Graph Search

1. You've been working hard for this class recently, so I'm not making you do a graph homework. But if I did, this is what would be on it (i.e., and on the final):
    a. Explain tradeoffs between adjacency matrix and adjacency list representations
    b. Change depth-first graph search into breadth-first graph search
    c. Implement isAcyclic() on adjacency list implementation
    d. Convert adjacency list into adjacency matrix implementation

2. Review MatrixGraph.getEdge
3. **MatrixGraph.isConnected**
    a. Recursive
    b. Depth-first using stack
    c. Breadth-first using queue
4. shortestPath
    a. *A* path vs. *best* path
    b. Best-first search
    c. Dijkstra shortest**-completed** first
        i. Guaranteed optimal result
        ii. May take a while…
    d. A*: shortest**-remaining** first
        i. Needs a **heuristic** to estimate how far the goal is from the current position
        ii. Often faster than Dijkstra
        iii. No optimality guarantee
        iv. (this is how people plan routes)
    e. Floyd's all pairs algorithm – see book

5. Administrative
    a. No class Friday (again!)
    b. Darwin lab due at midnight tonight
        i. Final contest creatures due tomorrow before 2:30pm to Kyle
    c. If you're doing the optional lab, you can come to either or both lab sections
        i. Otherwise, just show up at 2:20 pm for Darwin contest this week
        ii. Optional Graph lab goes out Wednesday
    d. Exam + grades

```
/** Returns true if v0 -> v1
    Breadth first implementation using a queue.*/
private boolean isConnectedBF(int start, int finalGoal) {
    Queue<Integer> todo = new LinkedList<Integer>();
    HashSet<Integer> visited = new HashSet<Integer>();
    todo.offer(start);

    while (! todo.isEmpty()) {
        int youAreHere = todo.remove();
        visited.add(youAreHere);

        if (youAreHere == finalGoal) {
            return true;
        }

        for (Edge edge : adj.getRow(youAreHere)) {
            if ((edge != null) &&
                ! visited.contains(edge.finish)) {
                todo.offer(edge.finish);
            }
        }
    }

    return false;
}
```

```
/** Finds the shortest path from v0 to v1 by always advancing the
        currently shortest path. (i.e., being greedy: best-first).  */
    public ArrayList<Edge> shortestPathDijkstra(Vertex v0, Vertex v1, LabelComparator<EdgeLabel> c) {
        return shortestPathDijkstra(v0.index, v1.index, c);
    }

    public ArrayList<Edge> shortestPathDijkstra(int start, int finalGoal, LabelComparator<EdgeLabel> c) {
        PriorityQueue<ArrayList<Edge>> queue = new PriorityQueue<ArrayList<Edge>>(10, new PathComparator(c));

        HashSet<Integer> visited = new HashSet<Integer>();

        if (start == finalGoal) {
            // We're at the goal
            return new ArrayList<Edge>();
        }

        visited.add(start);

        // Add initial paths out of v0
        for (Edge edge : adj.getRow(start)) {
            if (edge != null) {
                ArrayList<Edge> path = new ArrayList<Edge>();
                path.add(edge);
                queue.offer(path);
            }
        }

        while (true) {
            // Get the shortest path
            ArrayList<Edge> shortest = queue.remove();

            // Extend this path in every possible way
            int youAreHere = shortest.get(shortest.size() - 1).finish;
            visited.add(youAreHere);

            if (youAreHere == finalGoal) {
                // This path just reached the goal
                return shortest;
            }

            for (Edge edge : adj.getRow(youAreHere)) {
                if ((edge != null) &&
                    ! visited.contains(edge.finish)) {
                    // The following line gives an unchecked cast warning; that is
                    // a flaw in the Object.clone method and not this code.
                    ArrayList<Edge> path = (ArrayList<Edge>)shortest.clone();
                    path.add(edge);
                    queue.offer(path);
                }
            }
        }
    }

    /** Used by shortestPath code to compare two paths and see which is longer.*/
    public static interface LabelComparator<EdgeLabel> extends Comparator<EdgeLabel> {
        /** if a is null, return b, otherwise combine them */
        public EdgeLabel combine(EdgeLabel a, EdgeLabel b);
        public int compare(EdgeLabel a, EdgeLabel b);
    }
```