Graph Theory

Basic functions for most graph theory problems. Both types of searches are provided where the path is stored backwards in the *parent* array. An Adjacency matrix is used to store the graph.

```
public int nodeCount;
public int edge[][];
public int flow[][];
public int parent[];

// Prepares a graph for a given number of nodes.
public void createGraph(int nodeCount) {
	this.nodeCount = nodeCount;
	this.edge = new int[nodeCount][nodeCount];
	this.parent = new int[nodeCount];
}
// Adds a weight to a given edge.
public void addEdge(int start, int end, int weight) {
	edge[start][end] = weight;
}

//Breadth-First-Search of Graph using a Queue.
public boolean breadthFirstSearch(int start, int target) {
	boolean[] visited = new boolean[nodeCount];
	Queue<Integer> queue = new ArrayDeque<Integer>(nodeCount + 2);
	queue.offer(start);			// Start searching on the first node
	parent[start] = -1;			// Clear its parent.
	while (!queue.isEmpty()) {		// While nodes exist in the Q
		int u = queue.poll();	/ Remove and traverse the next node
		if (u == target)			// If we've reached the target, return success
			return true;
		for (int v = 0; v < nodeCount; v++) {		// Search all nodes...
			// Must be an unvisited node, must have flow left...
			if (!visited[v] && (edge[u][v] > flow[u][v])) {
				queue.offer(v);		// Add it as a possibility for traversal
				visited[v] = true;	// Mark it as visited
				parent[v] = u;		// Save the parent of this node for backtracking
			}
		}
	}
	return false; // The target node has not been reached.
}

//Depth-First-Search of Graph using a Queue.
public boolean depthFirstSearch(int start, int target) {
	boolean[] visited = new boolean[nodeCount];
	Stack<Integer> stack = new Stack<Integer>();
	stack.push(start);			// Start searching on the first node
	parent[start] = -1;			// Clear its parent.
	while (!stack.isEmpty()) {		// While nodes exist in the Q
		int u = stack.pop();	// Remove and traverse the next node
		if (u == target)			// If we've reached the target, return success
			return true;
		for (int v = 0; v < nodeCount; v++) {		// Search all nodes...
			// Must be an unvisited node, must have flow left...
			if (!visited[v] && (edge[u][v] > flow[u][v])) {
				stack.push(v);			// Add it as a possibility for traversal
				visited[v] = true;	// Mark it as visited
				parent[v] = u;		// Save the parent of this node for backtracking
			}
		}
	}
	return false; // The target node has not been reached.
}
```

Network Flow

Given a graph with weighted edges (where the weight is the maximum flow capacity of the edge) and unweighted nodes this will determine the maximum amount of flow that can pass from any source node to any sink node.

```
//Ford-Fulkerson Max-Flow Min-Cut Theorem
public int maxFlow(int source, int sink) {
	int maxFlow = 0;
	// Initialize the flow matrix (how much has flowed on each edge).
	flow = new int[nodeCount][nodeCount];
	// While there's a path with flow between the source and the sink...
	while (breadthFirstSearch(source,  sink)) {
		// Backtrack and find minimum cut.
		int minCut = Integer.MAX_VALUE;
		for (int u = sink; parent[u] >= 0; u = parent[u]) {
			minCut = Math.min(minCut, edge[ parent[u] ][u] - flow[ parent[u] ][u]);
		}
		// Backtrack and adjust total flow of all edges.
		for (int u = sink; parent[u] >= 0; u = parent[u]) {
			flow[ parent[u] ][u] += minCut;
		}
		// Add how much flowed in this path.
		maxFlow += minCut;
	}
	return maxFlow;
}
```

Shortest Path

Given a graph with weighted edges (where the weight is the distance between the adjacent nodes) and unweighted nodes this will determine the shortest path between a source node and a target node.

```
// Dijkstra's Shortest Path Algorithm
public void shortestPath(int source, int target) {
	// The distance cost for each node.
	int[] dist = new int[nodeCount];
	// The array to keep track of visited nodes.
	boolean[] visited = new boolean[nodeCount];
	// Clear all parents and distances
	for (int i = 0; i < nodeCount; i++) {
		dist[i] = Integer.MAX_VALUE;
		parent[i] = -1;
	}
	// No cost for the source node.
	dist[source] = 0;
	// Test all nodes until target is found.
	for (int i = 0; i < nodeCount; i++) {
		// Find the index of the shortest unvisited edge
		int minIndex = -1;
		for (int j = 0; j < nodeCount; j++) {
			if (!visited[j] && (minIndex == -1 || (dist[j] < dist[minIndex])))
				minIndex = j;
		}
		// If the minimum is infinity then exit
		if (dist[minIndex] == Integer.MAX_VALUE)
			break;
		// This node has been visited.
		visited[minIndex] = true;
		// Traverse all neighboring nodes connected to minIndex.
		for (int j = 0; j < nodeCount; j++) {
			// The edge must exist...
			if (edge[minIndex][j] != 0) {
				// Calculate the travel distance if we go to this edge.
				int newDistance = dist[minIndex] + edge[minIndex][j];
				// If the new distance is less then track the movement.
				if (newDistance < dist[j]) {
					dist[j] = newDistance;
					parent[j] = minIndex;
				}
```

Path Traversal

Backtracks through a path computed in the DFS or BFS methods.

```
public void backtrackPath(int target) {
        // Traverse nodes
        for (int i = target; i >= 0; i = parent[i]) {
                //.. do whatever with node[i]
        }
        // Traverse edges
        for (int i = target; parent[i] >= 0; i = parent[i]) {
                //.. do whatever with edge[ parent[i] ][i]
        }
}
```

Traversal (preOrder, postOrder, inOrder)

```
        final int NULLCHILD = -1;
        int[] tree = {0, 1, 2, 3, NULLCHILD, 5, 6};

        void dfPreOrder(int n)
        {
                if(tree[n] == NULLCHILD) return;

                System.out.printf("%d\n", tree[n]); //do us
                dfPreOrder(2*n+1); //do left child
                dfPreOrder(2*n+2); //do right child
        }

        void dfInOrder(int n)
        {
                if(tree[n] == NULLCHILD) return;

                dfInOrder(2*n+1); //do left child
                System.out.printf("%d\n", tree[n]); //do us
                dfInOrder(2*n+2); //do right child
        }

        void dfPostOrder(int n)
        {
                if(tree[n] == NULLCHILD) return;

                dfPostOrder(2*n+1); //do left child
                dfPostOrder(2*n+2); //do right child
                System.out.printf("%d\n", tree[n]); //do us
        }
```

Transitive Closure

```
boolean edgeE[][]; //edgeE[i][j] = there exists an edge from i to j
boolean exists[][]; //exists[i][j] = there exists a path from i to j

void trasitiveClosure()
{
	int i, j, k;

	//n = # of nodes
	//copy over edges
	for(i =0; i < n; i++)
	{
		for(j=0; j< n; j++)
		{
			exists[i][j] = edgeE[i][j];
		}
	}
	//every node can reach itself
	for(i =0; i < n; i++)
	{
		edgeE[i][i] = true;
	}
	//Perform Floyd Warshall Aglorithm
	for(k = 0; k < n; k++) //length of current paths
	{
		for(i = 0; i < n; i++)
		{
			for(j = 0; j < n; j++)
			{
				exists[i][j] = exists[i][j] || (exists[i][k] && exists[k][j]);
			}
		}
	}
}
```

Shortest Path Between I and J

```
int edges[][]; //edge[i][j] = length btwn direct edge btwn i and j
        int dist[][];  //dist[i][j] = length of shortest path btwn i and j
        int path[][]; //path[i][j] = on shrotest path from i to j, path[i][j] is the last node
before j
        int n;

        void solveFloydWarshall()
        {
                int i, j,k;
                //copy edge weight over and set each path pointing to itself
                for(i = 0; i < n; i++)
                {
                        for(j =0; j< n; j++)
                        {
                                dist[i][j] = edges[i][j];
                                path[i][j] = i;
                        }
                }
                //all diagonal distances is 0
                for(i = 0; i < n; i++)
                {
                        dist[i][i] = 0;
                }
                //perform the algorithm
                for(k = 0; k < n; k++)//length of the current paths
                {
                        for(i = 0; i < n; i++) //start at node i
                        {
                                for(j =0; j < n; j++) //point to node j
                                {
                                        int cost = dist[i][k] + dist[k][j];
                                        if( cost < dist[i][j])
                                        {
                                                dist[i][j] = cost; // reduce i -> j to smaller i ->
                                                                   //k -> j
                                                path[i][j] = path[k][j]; //update path for i -> j
                                        }
                                }
                        }
                }
        }
```

Minimum Spanning Tree

MST can be solved several ways, here are two (pseudo code).

Prim's Algorithm

Start at arbitrary node (0)
visitied[0] = true
visitedCount = 1;
while (visitedCount < nodeCount) {
    Traverse all visited nodes and look at all adjacent edges and determine the least weighted edge.
    If the edge has an unvisited node on the end of it then mark that node as visited and increment visitedCount
}

Kruskal's Algorithm

Add all edges to a priority queue ordered from least weighted edge to most weighted edge.
visitedCount = 0;
while (visitedCount < nodeCount) {
    Remove the edge (with the least weight) from the queue
    If one of the nodes on the edge has not been visited {
        Mark any unvisited node as visited
        For each recently visited node increment visitedCount
    }
}

To approximate a zero of a function F(x) to arbitrary precision
1) Choose a starting value x0
2) compute or approximate the derivative
f(x0) = dF/dt (at x0)
3) construct the tangent line at F(x):
y = F(x0) + f(x0)*(x - x0)
4) Set y=0 to find the zero-crossing of this line:
x = x0 - (F(x0)/f(x0))
5) goto (2) with this x as your new x0
Terminate when (x == x0) within the required tolerance.
This will tend to find zeros near x0; it is not guaranteed to find a particular zero (some are repellers, not attractors of this algorithm), but it should find at least one zero for any analytic function / function with a continuous second derivative over the area of interest (i.e. any function you'd see in competition).