**Kerry Ojakian's CSI 32 Class**

# Midterm Project List

- Do ONE of the projects listed below. As usual for all homework: Work on your own, without help from others, whether a person, someone online, or by copying code from some source (see the Class Contract that you signed).
- Design and program must be object-oriented.
- For each project there is a requirements section. You must follow the requirements section **exactly**! By follow **exactly**, I mean follow **exactly**, i.e. the names of the methods must be exactly as named in the requirements section (case included!), the order of the arguments to a method must be exactly as in the requirements section, etc.
- There must be NO graphics. These are all console applications.
- They should all be user-friendly and robust, being able to deal with user input errors; however, don't go overboard, get it working first!

## Project Descriptions

1. **Cop and Robber.**

   **Overview:** We describe the 2-player game called "Cop and Robber". There are two players: one is the cop and the other is the robber. The game can be played on any graph (i.e. a set of vertices, and edges that connect them to make a network). The game is put into initial position by first allowing the cop to place herself at any vertex, followed by allowing the robber to place himself at any vertex. After the players have both placed themselves, the players alternate taking turns, with the cop starting. On a turn, a player can move from a vertex to an adjacent vertex, or remain at the same vertex. The cop wins if she ever occupies the same vertex as the robber. The robber wins if the cop gives up, or the number of moves more than some user inputed number (for example 25 moves).

   **Requirements:** You must 1) create a class named **CopRobGame**, which will be described below, 2) create other classes as needed, which will all be known by the top level class **CopRobGame**, 3) write a simple console game (no graphics!) for the game using the class **CopRobGame** and a few simple calls to it.

   The class **CopRobGame**, must have *at least* the following methods (it may have more methods):

   (a) The initializer takes one positive integer argument, which is the limit on the number of moves before the game is over (and the robber wins).

(b) `createVertex`: Takes a single string input as an argument and adds this vertex to the graph, attaching it to no other vertices.

(c) `createEdge`: Takes 2 strings as input, and creates an edge between the 2 vertices with those string names. If the 2 strings are the same, or one of the strings is not the name of a vertex, then no edge is created. If the edge already exists, nothing new is created. Returns the boolean True if a new edge was created, and False otherwise.

(d) `placePlayer`: Takes 2 strings as input. The first string should be either "C" or "R", to indicate Cop or Robber, respectively. The second string should be the name of a vertex. The indicated player is placed at that vertex.

(e) `movePlayer`: Takes 2 string inputs; the first should be either "C" or "R", to indicate Cop or Robber, respectively. The second string should be the name of the vertex to move the player to. The method should return True if the player was successfully moved there, and False if there was some problem.

(f) `winCheck`: Takes no inputs and returns one of 3 strings: "C" if the cop and robber occupy the same vertex; "R" if the number of moves is past the limit; and "X" if neither player has won yet.

(g) `display`: This creates a text output of the current state of the graph and the positions of the players (if they are placed), in a the following series of lines, each separated by a single newline character.

   i. The string "Cop: " followed by the name of the vertex where the cop is; if the Cop has not been placed, then the string "Cop not placed." appears. Then there are some spaces, and an analogous printout for the Robber: "Robber: " followed by the name of the vertex where the robber is; if the Robber has not been placed, then the string "Robber not placed."

   ii. The string "Vertices: " followed by a comma separated listing of the vertices.

   iii. The string "Edges: " followed by a comma separated listing of the edges, where each edge is of the form "( [vertex name], [vertex name])".

**Example:** The following code should print or return what is in the comments.

```
G = CopRobGame(10)
G.createVertex('a')
G.createVertex('b')
G.createVertex('c')
G.createVertex('d')
G.createEdge('a', 'b')
G.createEdge('b', 'c')
G.createEdge('c', 'a')
G.createEdge('c', 'd') # Returns True
G.createEdge('d', 'c') # Returns False
G.placePlayer('C', 'c')
G.placePlayer('R', 'a')
print(G.winCheck()) # Prints 'X'
G.movePlayer('C', 'a')
print(G.winCheck()) # Prints 'C'
G.display() # PRINTS WHAT FOLLOWS ...

Cop: a    Robber: a
Vertices: a, b, c, d
Edges: (a,b) (b,c) (c,a) (a,d)
```

2. **Solitaire Peg Game.**

   **Overview:** In this program, you will implement a version of the game of Peg Solitaire. It is played on a rectangular grid of holes. Some of the holes will have pegs in them and some will not. On a turn, the player can jump a peg A over another peg B (in a horizontal or vertical fashion), if there is a space just after B and this space is unoccupied. Peg B is then removed from the board (but peg A stays). The user does this till she has no moves, the goal being to have as few pegs left as possible (having just one left is the best possible under some configurations).

   **Requirements:** You must 1) create a class named **PegSolitaire**, which will be described below, 2) create other classes as needed, which will all be known by the top level class **PegSolitaire**, 3) write a simple console application (no graphics!) that uses the class **PegSolitaire**, with a few simple method calls, to create an interactive game of Peg Solitaire.

   In the following discussion, by a **position** we mean a tuple (X, Y), where X and Y are positive integers; a position is meant to refer to a location on the board. The class **PegSolitaire**, must have *at least* the following methods (it may have more methods):

   (a) The initializer takes two positive integer arguments X and Y. This creates a X by Y grid of holes as the playing board.

   (b) `placePeg`: Takes integer arguments X and Y, and places a peg at row X and column Y.

   (c) `removePeg`: Takes integer arguments X and Y, and removes a peg at row X and column Y.

   (d) `jumpPeg`: Takes two *positions* and attempts to move the peg at the first position to the location indicated by the second position. If the move is legal, it is carried out, along with all appropriate adjustments to the board, and True is returned. If there is anything that forbids this move, the the method does not change the board, and returns False.

   (e) `isStuck`: Takes no inputs, and returns True if there are no possible moves left, and False if there are possible moves.

   (f) `display`: This creates a text output of the current state of the board. It should just be a grid of letters with the character "P" for a hole occupied by a peg, and "X" for a hole not occupied by a peg. An example of a possible result of display on a 2 by 3 board:

   ```
   XPP
   PXP
   ```

**Example:** The following code should print what is in the comments.

```
P = PegSolitaire(2,3)
P.placePeg(1,2)
P.placePeg(1,3)
P.placePeg(2,1)
P.placePeg(2,2)
P.placePeg(2,3)
P.removePeg(2,2)
P.display() # should print:
# XPP
# PXP

P.isStuck() # should return False
P.jumpPeg( (2,1), (2,2) ) # returns False and nothing happens.
P.jumpPeg( (1,3), (1,1) ) # returns True
P.isStuck() # should return True now
P.display() # should print:
# PXX
# PXP
```

3. **Subway Planner.**

   **Overview:** *This is a great project, but there is a technical issue you should discuss with me if you want to do it!* The user will be able to do two things: 1) Create a subway network, and 2) Ask about various routes. To create the network, the user can do the following:

   - Create a subway station with some name.
   - Connect two subway stations by some subway line, and indicate how long it takes to travel between those two stations using that line.

   Then the user can ask about routes, by entering the names of two subway stations, and the program will tell the user the fastest way to get from one to the other using the current subway network.

   **Requirements:** You must 1) create a class named **SubwayPlanner**, which will be described below, 2) create other classes as needed, which will all be known by the top level class **SubwayPlanner**, 3) write a simple console application (no graphics!) that uses the class **SubwayPlanner**, with a few simple method calls, to create an interactive subway planner.

   The class **SubwayPlanner**, must have *at least* the following methods (it may have more methods):

   (a) The initializer takes no inputs.

   (b) `createStation`: Takes a single string input as an argument and adds this station to the network, attaching it to no other stations.

   (c) `createLine`: Takes as input (A, B, L, T), where A, B, and L are strings, and T is a float. It creates a subway line between the 2 stations with names A and B. It connects A and B by the subway line with name L, and using line L the time it takes to go between A and B is time T (where T is measured in minutes). If A and B are the same, or one of A or B is not a station, then no subway line is created. If the exact subway line already exists (i.e. same A, B, and L) then nothing new is created. Returns the boolean True if a new subway line was created, and False otherwise.

   (d) `fastestRoute`: Takes two string inputs A and B. It returns two things: 1) The fastest travel time between A and B, and 2) A route that achieves this fastest time. The route should be returned as a list, where each entry of the list is a tuple (X, Y, L), where X is a station, Y is a station, and L is a line between X and Y.

   (e) `display`: This creates a text output of the current state of the network.

      i. The string "Stations: " followed by a comma separated listing of the stations.

ii. The string “Line: ” followed by a comma separated listing of the subways lines, where each subway line is of the form “( [station name], [station name], [line name], [travel time])”.

**Example:** The following code should print what is in the comments.

```
Subway = SubwayPlanner()
Subway.createStation('a')
Subway.createStation('b')
Subway.createStation('c')
Subway.createStation('d')
Subway.createLine('a', 'b', 'lineX', 1)
Subway.createLine('b', 'c', 'lineX', 2)
Subway.createLine('b', 'd', 'lineY', 10)
Subway.createLine('c', 'a', 'lineX', 50)
Subway.createLine('c', 'd', 'lineX', 8)
Subway.createLine('d', 'c', 'lineY', 3)

# Next line should return:
# 6, [ ('a','b','lineX'), ('b','c','lineX'), ('c','d','lineY') ]
Subway.fastestRoute('a', 'd')

Subway.display() # prints what follows ...

Stations: a, b, c, d
Lines:
(a, b, lineX, 1), (b, c, lineX, 2), (b, d, lineY, 10),
(c, a, lineX, 50), (c, d, lineX, 8), (d, c, lineY, 3)
```