Graph basics

Susan Eileen Fox

August 19, 2008

1 Graphs: definition, terminology, and implementation

A graph is a way of representing data that has complex, non-linear, and non-hierarchical structure to it. A graph consists of a set of vertices, connected by edges. The edges may be directed or undirected, weighted or unweighted. Vertices are identified either by some kind of data associated with each vertex, or at least by number. Note that the word "node" is also used to refer to a vertex.

Figure 1 shows a graph with 5 vertices (labeled A through E) and 7 edges. It is an "undirected" graph, because there are no arrow points on the edges. This means that we can move either way along an edge, for instance, from A to B and also from B to A.

Graphs may be represented using either an *adjacency matrix* or an *adjacency list*. In the adjacency matrix representation, a graph is a two-dimensional array of values. If there are n vertices, then the matrix is $n \times n$. The cell at (i, j) in the matrix contains a one (or a weight value) if there is an edge in the graph between vertex i and vertex j. This representation makes it very efficient to determine the existence of an edge between to vertices, but less efficient to collect up all the neighbors of a vertex. Figure 2 (a) shows the adjacency matrix representation for the graph in Figure 1.

The adjacency list representation uses a one-dimensional array or list n long. Each cell in the array contains a list: the values in the list are the vertices that are directly connected to the vertex associated with that cell. It takes constant time to gather the neighbors of a vertex, but determining whether two vertices are connected requires searching the relevant linked lists. Figure 2 (b) contains the adjacency list representation for the graph given above.

Graphs can have weights on the edges, that indicate cost or distance. We'll talk about dealing with weighted graphs later on.

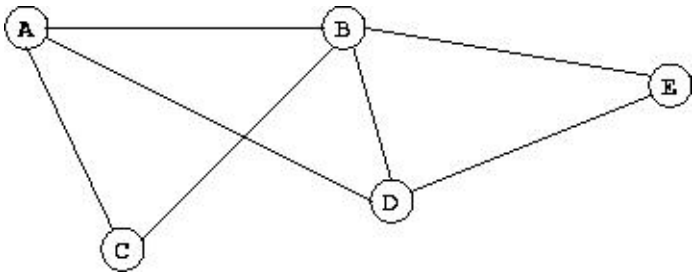

Figure 1: A simple graph

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

A	[B, C, D]
B	[A, C, D, E]
C	[A, B]
D	[A, B, E]
E	[B, D]

Figure 2: Adjacency matrix (a) and adjacency list (b) representations of a graph.

1.1 Class activity: Graphs of friends

This activity might be augmented using Facebook or MySpace or another online community website, or simply using acquaintances at their school.

Have two or three class volunteers begin to draw a graph on the board of "friends." Each person should put his or her name and add several friends by putting her name and several friends. Then examine the friends of the friends. Use the result to talk about graph properties: nodes versus edges, branching factors, connectedness, and paths. Ask the class if the graph is connected. Ask them to find the shortest path between two people in the graph.

Ask new volunteers to draw up the adjacency list and adjacency matrix representations for this graph.

Then they are ready for graph search algorithms

2 Graph search on unweighted graphs

2.1 Breadth-First Search

The word "search" here refers to a process of walking a graph, following the edges of the graph. It can be used for a variety of uses, including iterating over all the vertices of the graph, and finding a path from some vertex to others. Searching a list or matrix just means iterating through the values. But here, the connections between vertices are wild and woolly, so searching effectively becomes a bigger problem.

The Breadth-First Search algorithm (abbreviated BFS) searches a graph by considering vertices that are close to the starting point before those further away. You can picture the search as a set of ripples starting at the starting vertex: and gradually spreading out away from it. For some uses the algorithm continues until it has touched every vertex that is reachable from the starting vertex. As an example, if we started breadth-first searching the graph from Figure 1, starting at vertex E, then we would first visit B and D (in no particular order), and then A and C (in no particular order). This is because B and D are direct neighbors of E, one edge away, while A and C are each two edges away from E.

BFS has some nice features. If the edges of the graph are weighted equally (all have the same cost), then BFS will find the shortest path from some starting point to any other node. This is because it always examines first the paths of length 1, then those of length 2, and so forth. If a path exists, it will find it.

In order to function efficiently, BFS needs to keep track of which vertices it has already looked at, the *Visited* set. This ensures that it never revisits a vertex it has already reached. It also needs to keep track of the set of vertices that are unvisited neighbors of vertices in the *Visited* set: these will be the next vertices to be visited. This is sometimes called the "fringe" set, because it represents the fringe of the ripple, the boundary vertices between those already touched by the ripple, and those not yet touched. It turns out that

BFS can't just keep the fringe vertices as a set, it needs to keep them in a particular order (the order by which their visited neighbors were visited). To do this, BFS uses a data structure called a Queue.

We are going to assume the following operations on Queues:

Method	**Description**
QUEUE()	Creates a new, empty queue object
Q.ISEMPTY()	Checks if the queue is empty or not
Q.FIRSTELEMENT()	Returns the first element without removing it
Q.REMOVEFIRST()	Removes the first element from the queue and returns it
Q.INSERT(v)	Inserts the new value v at the end of the queue

Figure 3 contains *pseudocode* for the BFS algorithm. This one takes a starting vertex s, and a graph G as its inputs, and finds the shortest path between s and all other vertices. To keep track of visited nodes, we will keep a *visited* list or dictionary that contains True for visited nodes and false for unvisited ones. We also assume a dictionary or list *prevNode* that keeps, for each visited vertex, which vertex it was visited from. Using *prevNode* we can reconstruct the best path for any particular vertex after this is done.

```
BFS (G, s)
1.   for v in vertices of G do
2.       set visited[v] = False
3.   set visited[s] = True
4.   set prevNode[s] = None
5.   set Q = QUEUE ()
6.   Q. INSERT (s) // insert s into the queue
7.   while not Q. ISEMPTY () do
8.       set u = Q. REMOVEFIRST ()
9.       for each v in the neighbors of u do
10.          if not visited[v] then
11.              set visited[v] = True
12.              set prevNode[v] = u
13.              Q. INSERT (v)
```

Figure 3: Pseudocode for Breadth-First Search

2.1.1 Class activity: Walking through BFS

Take the graph that was drawn on the board earlier, or a new graph, and label each vertex in the graph with a letter or number. Divide the class into groups and set each group the same task:

> Work through the steps BFS would take in searching from a given starting point to the goal vertex. Show each vertex as it is marked as visited, and write out the contents of the queue at each step. How many vertices end up in the queue before it finds a path to the goal? Write out the final values in *prevNode*, and show how we could reconstruct the shortest path from it.

Afterwards, ask each group to show what they did and explain their work.

2.2 Depth-First Search

Depth-First Search (DFS) also may be used to find a path from some vertex to the others, or to iterate through the vertices of the graph according to the edges in the graph. While BFS might be seen as a cautious algorithm, slow and steady; DFS is more reckless. The DFS algorithm starts at a given vertex, and then chooses one of the neighbors to visit. At that neighbor, it chooses one of its neighbors to visit, and goes on. Thus, it will visit all the vertices that are reachable from the first neighbor, before it ever returns to consider the second neighbor of its first node.

DFS is a *backtracking* algorithm, because it plunges as far away from the start as it can go, but then backtracks to try alternative paths. DFS can also be thought of as an algorithm for finding a path through a maze: go until you hit a dead end, then backtrack to the most recent intersection with an unvisited path, and take that path.

Much like the BFS algorithm above, this keeps track of which vertices have been visited, to avoid revisiting them. For each visited vertex, it records the vertex it came from so we can reconstruct the DFS path to the vertex. Figure 4 contains pseudocode for DFS; note that the recursive helper has access to the variables of DFS, to make things easier.

```
DFS (G, s)
1.  for v in vertices of G do
2.      set visited[v] = False
3.  set visited[s] = True
4.  set prevNode[s] = None
5.  DoDFS (s)

DoDFS (u)
1.  for each v in the neighbors of u do
2.      if not visited[v] then
3.          set visited[u] = True
4.          set prevNode[v] = u
5.          DoDFS (v)
```

Figure 4: Pseudocode for Depth-First Search

It is hidden by the recursion above, but the DFS algorithm is actually using a stack data structure in much the same way that the BFS algorithm used a queue. The stack, here, is the program stack. But the stack maintains, through the recursive calls and the unfinished work for each call, the fringe vertices that are the boundary between visited and unseen parts of the graph.

We will assume the following operations on stacks:

Method	**Description**
STACK()	Creates a new, empty stack object
S.ISEMPTY()	Checks if the stack is empty or not
S.TOP()	Returns the first element without removing it
S.POP()	Removes the first element from the stack and returns it
S.PUSH(v)	Inserts the new value v at the top of the stack

Figure 5 contains an alternative form of Depth-First Search that uses a stack explicitly. It needs to mark

a node as visited when it is removed from the stack, and needs to record its predecessor at the same time. Therefore, both the node and its predecessor are pushed on the stack, as a tuple.

```
DFS2 (G, s)
1.    for v in vertices of G do
2.        set visited[v] = False
3.    set S = STACK ()
4.    S. PUSH ((s, None))
5.    while not S. ISEMPTY () do
6.        set u, pred = S. POP ()
7.        if not visited[u] then
8.            set visited[u] = True
9.            set prevNode[u] = pred
10.           for each v in the neighbors of u do
11.               if not visited[v] then
12.               S. PUSH (v)
```

Figure 5: Pseudocode for Depth-First Search, using an explicit stack and iteration

2.2.1 Class activity: Walking through DFS

Repeat the same activity as before, for the DFS algorithm.

3 Searching weighted graphs

The previous examples assumed that each edge was equal. However, weighted graphs are more realistic for many applications: where the cost of using an edge may be variable. For example, a graph that represents points in the world might use weighted edges to reflect the distance between two points. Figure 6 shows the earlier graph with weights on its edges.

The adjacency matrix representation of a graph is easy to convert to a weighted form. Instead of having each cell contain zero or one, each cell contains zero if there is no edge, and the weight value if there is an edge. For the adjacency list representation, both the neighbor node and the weight of the edge must

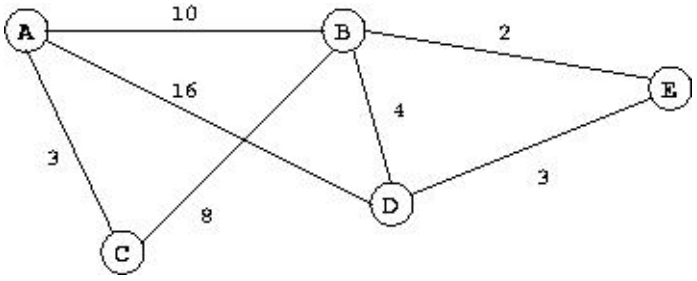

Figure 6: A simple graph with weighted edges

	A	B	C	D	E
A	0	10	3	16	0
B	10	0	8	4	2
C	3	8	0	0	0
D	16	4	0	0	3
E	0	2	0	3	0

A	[(B, 10), (C, 3), (D, 16)]
B	[(A, 10), (C, 8), (D, 4), (E, 2)]
C	[(A, 3), (B, 8)]
D	[(A, 16), (B, 4), (E, 3)]
E	[(B, 2), (D, 3)]

Figure 7: Adjacency matrix (a) and adjacency list (b) representations of a weighted graph.

be combined together as a tuple. Figure 7 shows the adjacency matrix and adjacency list for the weighted graph.

When we search for a path in a weighted graph, typically we want the path with the lowest total cost across all its edges. This is harder than BFS or DFS. We're going to look at two related algorithms for solving this problem. Best-First Search is a variations on Breadth-First Search. Dijkstra's Algorithm computes the shortest path from a starting point to all other points in the graph, in the process finding the shortest path to a particular goal. Best-First Search is a good lead-in to A* and related *heuristic* search algorithms that are more common in AI where large graph spaces make other techniques infeasible. Dijkstra's algorithm is what systems like Google Maps or Mapquest probably use to generate driving directions, because common values can be precomputed and stored, so that the actual graph to be searched may be limited.

Both these algorithms use a data structure to keep track of "fringe" elements of the graph. Instead of a simple queue or stack, however, they use a *priority queue*. A priority queue contains data, but associated with each piece of data is a priority value. The queue always organizes the data so that the value with highest priority comes first. For our application, we're going to use cost or distance as the value, so "highest priority" means "lowest cost". You can imagine the priority queue is just a list of data kept in order by priority, but more often priority queues are implemented as *heaps*. I won't talk about heaps here, so use an algorithms or data structures reference if you need it.

3.1 Best-First Search

Best-First Search is a close cousin of Breadth-First Search, but works on weighted graphs. Like BFS, Best-First wants to search the shortest paths first. Thus, as it visits each node in the graph it keeps track of the total cost of the edges leading up to that node. Because a single edge can have a greater cost than a bunch of edges with smaller weights, it cannot decide it is finished with a node until it *removes* it from the queue. Thus nodes aren't marked at the time that they are added to the queue, but when they are removed, and the same node may appear multiple times in the queue, associated with different paths. Because of this, Best-First keeps track of the path leading to each node as a part of the data in the queue. Figure 8 contains pseudocode for Best-First Search.

3.1.1 Class activity: Walking through Best-First

Put a weighted graph on the board. Working with the class, determine some example where the shortest path BFS would find is not the lowest-cost path. Then, repeat the same activity as before for Best-First search, showing how it finds the lowest-cost path.

```
BESTFIRSTSEARCH (G, s, t)
1.   for v in vertices of G do
2.       set visited[v] = False
3.   set Q = PRIORITYQUEUE ()
4.   Q. INSERT (0, (s, [s])) // insert s into the queue
5.   while not Q. ISEMPTY () do
6.       set p, (u, path) = Q. REMOVEFIRST ()
7.       if u = t then
8.           return path
9.       set visited[u] = True
10.      for each v in the neighbors of u do
11.          if not visited[v] then
12.              set newp = p + cost of edge from u to v
13.              set newPath = path + [v]
14.              Q. INSERT (newp, (v, newPath))
15.  return NO PATH
```

Figure 8: Pseudocode for Best-First Search

3.2 Dijkstra's Algorithm

Dijkstra's algorithm solves a bigger problem than the path from one node to another. It finds the shortest path from a starting node to *every other node in the graph*. In the process, it generates a compact representation of those shortest paths, which may be stored and used later to generate any of the paths we want.

Dijkstra's algorithm uses a priority queue, though in a slightly different way than the earlier algorithms. All the vertices of the graph are put into the priority queue at the start. The priorities of vertices in the queue reflect the current approximation to the length of the shortest path from start to that vertex. The priorities get updated as the algorithm goes along. The algorithm keeps the priority values in a table, for simplicity, and uses the same idea as BFS and DFS of keeping track of predecessors, to be able to reconstruct the actual path at the end. Figure 9 contains the pseudocode for this algorithm.

3.2.1 Class activity: Working through Dijkstra's

As a class walk through a small example with Dijkstra's algorithm. Have small groups of students reconstruct the shortest paths from the resulting data. Discuss the question: will Dijkstra's always give the same result as Best-First, for a particular starting node and goal node?

```
DIJKSTRA (G, s, t)
1.    set Q = PRIORITYQUEUE ()
2.    for v in vertices of G do
3.        set visited[v] = False
4.        set dist[v] = ∞
5.        set prevNode[v] = None
6.        Q. INSERT (dist[v], v)
7.    set dist[s] = 0
8.    Q. UPDATE (dist[s], s) // update s's priority
9.    while not Q. ISEMPTY () do
10.       set u = Q. REMOVEFIRST ()
11.       set visited[u] = True
12.       for each v in the neighbors of u do
13.           if not visited[v] then
14.               set newDist = dist[u] + cost of edge from u to v
15.               if newDist < dist[v] then
16.                   set dist[v] = newDist
17.                   set prevNode[v] = u
18.                   Q. UPDATE (newDist, v)
19.   return RECONSTRUCTPATH (s, t, prevNode)
```

Figure 9: Pseudocode for Dijkstra's algorithm