Graph Traversal Algorithms

Classes that implement the algorithms

Posn, CartPt, Place

The classes `Posn`, `CartPt`, nad `Place` are those you have designed in the earlier assignments.

The `Place` class has a `name` given as a `String`, a location `loc` that is represented as a `CartPt`, and a list of neighbors. You can use your earlier code, on you can change the list of neighbors to be represented by an `ArrayList`.

The `Place` class has methods that compute the distance from`this` place to the given one, a method that tells us the direction of travel from this place to the given one (N, E, S, W, or also NE, NW, SE, SW), and should have a method that draws on the `Canvas` `this` place and another one that draws the line from `this` place to the given one, possibly in the given color.

The Capital class

In one of the earliler assignments we also designed a class `Capital` that extended the class `Place`. It allowed us to add the information about the city, the state, the zip code, and the earth coordinates (the latitude and the longitude), while retaining the ability to draw the city on the `Canvas` and provide the routing directions.

Of course, it also has a list of neighbors — already guaranteed by the `Place` class we extend.

The Path class or the Route class

The path from the origin to the destination is just a list of links. One may want to design a class to represent this information, to add a method that computes the total length of the path, checks for circularity, etc.

The `Route` class in the previous assignment extended the `World` and animated the routing in response to the space bar key events.

The StateMap class

This class represents the graph we are traversing. It contains a list of all cities (`Capitals`). The cities already record the information about their

neighbors, and so no additional information is required.

In general, the graph traversal algorithms we are designing require that the graph be given as a list (collection) of nodes (in our case the capitals), with the list of links to the neighbors given for each node.

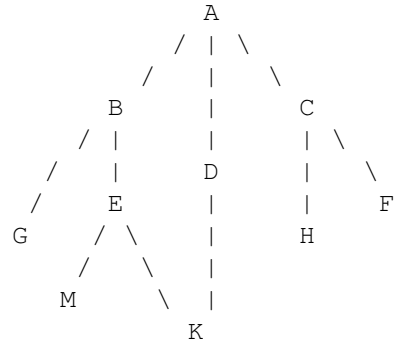
The FromTo class

As we examine the possible next steps along the route, we need to keep track of the possible legs of the route. To do so, we design a class `FromTo` that contains the place we are coming from, or the `source`, the place we are going to or the `target`, and the distance for the routing. The distance maybe the actual distance between the two places, or, in the *shortest path* algorithm, the distance to the `target` when coming from the `source`.

The backtrack information

During the running of the algorithm we will select certain links between the places as the ones that should be in the final route.

For example in the graph G shown below, the *breadth-first search* algorithm will select the links $(A - to - B : dist - 2)$, $(A - to - D : dist - 4)$, $(A - to - C : dist - 2)$, $(B - to - G : dist - 3)$. They are saved as the *backtrack* data. Additionally, when we start, we begin with the special link, $(A - to - A : dist - 0)$, to indicate the origin for the route we are seeking. Meanwhile, the links $(B - to - E : dist - 2)$, $(D - to - K : dist - 4)$, $(C - to - H : dist - 3)$, and $(C - to - F : dist - 2)$ are waiting-to-be considered, in the `ToDo` queue. We do not get-to-look at the link $(E - to - M : dist - 2)$, and $(E - to - K : dist - 3)$, until we start looking at the neighbors of the node E.



As we run the algorithm we need to check whether the target of a link we are considering is already a target of a link in the *backtrack* data. That means we need a method `contains` that determines whether the *backtrack* data contains a `FromTo` object with the given target.

The class should also include a method `pathTo` that for the given target produces a route from the origin to the given target. The route is best defined at a list of `FromTo` objects — that way all information we need to show the path, to animate it, to produce the GPS directions, is readily available.

So, in the above example, if we were looking for a path from A to K and the *backtrack* data included all the information shown earlier, as well as the link $(E - to - K : dist - 3)$, we would find the route by finding the link with the target K, then find the link with the target E, then the link with the target B, and, finally the link with the target A. Here we recognize the origin and stop the search.

You may want to use the `HashMap` with the name of the `target` as its *key* to record the *backtrack* data.

The ToDo interface and classes that implement it

The `ToDo` interface defines the two methods we need to keep track of the *to-do* list of tasks — the collection of the links to consider next. Depending on the algorithm we choose, the interface may be implemented as a stack, as a queue, or as a priority queue.

The interface may be defined as:

```
/**
 * To represent a collection of the links
 * from the visited places
 * to their neighbors, not yet visited.
 */
interface ToDo{

  /**
   * Add a new neighbor to the ToDo checklist
   * unless it already appears
   * in the given <code>Path</code>
   * @param city the <code>City</code> to add
   * @param path the given <code>Path</code>
   */
  public void add(FromTo ft, ArrayList<FromTo> backtrack);

  /**
   * Remove the next item from the ToDo checklist
   *
   * @return the FromTo information for the next connection
   * to record
   */
  public FromTo remove();
}
```

Of course, the type of data for the `backtrack` argument depends on how the *backtrack* data is represented in your program.

The first method allows us to add a new link to the *to-do* tasks, the second method selects the next task to work on: the next link to process.

The GraphTraversal class

This class implements the graph traversal algorithm. Its main reason for existence is the method `findRouteFromToInMapAlgoType`. Well, this is not the name you should use, but the method has available the map of the USA - or the graph in which we are looking for a route, it is given the desired origin and destination, and the algorithm choice: *breadth-first search*, *depth-first search*, or the *shortest path*. This method then invokes the actual search with the initial `ToDo` list (a queue, a stack, or a priority queue), a new backtrack list with just the origin link in it and returns the route the search produces.

This is the part that is described in detail in the original homework assignment.

We did not discuss how the final version of the route should be represented, but you can figure that out. You can use what you have designed in the earlier assignment, so you can then animate the routing.

User interactions and displaying the results

Download the file **GraphAlgoView.java** and include it in your project. The file defines its own `main` method. Create a configuration that uses this `main` and run it. You will see a simple GUI that allows you to select the *origin* and *destination* for your route, as well as one of the three types of algorithms. Observe what happens.

When you select one of the algorithms, the program reads the current values in the GUI, prints the information, and invokes one of three methods - depending on the choosen algorithm. These methods are *stubs* - they do not do anything useful, but act as placeholders for your code.

You need to add to this class a field that represents your `GraphTraversals` class, properly initialized to the map of the USA. You can then modify the methods `bfs`, `dfs`, and `sp` to include the invocation of the appropriate algorithm in your `GraphTraversal` class.

Add to the class that represents the route a method `toString()` that shows the directions of travel as text, preferably one line per one leg of the

journey. Now, show the computed route by adding

```
System.out.println(myRoute.toString());
```

after the route has been computed.

You do not need to display the graph of the map or animate the routing.

Testing

Remember, designing tests for every little part of your program will make your life much easier. You will know what it should do (especially if you write the purpose statements carefully), you will know it does what it is supposed to do, and you will know how it can be used in further design. If that is not enough, you will also get points for the good design (and loose them if it is lousy, and the tests are not sufficient).