw[->] (0) -- (1);
\draw[->] (0) -- (2);
\draw[->] (0) -- (3);
\draw[->] (1) -- (3);
\draw[->] (1) -- (4);
\draw[->] (2) -- (3);
\draw[->] (2) -- (5);
\draw[->] (3) -- (5);
\draw[->] (3) -- (6);
\draw[->] (4) -- (5);
\draw[->] (4) -- (6);
\end{tikzpicture}
```

• \textit{Subgraph} – a graph that consists of a subset of vertices and a subset of edges. Normally, the edges are all edges between the vertices and no others.
• By definition a graph does not contain
  o Multiple copies of the same edge.
  o \textit{Self-edges} – edges of the form \((i,i)\). This is also called a \textit{loop}.
• If we need multiple edges between nodes, it is termed a \textit{Multigraph}. In our hotel example, multiple edges between 6 and 5 could represent two different roads between the hotels.
• Sometimes we want to talk about a sequence of edges:
  o \textit{Path} – a sequence of edges (one ends where next begins)
  o \textit{Simple path} – there are no repeated vertices or edges.
  o \textit{Length of a path} – the sum of the lengths of the edges on the path.
• Sometimes we want to talk about cyclces (a path that begins and ends at the same place)
• **Simple cycle** – a cycle with no repeated vertices (except at the beginning and end). The cycle \(0 \rightarrow 3 \rightarrow 2 \rightarrow 0\) in the graph above is a simple cycle.

• **Connected component** – A component is a subgraph in which for every pair of vertices in the component, there exists some path that connects them.

• **Connected graph** – there is a path between every pair of vertices in the graph.
  
  o This implies that a graph with \(n\) vertices must have at least \(n - 1\) edges.

**Graph Traversal**

• Just like we wanted to visit all the nodes of a tree, we may want to visit all the nodes of a graph by starting at a node and following incident edges. This is called a **traversal**.

• Unlike a tree traversal, a graph traversal may not visit all the nodes. (Can you think of an example?) A graph may not even be connected.

• It will visit everything it can reach, but you may need to restart the traversal to visit all the nodes.

**Depth-first Search**

• It is like a preorder traversal on a tree.

• Consider the graph is Figure 4(a). A depth-first search (starting at node A) gives the following as the order the nodes are visited: A B E H C F G I (there are other choices as we don’t know which child to visit first. We are assuming kids are visited in alphabetic order). Notice we never hit J.

• How would this be implemented? (Use a recursive algorithm following down at deep as possible before returning from the recursion. Need to mark nodes visited.)

• You may need to restart the process if all nodes aren’t visited when you finish.

• At your seats: Write the code to visit every node and print its name in the order it is visited.
  
  o Note that with graphs, you may find yourself going in circles if you don’t mark the nodes as “visited” in some way.
Breadth-first Search

- This is like a “by level” traversal in trees.
- It is similar to a “by level” tree traversal, except:
  - Need visited flag so you don’t get into an infinite loop.
  - May need to restart the process if all nodes aren’t visited when you finish.
- For the graph shown in Figure 4(a), the breadth-first search is shown in Figure 4(b).
- The nodes are visited in the order: A B C D E F G H I  (again, it depends on which is the first child of a node).
- How would we implement the breadth-first search?
  - Visit with your neighbor and we’ll vote on your suggestions. (You use a queue, placing the starting vertex in the queue. Then take a vertex off the queue, add unvisited adjacent vertices to the queue and mark them visited. Continue until the queue is empty.)

Finding a Path

- Suppose you wanted to know if there exists a path from ‘beg’ to ‘end’. At your seats, write the recursive routine isPath which finds the answer.
- We can find a path from one vertex to another by starting at one node and doing a traversal (stopping when the desired node is reached).
  - Be careful to avoid infinite recursion by marking nodes that have already been visited.
- This is similar to a graph traversal except:
  - Don’t need to visit all nodes
  - Need to keep track of the path. One way to do this is to pass into the recursion an array of nodes on the path so far. It is important to clean up after yourself and remove an item from the array when you return from recursion.
- The pseudo-code looks something like that shown in Figure 5.

```c
char path[MAX];
int pathct=0;
// Depth First Search
DFS(&G[Source], path, pathct);
...
void DFS(GraphNode * n, char path[], int pathct)
{  if (n->id == Dest)
{  for (int i = 0; i < pathct; i++) cout << path[i];
    cout << endl;
    return;
}
    path[pathct++] = n->Element;
    n->visited = true;
    for (EdgeNode * e = n->Adj; e != NULL; e = e->next)
    {  GraphNode *to = &G[e->ToNode];
        if (!to->visited) DFS(to,path,pathct);
    }
    path[pathct--] = ' ';  // remove myself from the path
    
}
```

Figure 5  Finding a Path in a Graph
• Be careful about destructors when you don’t pass things via a pointer or reference.
• When you pass an object in (not a pointer to it) as a parameter, a copy is made. When you exit the method, the destructor is called (for the copy).
• If you have done a shallow copy going in, a deep destructor will wipe out the only data you have.
• Discuss - shallow copy and deep copy.

Connected Graphs and Components

• On an undirected graph, to make sure the graph is connected we can do a regular traversal and make sure everything has been reached.
• At your seats, write the code to see if an undirected graph is connected. Note, each edge will have been added twice as an edge from A→B also implies there is an edge B→A.
• On a directed graph what do we even mean by connectivity? There are two choices:
  1. Treat edges as undirected, and consider regular connectivity. Termed weakly connected.
  2. Only consider edges in direction of arcs. Need to make sure we can get from everywhere to everywhere. This is a bit more complicated. Termed strongly connected.
• Consider examples of connected, weakly connected, and strongly connected.

Shortest Paths

• Given a digraph (directed graph) with each edge in the graph having a nonnegative cost (or length), try to find the shortest paths (in a digraph) from a node to all other nodes.
• The basic algorithm for all points shortest path is as follows:

  Floyd Warshall Algorithm
  for k = 1 to v
    for i = 1 to v
      if path[i,k] != infinity (no path)
        for j = 1 to v
          path[i,j] = min(path[i,j], path[i,k]+path[k,j])

  For me, it is obvious that the algorithm never finds bad path lengths – as it merely replaces the current length from i to j with a path through another node (k) if it is shorter. But it isn’t as obvious that it considers every path between i and j. Can you explain why it works?
• The path[i,j] is initialized to the weight of the arc from i to j. path[i][i]=0.
  path[i][j]=∞ if there is no edge between i and j. However, by the time the algorithm finishes, path contains the weight of the path between i and j. In fact, at intermediate points it contains the minimal weight between the points for paths possibly going through vertices 1..k.
• For example, consider the digraph in Figure 6. Calculate the shortest path.
• Path Array (\& marks infinity)
  0 \ 8 \ & \ 9 \ 4 \\
  & \ 0 \ 1 \ & \ & \\
  & \ 2 \ 0 \ 2 \ & \\
  & \ & \ 3 \ 0 \ 7 \\
  & \ & \ 1 \ & \ 0 \\

• Path Array (allow paths going through 0)
  0 \ 8 \ & \ 9 \ 4 \\
  & \ 0 \ 1 \ & \ & \\
  & \ 2 \ 0 \ 2 \ & \\
  & \ & \ 3 \ 0 \ 7 \\
  & \ & \ 1 \ & \ 0 \\

• Path Array (allowing any paths going through node 1)
  0 \ 8 \ 9 \ 9 \ 4 \\
  & \ 0 \ 1 \ & \ & \\
  & \ 2 \ 0 \ 2 \ & \\
  & \ & \ 3 \ 0 \ 7 \\
  & \ & \ 1 \ & \ 0 \\

• Path Array (allowing any paths going through node 2)
  0 \ 8 \ 9 \ 9 \ 4 \\
  & \ 0 \ 1 \ 3 \ & \\
  & \ 2 \ 0 \ 2 \ & \\
  & \ 5 \ 3 \ 0 \ 7 \\
  & \ 3 \ 1 \ 3 \ 0 \\

• Path Array (allowing any paths going through node 3)
  0 \ 8 \ 9 \ 9 \ 4 \\
  & \ 0 \ 1 \ 3 \ 10 \\
  & \ 2 \ 0 \ 2 \ 9 \\
  & \ 5 \ 3 \ 0 \ 7 \\
  & \ 3 \ 1 \ 3 \ 0 \\

• Path Array (allowing any paths going through node 4)
  0 \ 7 \ 5 \ 7 \ 4 \\
  & \ 0 \ 1 \ 3 \ 10 \\
  & \ 2 \ 0 \ 2 \ 9 \\
  & \ 5 \ 3 \ 0 \ 7 \\
  & \ 3 \ 1 \ 3 \ 0
Single Source Shortest Path

- We could use a breadth first search (using a queue to store the nodes) to determine the shortest path if every edge had the same distance.

```cpp
void PrintShortestPath( int source )
Queue q;
q.enqueue( Pair(source,0 ) );
while (!q.isEmpty())
{  Pair p = dequeue();
    if ( !V[p->node].isVisited )
    {
        V[p->node].isVisited = true;
        cout << "The shortest distance from " << source << " to " << p.node << " is " << p.dist;
        for ( Edge * e = V[source].adj; e!= NULL; e = e->next )
            q.enqueue( Pair(e->toNode, p.dist+1));
    } // if
} // PrintShortestPath
```

- The first time we pull a node off the queue, its best distance is known.
- However, if all edges are not the same length, it won’t work; we must look at edge weights.
- Since we have weighted edges, the only differences are:
  - Add the edgewt from node to its successor to the current distance.
  - Pull nodes off based on shortest distance – using a priority queue
- A priority queue is an abstract data type in computer programming, supporting the following two operations
  - Add an element to the queue with an associated priority.
  - Remove the element from the queue that has the highest priority, and return it.
- This is termed Dijkstra’s algorithm.
- Here is the pseudocode

```
function Dijkstra(Graph, source):
    for each vertex v in Graph:   // Initializations
        dist[v] := infinity       // Unknown distance function
        previous[v] := undefined  // stores the best predecessor
    dist[source] := 0            // Distance from s to s
    Q := copy(Graph)             // Set of all unvisited vertices
    while Q is not empty:         // The main loop
        u := extract_min(Q)      // Remove best vertex from priority queue;
        // returns source on first
        // iteration
        for each neighbor v of u:
            alt = dist[u] + length(u, v)
            if alt < dist[v]      // Relax (u,v)
                dist[v] := alt
                previous[v] := u
```

• Go thru an example on the board – not tracing the code, but just showing the functioning.

Spanning Trees

• A spanning tree is a set of all of the nodes of a graph and some of the edges such that you have a tree (every node is reached).
• For example, suppose we have a graph where the nodes are people in this class and the edges are “knows the phone number of”. Now suppose class is cancelled. We want to tell one person and have everybody be notified. However, we don’t want a person called multiple times. If we create a spanning tree where the arcs selected represent “will call to tell about cancellation” – we can create a way of informing everyone. You may have heard this termed a “calling tree” in everyday English.
• How do we create a spanning tree?
• BFST (graph): breadth first spanning tree: Do a breadth first traversal, but don’t consider previously visited nodes. Visit and mark nodes as they are enqueued.
• DFST (graph): depth first spanning tree: Do a depth first traversal, but don’t consider previously visited nodes. Visit and mark nodes as they are visited via recursion. Sometimes we number the nodes sequentially as we visit them. Such numbers are called DFS (depth first spanning) tree numbers.
• DFS numbers are useful because you are attempting to give an order to nodes in the graph so you can reason about them (e.g., finding strongly connected components)

Minimum-Cost Spanning Trees

• Suppose I wanted to build a power system. Everybody needed to be connected to the power supply, but no one really cared how far away they were. Now, suppose that houses are the nodes and the edges are weighted with the cost of running electrical wires between the locations. Can you see why we need a minimum cost spanning tree?
• Given a weighted graph, construct the minimum-cost spanning tree.
• Use the following criterion to choose the next edge to add:
  o From the remaining edges, select a least-cost edge whose addition to the set of selected edges forms a tree.
  o Very similar to Dijkstra’s shortest path algorithm. The only difference is in how you update the distance to a node.
  o You grow the tree starting at the source.
  o You place all the edges emanating from the source on the PriorityQueue.
  o You pull off the smallest edge in PQ which connects something that isn’t already in the tree.
• Use the following algorithm (called Prim’s algorithm):
  Let \( T \) be the set of selected edges. Initialize \( T = \emptyset \)
  Let \( TV \) be the set of vertices already in the tree. Set \( TV = \{1\} \)
\[ n = \text{source} \]
place all edges from \( n \) on PQ (a priority queue)
while (\(!\text{PQ isempty}\))
{ 
\( \text{Edge } (u,v) = \text{PQ dequeue}() \);
if (\( v \) is already in tree) break;
Add edge (\( u,v \)) to \( T \)
Mark \( v \) as being part the tree.
place all edges from \( v \) (to nodes not in tree) on PQ
}
if (\( |T| == n -1 \))
\( T \) is a minimum-cost spanning tree
else
There is no minimum-cost spanning tree
• See example in Figure 6

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{prim_algorithm.png}
\caption{Prim's Algorithm}
\end{figure}

**Topological Ordering**

• I am having company for dinner. There are several things that must happen. Some are ordered with respect to each other. Let the nodes be “activities” and the arcs be “must be done before”.
  o Come up with several activities (set table, send out invitations, buy food, cook steak, peel potatoes, cook potatoes, mash potatoes, eat dinner) and draw the directed acyclic graph (also termed a dag) representing the activities.
  o A topological order is an order (not necessarily unique) in which I could do the tasks and not violate the “must be done before” relationship.
• The text calls it topological sorting. While this is a common name, I find students get confused and think of a mergesort or bubble type of sort, so I don’t use that name. I just call it a topological order.
• Assume we have a bunch of tasks that need to be completed and there is a partial ordering between some of the tasks.
  o We could represent this as a task directed graph.
  o Nodes are tasks.
  o Edge \((i,j)\) denotes that \( i \) must be completed before \( j \) starts.
• Try drawing a directed graph for making chocolates. The nodes represent steps and the directed arcs represent “do before.” See Figure 8.

![Graph](image)

**Figure 8 Make Chocolates**

• *Dag* – directed acyclic graph. Arcs represent a partial order among vertices.
• *Flowgraph* – dag with a unique entry node.
• *Total ordering* – when there is an ordering between every two vertices.
• *Topological ordering* – an ordering such that for each edge \((i, j)\), \(i\) precedes \(j\) in the list.
• *Greedy criteria* – for the unnumbered nodes, assign a number to a node for which all predecessors have been numbered.
• The algorithm to find a topological order is a bit tricky.
• Algorithm
  o Initially count how many predecessors each node has. The predecessor count is best thought of as the number of “unscheduled” or “unordered” predecessors it has.
  o For each node with predecessor count of zero, put it in a container (stack, queue, anything will do).
  o Remove a node with predecessor count of zero from the container, order it sequentially, then print it.
  o For each successor, decrement its predecessor count.
  o If a successor now has a predecessor count of zero, add it to the container.
• What is the Complexity – assuming an adjacency list representation for the graph.
1. Find predecessor count. $O(e)$, where $e$ is number of edges.

2. List node – for each successor, update predecessor count. $O(e + n)$, since if have lots of nodes with no predecessors, $n$ could be more than $e$. 