Introductory Graph Theory Part I

Senior Computer Team
Jeff Chen

October 12, 2006

1 What are Graphs and Why Do I Care?

Welcome to the wonderful world of graph theory. In computer science, a graph is a set of objects (vertices or nodes) connected by links (edges). Graphs are used everywhere. In contests like USACO and Topcoder, a large amount of problems will require you to know and utilize graph theory. Below are some major definitions that every programmer should know:

2 Definitions

Weight: values attached to edges or vertices in order to model specific networks.

Path: a sequence of connected edges and vertices. The **length** of a path is the number of edges traversed.

Circuit: a path that ends at the vertex it begins.

Loop: an edge that connects a vertex to itself (a circuit of length 1).

Degree: the number of edges that end at a vertex.

Undirected/Directed Graphs: a directed graph, or digraph, is a graph in which the edges are directed to one direction. paths cannot travel in the opposite direction.

Sparse/Dense Graphs: Density of a graph is defined as $|E|/|V|^2$ where $|E|$=number of edges and $|V|$=number of vertices. A sparse graph has low density – the number of edges is far from the maximum.

Connected/Complete Graphs: connected=a path connecting every pair of vertices, complete=each vertex is directly connected to each of the others.

Network: a digraph with weighted edges.

Adjacency List/Matrix: a data structure which stores the neighbors of each vertex. An adjacency matrix is a VxV matrix, in which if x is adjacent to y, the element $M_{x,y} = 1$, else $M_{x,y} = 0$.

3 Searching

It is arguable that graphs are most commonly employed for searching. For all graph searching algorithms, there exists a general outline for graph traversal. The idea is to begin at a vertex and branch out to other vertices until you reach your destination. Most searches can be done with a list and a loop. You start at the root vertex, perform an operation, add in more vertices to the list, and repeat until you have reached the destination or the list is empty.

The difference in searching techniques is the order in which neighboring vertices are placed in the queue. the order is designated by $f(n) = g(n) + h(n)$, where $g(n)$ is the current cost of the edges traversed and $h(n)$ is an underestimated heuristic (an estimated cost) from the current vertex to the destination. Below is a table of different searches based on the utilization of $g(n)$ and $h(n)$.

$g \backslash h$	0	nonzero
0	uninformed search	greedy best-first search
nonzero	uniform-cost search	A* search

Note that $g(n) = 0$ denotes that there are no costs (weights) associated with the graph and that $h(n) = 0$ signifies that the algorithm does not utilize a heuristic. Next we will go through each type of search. As greedy best-first search is highly limited in effective usage, we will skip over it (you can search it up on your own time, using Wikipedia).

3.1 Uninformed Search

When you have no cost and no heuristic for finding the next vertex, you have a *blind or uninformed search*. The most common and simplest blind search is called the Depth-First Search, or DFS.

3.1.1 Depth-First Search (DFS)

DFS is the most basic recursive tree/graph traversal algorithm (note that you can also implement DFS with a stack). It is used for unweighted graphs. Given a tree or graph, We start at the root and begin to deepen into the tree or graph until it hits a wall and needs to backtrack. Recursive DFS doesn't need a list to maintain edges, so its memory consumption is relatively low. The time complexity for DFS is $O(|E|)$. DFS is very easy to code, but its recursive nature can lead to branching problems and stack overflow. Below is a bare-bones pseudocode of DFS:

```
dfs(vertex v)
{
   visited[v]=1;
   performOperation();
   for(unvisited vertices i adjacent to v)
      dfs(i);
}
```

[Traditional] Knights Tour: Given an NxN chessboard and a knight at a certain position, find a path in which the knight traverses each square exactly once and returns to its original location.

If the constraints on N are low, we may simply search recursively. We keep an array of visited squares, and DFS from the starting position:

```
ktour(int row, int col)
{
   if(row and col aren't in bounds)
      return;
   mark square as visited;
   record step number;
   for(all 8 adjacent knight moves)
      ktour(respective move);
      mark square as unvisited;
      erase step number;
}
```

note the last two steps: if we become stuck and backtrack, we simply undo what was performed. Another way is to pass individual visited and marker arrays for eeach recursive sequence.

3.1.2 Flood Fill

Flood fill is a specialized DFS designed for covering bounded areas – like floods. It can simulate viral infection, and other "spreading" models. The difference between flood-fill and generic DFS is that the purpose of flood fill is to split the graph into connected components. When you use the "paint bucket" in MS Paint, you *are* performing a flood fill.

3.2 Uniform Cost Searches

Uniform cost searches is a weighted graph search which uses a cost storage to determine the order of vertices to visit. The actual Uniform Cost Search (UCS) simply sorts its neighbors in ascending order of cost and stores them in the traversal list. Although you won't use UCS itself very often, you will be using some of its special cases, described below.

3.2.1 Breadth-First Search (BFS)

BFS is simply a UCS that gives all its edges the same weight. It is essentially UCS dumbed down for unweighted graphs. It uses a FIFO queue to store the list of vertices to visit. Since it uses a queue, its memory consumption is high, but it interates level-wise, which is better than DFS in many problems. Here is the pseudocode:

```
void bfs(vertex v)
{
   queue q;
   q.enqueue(v);
   visited[v]=1;
   while(q is not empty)
   {
      v=q.dequeue();
      process;
      for(all unvisited vertices i adjacent to v)
         q.enqueue(i);
 }
}
```

3.2.2 Iterative Deepening Depth-First Search (DFSID, ITDFS or IDDFS)

ITDFS is a compromise between BFS and DFS. It searches level-wise like BFS, but is recursive like DFS. This translates to a search with low memory consumption that still maintains DFS time complexity. The idea is to DFS over and over again, each time with a level restriction that it cannot cross. We DFS down to one level, and see if we get the solution. If not, we DFS down to two levels, and so on. The consequence is that we are repeating the initial levels many, many times, but since there are relatively few nodes in the initial levels, we don't lose much time. Use DFSID when a BFS requires too much memory.

```
//repeat this operation for each level
void dfsid(vertex v, level l, bound)
{
if(level > bound)
     return -1;
  visited[v]=1;
  performOperation();
for(unvisited vertices i adjacent to v)
 dfsid(i,level+1,bound);
}
```

3.2.3 Dijkstra's Algorithm

Dijkstra's is an effective algorithm for finding the shortest path between two vertices in a directed graph with nonnegative edge weights. Using a simple implementation, the runtime is $O(V^2)$. It is like a UCS except that it keeps track of the current best distance from the source to all the other vertices at any point in the traversal. This is called *memoization*, we will go more in depth later.

Dijkstra's algorithm works using edge relaxation. The notion of "relaxation" comes from an analogy between shortest path and a tension spring. First we overestimate a shortest path. Over time, as shorter paths are found, we "relax" the spring, or lower our cost.

Given an array of weights $w[i][j]$ =weight of edge from vertex i to vertex j, a boolean array visited[], and a dist[] array being the distance from the source vertex to every other vertex, the pseudocode for dijkstra's is as follows:

```
void dijkstra()
{
   set all dist to infinity;
   set all visited to 0;
   dist[source]=0;//source vertex to source vertex=0
 while(not all visited)
 {
  find unvisited vertex i with shortest path to source;
  visited[i]=1;
  for(each neighbor vertex i of j)
  if(dist[i]+w[i][j]<dist[j])
   dist[j]=dist[i]+w[i][j];
 }
}
```

Now we have found the shortest path from the source vertex to every other vertex.

3.3 A* Search

The A* Search is a heuristic best-first search algorithm that can be used for traversing weighted graphs. Used in the appropriate environment, it can be faster than Dijkstra's. It is complete, so it will always find the optimal solution if there is one. The disadvantage of A* is that its heuristic usually requires extra knowledge. For instance, to utilize A* for shortest-route finding, you need to know the straight line distance between each node and the destination. The actual algorithm is similar to a BFS, except we use a priority queue to determine the order of nodes to visit. Priority is determined by P=the total cost of the edges followed so far + the straight-line distance between the current vertex and the destination vertex. The lower the P, the higher the priority.

```
//dest vertex d
astar(vertex v)
{
    priority queue pq;
    pq.enqueue(v, 0);
    visited[v]=1;
    while(pq is not empty)
    {
        vertex g, cost c=pq.dequeue();
        for all nonvisited neighbors i of g
            visited[i]=1;
```

```
            enqueue in order of c + dist(i,d);
    }
}
```