Implementing the shortest path algorithm

To the right is the invariant of the shortest path algorithm, which calculates shortest paths and their distances from node `v` to all nodes. Below is the algorithm itself. We discuss implementing this abstract algorithm in a professional way.

Invariant

P1. For node `s` in settled set `S`, at least one shortest `v-s` path contains only settled nodes and `d[s]` is its distance.

P2. For `f` in *frontier set* `F`, at least one `v-f` path contains only *settled* nodes, except for `f`, and `d[f]` is the minimum distance from `v` to `f` over all such paths from `v` to `f`.

P3. Edges leaving `S` end in frontier set `F`.

P4. For `w` in `S` or `F`, `bp[w]` is `w`'s backpointer on the shortest known path from `v` to `w`.

Theorem. For `f` in *frontier* set `F` with minimum `d` value (over nodes in `F`), `d[f]` is the shortest-path distance from `v` to `f`.

```
F= {v}; d[v]= 0; S= {};
// invariant: P1, P2, and P3
while (F != {}) {
   f= a node in F with minimum d value;
   Remove f from F and add it to S;
   for each w with (f, w) an edge {
      if (w not in S or F) {
         d[w]= d[f] + wgt(f, w); bp[w]= f;
         add w to F;
      }
      else if (d[f] + wgt(f, w) < d[w]) {
            d[w]= d[f] + wgt(f, w); bp[w]= f;
         }
   }
}
```

In this rather abstract version of the algorithm, the nodes are numbered, and these node numbers are used to access the shortest known distance `d[w]` and backpointer `bp[w]` for a node `w`.

Consider the gps system in a car. It finds the shortest path from a start node to *one* desired end node, not to all nodes. We want to implement the above algorithm to find not all paths but only one, like the gps system.

Also, it's not realistic to use node numbers and arrays `d` […] and `bp` […] in such a system. Think of googlemaps, finding a shortest route from Gates Hall in Cornell, Ithaca, NY to Fisherman's Wharf in San Francisco. There are at least 19,500 cities in the U.S. Suppose each has 100 intersections, each of which would be a node of the graph of the U.S. We doubt whether googlemaps starts a search for a shortest path by initializing two arrays of size 1,950,000!

In a more realistic, object-oriented version of the algorithm, say in Java, each node would be an object of some class `Node`. There would also be a class Edge. These nodes —i.e. pointers to the nodes— would be used, not node numbers. Using Java, the following data structures could be used:

Data Structures

1. Frontier set `F` is implemented as a min-heap. The values are nodes, and the priority of a node in `F` is its shortest-path distance from the start node, as described in invariant P2 above. The min-heap must allow priorities of elements in the min-heap to change.

2. We don't want to use arrays `d` and `bp`. Instead, place each pair (`d[w]`, `bp[w]`) in an object of class `Info`, shown to the right. `Info` will have a constructor and method `toString`. The fields are public so they can be referenced quickly.

```
/** An instance contains the distance of
  * a shortest path to a node and the back
  * pointer for that node */
public class Info {
   public double dist;      // the distance
   public Node bckptr;      // the backpointer
   …
```

3. Look at the invariant, above. For what nodes must a distance and backpointer be maintained? Answer: For every node in the settled and frontier sets. Therefore, we introduce a variable `data`:

   ```
   HashMap<Node, Info> data
   ```

   Thus, `data` contains an entry for each settled node and each frontier node. Give `data` such a node and it returns an object of class `Info`.

4. Now one can see that set `S` itself is unnecessary. *Do not implement set `S`.* Where is it used in the above algorithm? In two places. There is a statement "add it to `S`". Don't implement it. There is a condition "`w` not in `S` or `F`". Use `HashMap data` for that, since it contains an entry for every node in `S` or `F`. Isn't that neat?

Writing the algorithm as a method

Here is the specification and header of the method:

```
/** Return the shortest path from start to end ---or the empty list if a path does not exist.
  * Note: The empty list is NOT "null"; it is a list with 0 elements. */
public static List<Node> shortestPath(Node start, Node end) {
```

Class `Info` can be a static inner class of the class in which this method appears. That way, the user doesn't have to know about it. Data structures `F` and `data` will be local variables of this method.

Important points about the implementation of the method

The shortest-path algorithm will be used often. It should be as efficient as possible. To that end, every effort must be made to streamline it, to avoid evaluating the same expression twice, for example. Below, we list some points about this. If they are followed carefully, a short, simple, easy-to-understand method emerges.

But don't wait until the algorithm is "done" and debugged to check the points below. As you are implementing, *always* keep them in mind and change the method immediately if necessary. Doing this sometimes uncovers more simplifications and efficiencies.

1. The method *must* return with the result as soon as the shortest path to node end is known. The theorem given after the invariant at the beginning of this document tells you when that is! Check that condition —and use a return statement; don't use a break statement.
2. Keep the abstract algorithm and its implementation as close as possible. For example, use the names `F`, `f`, and `w` in the implementation since the variables appear in both the abstract algorithm and its implementation.
3. Put in good definitions of `F` and the `HashMap`!
4. Give mnemonic names to local variables. For example, if you save the `data` object for node `f`, save it in a variable `fData` or something like that. Calling that variable `Mary` or `John` is not good.
5. Place local variable declarations as close to their first use as possible. Do *not* place them all at the beginning of the method.
6. Placement of assignments: Do not place an assignment to a variable in a loop if it has to be calculated only once and can be done before the loop.
7. Don't have a method call with the same arguments in two different places. Instead, call the method once and save its value in a local variable.
8. Don't have the same expression, e.g. the equivalent of `d[w]= d[f] + wgt(f, w)`, in two different places. Instead, evaluate it once and save its value in a local variable.
9. Don't create an instance of `data` or call `data.put(…)` when not needed. For example, they are not needed when processing a node `w` that is in `S` or `F`. Instead, change the fields of the existing `data` object. Remember, a variable of type `Info` contains a pointer to an object, not the object itself. This is a good thing here.