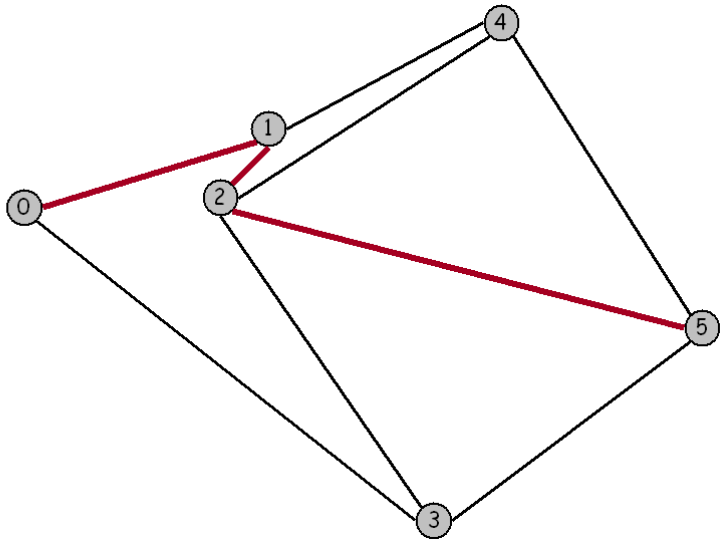
Shortest Path

In this lab, your goal is to find the shortest path through a graph from one node to another using Dijkstra's algorithm. Don't even think about proceeding without reviewing the notes first!

You will read in a graph from a text file. The input files will have the format shown below on the left:

```
6 9
0 1000 2400
1 2800 3000
2 2400 2500
3 4000    0
4 4500 3800
5 6000 1500

0 1
0 3
1 2
1 4
2 4
2 3
2 5
3 5
4 5
0 5
```



The first two integers `V` and `E` represent the number of vertices (points) and edges (connections) in the graph. The next `V` lines represent the points' "IDs" (index numbers), followed by its X- and Y-locations on the 2D plane. The following `E` lines represent the list of edges, the connections from one point to another (e.g. point 0 connects to point 1 and point 3). The final line is the desired route (i.e. find the shortest path from 0 to 5). The file "input6.txt" represents the graph shown above (the lines in red show the shortest path).

One of the trickiest parts of graph problems is choosing the right data structures for representing the graph. There are many options: adjacency matrix, adjacency list, edge list + vertex list, adjacency map (adjacency list with maps), etc. In addition, there are many decisions to be made for each option: do you wrap the underlying data into a class, e.g. Vertex and Edge, or do you use simple Strings and Integers? Frustratingly for students, there is no universal right or wrong way.

You are encouraged, given your knowledge of graphs, to represent the graph in whatever way you see fit. Creating your own abstraction of this problem will definitely deepen your understanding of graph algorithms, even if you are not able to finish the program on your own.

If you're suitably terrified, read on; a suggested implementation is provided.

1. Define a class **Vertex.java**, an abstraction of a node on the 2D plane (a node in the graph). Vertex should implement the Comparable<T> interface (more info to follow), and should have the following:

    a. Three integers `x`, `y`, and `ID`, that represent this node's X- and Y-location on the 2D plane, and the "vertex number" (ID) as shown in the graph image previously, respectively.

i. If you wanted, you *could* wrap the `x` and `y` variables into a Point object, such that a Vertex has a Point. Follow your heart; I don't think it's a necessary for this project.

b. In addition, a Vertex should have these instance variables:

i. `List<Integer> edges`: the list of edges (connections) for this node.

1. We will take advantage of the fact that nodes in this graph have simple "IDs" (integers, starting from 0). Were this not the case, you would probably need to define an Edge class that would store its ID, source and destination, and (possibly) weight.

ii. `boolean visited`: indicates whether or not this node has been explored, specifically used for the shortest path algorithm (to prevent infinite cycles).

iii. `double distance`: the distance to this node *from some source node*. This value will be updated as the program finds the shortest path through the map; used specifically for the shortest path algorithm.

1. Normally this value would be part of an Edge class, such that an edge has a distance from a source to a destination Vertex. However, this value is included in Vertex to simplify some of the logic; hopefully you'll see why later.

c. A suitable constructor and overridden `toString` method.

i. All nodes should start off unvisited, with all distances set to "infinity" (use `Double.POSITIVE_INFINITY`) as per Dijkstra's algorithm.

d. A method that will return the Euclidean distance from `this` point to another point.

e. An overridden `compareTo` method; nodes with smaller `distance`s should come before larger.

2. Define a class **Graph.java**, which will represent the graph shown previously. Graph uses an adjacency list representation (the graphs in this lab are too sparse to represent with an adjacency matrix), a list of vertices where each vertex maintains a list of its connections. Graph should have the following:

a. Two integers `V` and `E`, the number of vertices and edges in this graph.

b. An array of Vertex objects (the adjacency list - each Vertex stores its edges).

c. A constructor that accepts a Scanner that represents a valid input stream. Use the Scanner to read space-separated values from the underlying file and initialize the instance variables.

i. Recall that points are listed as follows: `vertexNumber xLocation yLocation`.

d. A method `double distance(int from, int to)` that will return the distance between two vertices (two array indexes).

e. It would be wise to test that this class is working before proceeding. Add a `main` method that uses the "input6.txt" file, shown previously. It would probably help to make a reasonable `toString` method to see the state of a graph object.

3. Now that you have created the necessary data types for representing the graph, it's time to find the shortest path. Define a new class **Dijkstra.java** that will use Dijkstra's (single source shortest path) algorithm to find the shortest path between two point through the supplied graph. This class should have the following:

   a. A constructor that accepts and stores a Graph variable.

   b. A `private` method `void dijkstra(int source, int destination)` that uses Dijkstra's algorithm to calculate the shortest path and store the results in the Vertex class' `distance` field. Recall that Dijkstra's algorithm is as follows:

      - All vertices' distances start at "infinity" except for source vertex; source distance = 0.
        - Use the constant `Double.POSITIVE_INFINITY` to represent infinity (this should already be handled by the Vertex class).
      - Push the source (start) vertex in a min-priority queue; the comparison in the min-priority queue will be according to vertices' distances from the source node.
      - Pop the vertex with the minimum distance from the priority queue (at first the popped vertex will be the source).
      - Update ("relax") the distances of the neighbors to the popped vertex (its edges); in case of "current vertex distance + edge weight < next vertex distance" (a shortest link is found), add the vertex with the updated distance to the priority queue (if not visited).
        - There are many ways you could track already explored nodes (a boolean array, a Set of "settled" nodes, etc.); for simplicity this functionality was bundled into Vertex.
      - Apply the same algorithm again until the priority queue is empty.

Optional information: *At this point, you may be able to understand why `distance` (the cost of travelling from some source to a destination) was bundled into Vertex. Were it not, you would need to create a new Comparable node class specifically for the priority queue, in the form of (distance, vertex), or use an array where all distances start at "infinity". Adding a distance field to Vertex allowed it to be put into the priority queue directly. There are many other ways this could have been done.*

Cloning this project and using different data structures to store the distances (e.g. a `double[]` where the value at a particular index stores a distance to that node from the source, or a `Map<Vertex, Double>`) may really help to cement graph algorithms in your mind. This stuff isn't easy, it takes practice!

Your output should provide a step-by-step description of how Dijkstra's algorithm finds the distance (rounded to the nearest tenth) of the shortest path from 0 to 5 for the "input6.txt" graph (shown previously):

```
process node 0 (dist 0.0)
    lower 3 to 3841.9 //the "relaxing" step
    lower 1 to 1897.4
process node 1 (dist 1897.4) //greedily choose next (closest) node
    lower 4 to 3776.2
    lower 2 to 2537.7
process node 2 (dist 2537.7)
    lower 5 to 6274.0
process node 4 (dist 3776.2)
process node 3 (dist 3841.9)
process node 5 (dist 6274.0) //queue is now empty, we're done!
```

4. Did it work? If so, congratulations! If not, you should probably figure out why.

5. Finally, notice that at the moment your program only finds the *distance* of the shortest path through the map. Add the ability to track and output the shortest path itself (e.g. 0 -> 1 -> 2 -> 5, for the "input6.txt" file shown previously). Choose the methodology / data structures that make the most sense to you.

```
Shortest path from 0 to 5:
0 -> 1 -> 2 -> 5
```

(Optional) SMPT servers

There is a nice introductory shortest path competition style problem in the lab folder if you'd like a little more practice. Practice makes perfect! All the vertices, edges, and weights are simple integers. Woah!

(Advanced) The A* algorithm

The A* (pronounced "A star") algorithm is a slight modification on the Dijkstra algorithm that uses additional ***heuristics*** to produce a shortest path faster than the greedy algorithm used with Dijkstra.

Think of A* as basically an informed variation of Dijkstra.

A* is a "best first search" because it greedily chooses which vertex to explore next, according to the value of a function `f(v) = g(v) + h(v)`, where `g` is the cost so far (to a particular node), and `h` is the heuristic.
To put it another way, the A* algorithm, without heuristics, is simply the Dijkstra algorithm. A* has two "cost" functions:

- `g(v)`: same as Dijkstra. The real cost (edge weight / distance) to reach a node `v`.
- `h(v)`: *estimated* cost from node `v` to the goal node. This is the heuristic function; it should never overestimate the cost (the real cost to reach the goal node from node `v` should be greater than or equal to `h(v)`, otherwise the heuristic overpowers the actual cost).

    - Use the web to find some common heuristic methodologies, based on the type of graph you're working with.

Dijkstra has just one cost function, which is the real cost value from source to each node: `f(v) = g(v)`. It finds the shortest path from source to every other node by considering only the real cost.

For MUCH more information on pathfinding, see here: theory.stanford.edu/.../AStarComparison.html

Use the "usa.txt" to test the A* algorithm against Dijkstra's algorithm.