COMP 360, Spring 2013, Assignment 3

In this assignment, you will explore using mutation and closures to create a new data type in an object-oriented style, as well as implement a classic algorithm using the functional programming paradigm.

Part 1

Using the same OOP-using-closures style from the stack data type discussed in class earlier (see the code from 2/14 or 2/21), create a graph data type. We use "graph" here to refer to the mathematical notion of a graph: a structure consisting of a set of vertices and a set of edges, where each edge connects two vertices. Furthermore, your graph should be undirected (meaning an edge from vertex v to vertex w is equivalent to an edge from w to v), and weighted, meaning each edge has an integer weight associated with it. Weights in graphs commonly represent distances or costs. Vertices in the graph will be represented by symbols (that is, the Racket data type "symbol").

Just like the stack example, your graph should use closures to simulate "private" fields in your graph object. You may choose whatever graph representation you prefer (e.g., adjacency lists, adjacency matrix, or something else), and pick whatever variables you want to store the information in your graph. For instance, you could use one variable to store a list of the vertices in your graph, and another variable to store a list of the edges in the graph, where each edge would be a list of three things: the starting vertex, the ending vertex, and the weight of the edge (that was my solution).

Your graph's variables will need to be mutated whenever vertices or edges are added. Therefore, you will need to use `set!` and/or mutable pairs with `set-mcar!` and `set-mcdr!`. Because mutable pairs are a distinct data type from regular pairs, see the notes later in this document for more information about dealing with mutable pairs.

Your data structure should support the following operations: *(All of the examples below form a single program, in that each example depends on the results of the previous examples.)*

- Create a new graph. The call `(make-graph)` should create a new graph object and return it. All of the graph's functionality should be encapsulated in the returned structure (hence, it will need to be a closure).

  ```
  Ex: (define G (make-graph))
  ```

- Add a vertex to the graph. The new vertex is represented by a Racket symbol. You may assume the same vertex will not be added twice.

  ```
  Ex: ((G 'add-vertex!) 'ohlendorf)
      ((G 'add-vertex!) 'rat)
      ((G 'add-vertex!) 'library)
      ((G 'add-vertex!) 'rhodes-tower)
  ```

- Get a list of all the vertices in a graph. Returns the empty list if there are no vertices in the graph.

  ```
  Ex: ((G 'get-vertices)) ==> '(ohlendorf rat library rhodes-tower)
           (the order of the vertices above is unimportant)
  ```

- Add an edge to the graph. You may assume there will be at most one edge connecting any two vertices in the graph, and the same edge will not be added twice. Once an edge is added, it cannot be removed, and its weight cannot be changed. You may assume the vertices given as arguments will have already been added to the graph, and the weight argument will be a positive integer.

  ```
  Ex: ((G 'add-edge!) 'ohlendorf 'rat 3)
      ((G 'add-edge!) 'rat 'rhodes-tower 7)
      ((G 'add-edge!) 'library 'rhodes-tower 5)
      ((G 'add-edge!) 'library 'ohlendorf 4)
  ```

- Get the weight attached to an edge in the graph. Given two vertices as arguments, return the weight of the edge connecting them. You may assume that the vertices given have an edge connecting them in the graph.

  ```
  Ex: ((G 'get-weight) 'ohlendorf 'rat) ==> 3
  ```

- Retrieve a list of the neighboring vertices of a given vertex. You may assume the vertex argument exists in the graph. Return the empty list if the vertex has no edges incident to it.

  ```
  Ex: ((G 'get-neighbors) 'ohlendorf) ==> '(rat library)
      ((G 'get-neighbors) 'rat) ==> '(ohlendorf rhodes-tower)
        (the order of the vertices within the returned list is unimportant)
  ```

- You may add any other methods to your graph class that you would like (and that make sense for a graph object to have), but the ones above are the ones that will be explicitly tested.

  For example, you may wish to have a `contains-vertex?` method that tests whether a vertex is in the graph, or a `get-edges` method that retrieves all the edges in the graph.

Part 2

After you've written your graph class, use it to implement Dijkstra's algorithm for finding the shortest path between two vertices in your graph. If you don't remember Dijkstra's algorithm from 241, here's some pseudocode:

```
; Assume we're searching graph G to find the shortest path
; from vertex "start" to vertex "finish."

; A "path" is a list of vertices along with the sum of the weights on the edges
; making up the path.

initialize "frontier" to a list containing just the degenerate path starting
at start, with sum of distances for this path = 0

while frontier is not empty:
  curpath = path with smallest sum of weights from the frontier
  remove curpath from frontier
  if curpath ends with finish, then return curpath as shortest path
  else if curpath ends with a vertex in the explored list, continue loop
  else:
    curvertex = curpath's ending vertex
    for each edge (curvertex, w) do:
      add new path (curpath + w) to the frontier, provided w is not
        in the explored list, and w is not already in curpath (prevent cycles)
    re-sort frontier so it is ordered from shortest path to longest path
    add curvertex to the explored list
```

You must implement Dijkstra's algorithm using a functional programming style, meaning you may not use mutation when writing the algorithm. The only place mutation should appear in this project is in implementing Part 1, the graph class.

The algorithm should return a list containing the total distance from the start vertex to the finish vertex, and the list of vertices along the way.
`Ex: (dijkstra G 'ohlendorf 'rhodes-tower) ==> '(9 ohlendorf library rhodes-tower)`

Here are some suggestions to make this project easier:

- Choose a way to represent paths in the frontier. I suggest a list that looks like `(total-distance v1 v2 v3...)` where the vertices are in *reverse* order from the start vertex. This way you can easily find the "current vertex" by using `cadr`, rather than having to look at the end of the list.
- Use the Racket function `sort` to keep the frontier sorted by smallest distance. See the Racket reference for how to use `sort`. That way you'll always have access to the path with the shortest path so far.
- I suggest having a helper function that takes the frontier and explored lists as arguments. That way, the while loop in the pseudocode becomes recursive calls to this helper function with updated frontier and explored lists.

- Use lots of helper functions. My solution uses helper functions that build new paths, check paths for cycles, check if a path ends with a vertex in the explored list, and so on.
- Map and filter come in handy here. You can use map to iterate over the neighbors of a vertex to create new paths, and filter to eliminate the ones that have cycles or end with a vertex in the explored list.
- Make liberal use of `(display)` or `(displayln)` calls to help debugging. (Remove the calls or comment them out before submitting your code.)
- You can use the `(reverse)` function at the very end to reverse the path so the vertices appear in the right order (this will be needed if you use my "backwards path" idea from above).
- You can compare symbols for equality using `eq?` or `equal?`. The plain old equals sign will not work, because that's only for numbers.

Another example

Last semester's AI students will recognize this one:

```
(define G3 (make-graph))
((G3 'add-vertex!) 'a)
((G3 'add-vertex!) 'b)
((G3 'add-vertex!) 'c)
((G3 'add-vertex!) 'd)
((G3 'add-vertex!) 'e)
((G3 'add-vertex!) 'f)
((G3 'add-vertex!) 'g)
((G3 'add-vertex!) 'h)
((G3 'add-vertex!) 'i)
((G3 'add-edge!) 'a 'b 2)
((G3 'add-edge!) 'a 'd 6)
((G3 'add-edge!) 'a 'c 3)
((G3 'add-edge!) 'b 'd 3)
((G3 'add-edge!) 'c 'd 6)
((G3 'add-edge!) 'd 'e 7)
((G3 'add-edge!) 'd 'h 3)
((G3 'add-edge!) 'd 'i 6)
((G3 'add-edge!) 'e 'f 6)
((G3 'add-edge!) 'f 'g 5)
((G3 'add-edge!) 'f 'i 3)
((G3 'add-edge!) 'g 'h 8)
((G3 'add-edge!) 'g 'i 3)
((G3 'add-edge!) 'h 'i 2)
(dijkstra G3 'a 'i) ==> '(10 a b d h i)
```

About mutable pairs

If you choose to use mutable pairs in this assignment (note that you don't have to, you can just use `set!`), you may want to add the line `(require racket/mpair)` to the top of your program. This imports some useful functions you can use, such as `mmap` (map for mutable lists), and `mlist->list` and `list->mlist`, which convert between mutable lists and immutable lists, because those are separate data types. You also get `mlist, mappend, mlength, mreverse` which are all equivalent to their regular Racket counterpart functions, but work on mutable lists.

Note that you can still mutate individual variables with `set!`, even if the variable is currently referencing an immutable list. You just can't use `set-mcar!` or `set-mcdr!` with immutable lists, only lists created with `mcons`, `mlist`, or `mappend`.

Assessment

Solutions should be:

- Correct
- In good style, including indentation and line breaks
- Written using features discussed in class. In particular, for Part B, you must not use any mutation operations except for the graph mutation functions you wrote for Part A. And truthfully, your `dijkstra` function itself doesn't need to call `add-edge!` or `add-vertex!` anyway – the set-up code for constructing the graph will make those calls.

Turn-in Instructions

- Put all your solutions in one file, `hw3_lastname_firstname.rkt`, where `lastname` is replaced with your last name, and `firstname` is replaced with your first name.
- Upload your file to Moodle before the project deadline.