CSE 231 **Fall 2021**

Computer Project #11

This assignment focuses on the design, implementation, and testing of a Python program using classes.

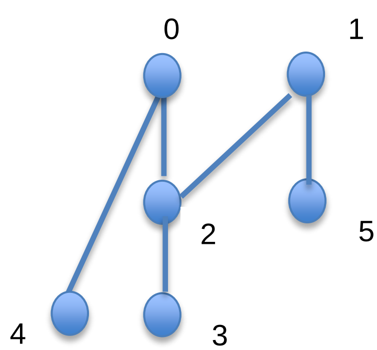
It is worth 55 points (5.5% of course grade) and must be completed no later than **11:59 PM on Wednesday, December 8, 2021.**

Assignment Background

How does Google maps provide a route to travel? How does a phone call connect to a distant phone? How does a packet of information find its way across the internet? The mathematical concept of graphs is the underlying data structure.

This project is going to be about graphs. For this project, we will consider the task of finding a route between two places.

Let us start by defining graphs. A ***graph*** is a collection of vertices. On paper, we represent vertices just by single points. These vertices are connected to each other by edges. Two vertices connected by edges can be seen as two points connected by lines. The following image shows an example of a graph.

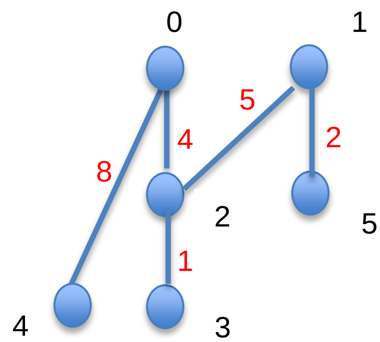


From this graph, we can conclude some simple things about it. First is that there are 6 vertices in total, and there are 5 edges in total. From the perspective of how the vertices are connected, to each other, we can make the following observations.

(a) Vertex 0 is connected to vertex 2 and vertex 4.
(b) Vertex 1 is connected to vertex 2 and vertex 5.
(c) Vertex 2 is connected to vertex 0, vertex 1 and vertex 3.
(d) Vertex 3 is connected to vertex 2 only.
(e) Vertex 4 is connected to vertex 0 only.
(f) Vertex 5 is connected to vertex 1 only.

We should note, to be mathematically precise, that in this graph we have only these edges and no other edges.

Such a model can be used to mimic several systems that we see around us. An example is as follows. We can have places (state, city, house) represented by vertices. We are going to model the edges as follows. For instance, if place ***a*** and place ***b*** are connected by a road directly, then we put an edge between vertex ***a*** and vertex ***b***. We can also show the length of that road. Note that one road is represented by one edge in the graph. We can show the length of that road by labelling the edge with a number. We show an example in the following figure.



We have colored red the numbers that we have used to label the edges. We also call those numbers the weight of the edges, in the sense that they represent the cost of reaching from one vertex to another (cost of travelling between those vertices). We can make the following observations about this graph.

(a) The cost of travelling between place 0 and place 2 is 4.
(b) The cost of travelling between place 0 and place 4 is 8.
(c) The cost of travelling between place 1 and place 2 is 5.
(d) The cost of travelling between place 1 and place 5 is 2.
(e) The cost of travelling between place 2 and place 3 is 1.

Remember that the weight of an edge, in this case, is representing the cost of travelling between a pair of vertices.

We can represent a graph in Python by using a 2-D list (two-dimensional list), also known as a matrix. Think about a matrix as an excel sheet. Each element in the outer list is a list that represents the row and each element in the inner list is a cell. We can do this as follows: Given that the number of vertices is ***n***, then the list that we construct will be an $n \times n$ list. For this project, all matrices will be square, that is every list in the two-dimensional list will be the same size. An example is as follows.

	0	1	2	3	4	5
0	0	0	**4**	0	**8**	0
1	0	0	**5**	0	0	**2**
2	**4**	**5**	0	**1**	0	0
3	0	0	**1**	0	0	0
4	**8**	0	0	0	0	0
5	0	**2**	0	0	0	0

Let this matrix (list-of-lists) be called ***G***. We call ***G*** the adjacency matrix of the graph drawn above. In ***G***, the cell at ***a***-th row and ***b***-th index of that row is equal to the weight of the edge that is connecting the vertex a and vertex b. So G[a][b] is equal to the weight of the edge, and so G[b][a] is also equal to the same weight. This symmetry makes our task easier. This symmetry also signifies that not only we can travel from a to b, we can also travel from b to a. For example,

(a) G[0][2] = G[2][0] = 4,
(b) G[1][5] = G[5][1] = 2,
(c) G[2][3] = G[3][2] = 1, and so on.

Observe that this matrix exactly corresponds to the graph drawn above.

Now let us suppose that we want to compute for each pair of places ***a*** and ***b***, the shortest distance between them. Let that ***G*** is the adjacency matrix constructed from the places and the distance to their nearest neighbors, as given in the matrix above.

For example, the distance matrix (called ***D***) of the above matrix will look like this.

	0	1	2	3	4	5
0	0	9	4	5	8	11
1	9	0	5	6	17	2
2	4	5	0	1	12	7
3	5	6	1	0	13	8
4	8	17	12	13	0	19
5	11	2	7	8	19	0

In ***D***, the cell at ***a***-th row and ***b***-th index of that row is the distance between a vertex a and vertex bW even if there is no edge between a and b. For example, there is no edge between vertex 0 and vertex 1. However, there is an edge between vertex 0 and vertex 2 and there is an edge between vertex 1 and 2. So the distance between vertex 0 and 1 is equal to the distance between 0 and 2 plus the distance between 2 and 1: 4 + 5 = 9.

Similarly, we can also construct the (shortest) path matrix for each pair of places ***a*** and ***b***. This matrix will contain a shortest path between place ***a*** and place ***b*** at position [***a***][***b***].

A path matrix (corresponding to the adjacency matrix and the distance matrix drawn above) looks like this:

	0	1	2	3	4	5
0	0	[0, 2, 1]	[0, 2]	[0, 2, 3]	[0, 4]	[0, 2, 1, 5]
1	[1, 2, 0]	0	[1, 2]	[1, 2, 3]	[1, 2, 0, 4]	[1, 5]
2	[2, 0]	[2, 1]	0	[2, 3]	[2, 0, 4]	[2, 1, 5]
3	[3, 2, 0]	[3, 2, 1]	[3, 2]	0	[3, 2, 0, 4]	[3, 2, 1, 5]
4	[4, 0]	[4, 0, 2, 1]	[4, 0, 2]	[4, 0, 2, 3]	0	[4, 0, 2, 1, 5]
5	[5, 1, 2, 0]	[5, 1]	[5, 1, 2]	[5, 1, 2, 3]	[5, 1, 2, 0, 4]	0

We provide the function to compute both the distance matrix and path matrix; this function takes the adjacency matrix as input, and returns both the distance matrix and path matrix.

This implies that, for example, if we have to travel from place 0 to place 5, then any shortest path will have a length of 11, and there is a shortest path which is a sequence for 4 places, namely, 0,2,1, and 5. (there can be other paths which are shortest paths, in the sense that their distance is equal to the shortest path that we have computed).

Assignment Specifications

The assignment contains the construction of the following class which goes in your `place.py` file. Do **not** put it in your `proj11.py` file.

`class` ***`Place`***`:`

(a) **`__init__(`*`self, name, i`*`):`**

Initialize ***`self.name`*** as the parameter ***`name`***, a string.
`self.index` is a unique identifier of this `Place`. Initialize ***`self.index`*** as the parameter ***`i`***, an int. This index identifies the index needed to index the "g" and "paths" parameters in set_distances and set_paths.
`self.dist` is the list of distances from other `Places`. Initialize ***`self.dist`*** as `None`. (We are initializing this list to `None` because we will be assigning a value to `self.dist,` not appending to an existing list. Using `None` helps us remember to not append.)
`self.paths` is the list of paths from other `Places.` Initialize ***`self.paths`*** as `None`.
`Argument: self, string, integer`
`Return: no return value`

(b) **`get_name(self):`**

This method will simply return ***`self.name`***.

`Argument: self`

`Return: string`

(c) **`get_index(self):`**

This method will simply return ***`self.index`***.

`Argument: self`

`Return: int`

(d) **`set_distances(self, g):`**

This method sets the distances of this `Place` from other `Places` in the list ***`self.dist`***. The parameter **`g`** is a list of lists of those distances. The value of the ***`self.dist`*** list is its index in the parameter **`g`**. That is, ***`self.dist`*** will be equal to the (***`self.index`***)-th list in **`g`**.

Hint: use `[:]` to make a copy when assigning ***`self.dist`*** so this `Place` instance will have its own copy of the list of distances from other `Places`. That is, use `g[self.index][:]`

`Argument: self, list of lists`

`Return: None`

(e) **`set_paths(self, paths):`**

This method is nearly identical to the `set_distances` method. This method sets the paths from this `Place` to other `Places` in the list ***`self.paths`***. The parameter **`paths`** is a list of lists of those paths. The value of the ***`self.paths`*** list is its index in the parameter **`paths`**. That is, ***`self.paths`*** will be equal to the (***`self.index`***)-th list in **`paths`**. As with the previous method, make a copy of the list of paths.

`Argument: self, list of lists of lists`

`Return: None`

(f) **`get_distance(self, j):`**

This method will return ***`self.dist`***`[`***`j`***`]` which is the distance between this `Place` instance, ***`self`***, and `Place` ***`j`*** in the list of distances to other `Places`.

`Argument: self, integer`

`Return: float`

(g) **`get_path(self, j):`**

This method will return ***`self.path`***`[`***`j`***`]` which is a shortest path between this `Place`, ***`self`***, and the `Place` ***`j`*** in the list of paths to other `Places`.

`Argument: self, integer`

`Return: list`

(h) **`__str__(self):`**

This method will return a formatted string representation of this `Place` instance. Begin by making a tuple

`tup = ( self.index, self.name, self.dist, self.paths)`

These are the only four values which identify a place.

Return a string using the following format on that tuple:

```
"Node {}, {}: distances {}, paths {}"
Argument: self
Return: string
```

Hint: We can use * to "move" items out of a list/tuple, and into a formatted string.

Ex:

```
my_list = ["I", "am", "awesome"]
"{} {} {}".format(*my_list)
```

→ I am awesome

(i) **`__repr__(self):`**

This method will also return an <u>unformatted</u> string representation of this `Place` instance. Create the same tuple created in `__str__` but simply return `str(tup).`

```
Argument: self
Return: string
```

The assignment contains the following functions that go in your `proj11.py` file.

(a) **`open_file():`**

This function repeatedly prompts the user for a filename until the file is opened successfully. An error message should be shown if the file cannot be opened. It returns a file pointer. Use try-except to determine if the file can be opened.

Argument: no argument

```
Return: <file_pointer>
```

(b) **`read_file(fp):`**

This function reads a file pointer ***`fp`*** and returns a list of tuples ***`L`***.

Each row of the file contains a pair of places and the distance between them (e.g., via a road connected them both). Each tuple will have the following structure:

(<place_1>,<place_2>,<distance between place_1 and place_2>)

place_1 and ***place_2*** must be arranged strictly as written in the file.

Remember that the places are strings and the distance between them is an int.

So your tuple will have

(str, str, int)

For example, the file `map0.csv` is

```
City 1,City 2,Distance
A,X,7
A,E,8
Y,X,5
Y,F,2
X,D,1
```

So the list of tuples returned will be

```
[('A', 'X', 7), ('A', 'E', 8), ('Y', 'X', 5), ('Y', 'F', 2), ('X', 'D', 1)]
```

```
Argument: <file_pointer>
Return: <list_of_tuples>
```

(c) **`adjacency_matrix_construction(`*`L`*`):`**

This function takes a list of tuples ***L*** as argument where each tuple is of the structure we discussed in **read_file()**). It will return two items: a list of places in L, and the adjacency matrix of distances between places.

First, build the list of places (you will need this list for the second part of this function).
Let's name the list: `places_lst.` It is a list of strings.
Read through the list of tuples `L` and make a list of all the places that appear in any tuple.
No place can be repeated in this list. Sort the list alphabetically (even if the places are digits).
Hint: a set is a useful intermediate data structure to get a collection of unique items.

For example, if L (from `map0.csv`) is

```
L = [('A', 'X', 7), ('A', 'E', 8), ('Y', 'X', 5), ('Y', 'F', 2), ('X', 'D', 1)]
```

Then `places_lst` will be

```
places_lst = ['A', 'D', 'E', 'F', 'X', 'Y']
```

Let that the length of ***places_lst*** be ***n***.
Create an ***n*** x ***n*** matrix ***g*** initialized to zeros. Remember that an ***n*** x ***n*** matrix can be represented by a list of lists where each row is a list. In this case, there will be ***n*** rows where each row will be initialized to be a list of ***n*** zeros. So a 6x6 matrix will be initialized as

```
g = [ [0,0,0,0,0,0], [0,0,0,0,0,0], [0,0,0,0,0,0], [0,0,0,0,0,0],
[0,0,0,0,0,0], [0,0,0,0,0,0]]
```

Next we need to fill the matrix ***g*** with values (noting that many zeros will remain as zeros).
The information we need is in the parameter `L`, the list of tuples, and in `places_lst`.
We use `places_lst` to get a place's index, e.g. 'A' is at index 0; 'X' is at index 4, and so on.
We use `L` to get the distance, so the first tuple of `L` is `('A', 'X', 7)`which tells us that the distance between 'A' and 'X' is 7. Since 'A' is at index 0 in `places_lst` and 'X' is at index 4 in `places_lst`, we will assign `g[0][4] = 7` and since the distance from 'A' to 'X' is the same as the distance from 'X' to 'A' we also assign `g[4][0] = 7`. Walk through every tuple in L, find the indices of the places in the tup, and then assign distances in ***g***.

In this example, the resulting adjacency matrix ***g*** will be:

```
[  [0  , 0  , 8  , 0  , 7  , 0 ]
   [0  , 0  , 0  , 0  , 1  , 0 ]
   [8  , 0  , 0  , 0  , 0  , 0 ]
   [0  , 0  , 0  , 0  , 0  , 2 ]
   [7  , 1  , 0  , 0  , 0  , 5 ]
   [0  , 0  , 0  , 2  , 5  , 0 ] ]
```

`Argument: <list_of_tuples>`
`Return: <list_of_strings>, <list_of_lists_of_integers>`

(d) **`make_objects(`*`places_lst`*`, `*`g`*`):`**

This function takes the list of places `places_lst` and the adjacency matrix ***g*** as arguments and returns a dictionary of Place objects that are in the places_lst. Actually, we will return two dictionaries: one indexed by name and one indexed by id because sometimes we want to find a Place by name and sometimes by id.

This function will invoke the provided function apsp() on top of your draft file by passing ***g*** to it, and it will take the return value in variables ***g***, ***paths***. The variable ***g*** is a matrix that now contain the shortest distances between each pair of places; whereas ***paths*** is a matrix that contain a shortest path between each pair of places.

Next create the dictionaries of Place objects:

for each index ***i*** of `places_lst`, you will (Hint: use `for … enumerate`)

a) create an object ***a_place*** for a place by initializing it with the place's name (found in `places_lst`***`[i]`*** and the place's index ***`i`***
b) Set the variable ***self.dist*** of this place by invoking the **set_distances()** method for your ***a_place*** object by passing ***g*** to it.
c) Set the variable ***self.path*** of this place by invoking the **set_paths()** method for your ***a_place*** object by passing ***paths*** to it.
d) Now add your a_place object to the indexed-by-name dictionary and the other a_place object to the indexed-by-id dictionary. Note that because the places_lst was created to have no duplicates (part of its specification), you do not have to worry about handling duplicates when creating dictionary entries.

`Argument: <list_of_strings>, <list_of_lists_of_integers>`
`Return: <dict_of_objects>, <dict_of_objects>`

(e) main()

(i) Open file. Invoke **open_file()**. Store the return value in ***fp***.

(ii) Read the file. Invoke ***read_file()*** by passing the argument ***fp*** to it. Store the return value in ***L***.

(iii) Construct the adjacency matrix. Invoke **adjacency_matrix()** by passing the argument ***L*** to it. Store the return values in ***places_lst***, ***g***.

(iv) Make place objects. Invoke **make_objects()** by passing the argument***s places_lst and g*** to it. Store the return value in ***name_dict_of_places, id_dict_of_places***.
You now have a dictionary of Place objects (two dictionaries to make life easier).

(v) Execute the following instructions again and again until the user terminates.

A. Display banner

B. Enter starting place. If it is `'q'` , end the program, otherwise check the validity of the outputIf not valid reprompt until it is valid.

C. Loop until the user enters `end'.

  a. Enter next destination (a name as a string)
     Error check: destination is in `places_lst` and this destination is not the same as the previous destination

  b. Store each destination (as a name string) in a list ***route_lst***

  You now have an ordered list of destinations on your route.
  It is simply a list of names (strings).

D. Taking stock: You will need to determine two things that you need in Step E. This code goes along with Step E, not separately. They are essentially the same step, but we broke it down for understanding.

  a. A **path** containing intermediate destinations to get to places on your route. For example, to drive from Lansing to Chicago the shortest path may take you by Benton Harbor. You need to find Lansing's Place object and get the path from Lansing to Chicago with the call:
     `Lansing_Place_Object.get_path('Chicago')`
     but instead of "Chicago" you will use Chicago's ID so it will actually be
     `Lansing_Place_Object.get_path(Chicago_ID)`
     How do you find Lansing's Place Object? You have a dictionary of place objects named ***name_dict_of_places*** that you can index by name so Lansing's Place Object will be at ***name_dict_of_places***['Lansing'].
     Similarly, you can find Chicago's Place Object and then use that to get its ID using `Chicago_Place_Object.get_index().`
     Note that its ID is actually an index into the adjacency matrix.

  b. A **distance**. Continuing our example, you need to find the distance from Lansing to Chicago which you can find in
     `Lansing_Place_Object.get_distance(Chicago_ID)`
     (If you are wondering, the path such as through Benton Harbor is already considered in this distance value.)

E. Initialize a path list. Initialize a distance. Then loop through your route list

  a. Add to your path: the intermediate nodes to the next destination
     Error check: if there is no path, print an error message (see Note II below)

  b. Add to the distance: the distance to the next destination.

F. If there is a path, output the path and its distance

Hints:

I. The coding standard for CSE 231 is posted on the course website:
   http://www.cse.msu.edu/~cse231/General/coding.standard.html
   Items 1-9 of the Coding Standard will be enforced for this project.

II. We have provided a function **apsp()**.
apsp(*g*) This function will take the adjacency matrix ***g*** as argument, modifies ***g*** such that ***g*** will now correspond to the shortest distances between each pair of places. This function also constructs a list of lists of lists ***paths*** corresponding to ***g*** such that ***paths*** will contain a shortest path between each pair of places. It returns the newly constructed ***g*** and ***paths***.
`Argument: list of lists`
`Return list of lists, list of list of lists`

III. If a pair of places is not connected, say the ***a***-th and the ***b***-th places in ***l_places***, then ***g***[***a***][***b***] will be 0.

IV. You may want to build your paths list in main(0 by iterating through `route_lst` backwards.

Assignment Deliverable

The deliverables for this assignment are the following TWO files:

`proj11.py` – the source code for your Python program
`place.py` – the file with your `Place` class

Be sure to use the specified file name and to submit it for grading via the **Mimir system** before the project deadline.

Grading Rubric

```
Computer Project #11                                  Scoring Summary
General Requirements:
  ( 4 pts) Coding Standard 1-9
     (descriptive comments, function headers, mnemonic identifiers,
     format, etc...)

Implementation:
 ( 9 pts)  Place Class test
 ( 3 pts)  open_file function (No Mimir tests)
 ( 5 pts)  read_file function
 ( 8 pts)  adjacency_matrix function
 ( 10 pts)  make_objects function
 ( 4 pts)  Test 1
 ( 4 pts)  Test 2
 ( 4 pts)  Test 3
 ( 4 pts)  Test 4
```

Test cases

<u>Test Case 1</u>

```
Enter the file name: map1.csv

Begin the search!
Enter starting place, enter 'q' to quit: 0
Enter next destination, enter ”end” to exit: 3
Enter next destination, enter ”end” to exit: 2
Enter next destination, enter ”end” to exit: 5
Enter next destination, enter ”end” to exit: end
Your route is:
       0
       2
       3
       2
       1
       5
Total distance = 13

Begin the search!
Enter starting place, enter 'q' to quit: q
Thanks for using the software
```

<u>Test Case 2</u>

```
Enter the file name: map2.csv

Begin the search.!
Enter starting place, enter 'q' to quit: 0
Enter next destination, enter ”end” to exit: 3
Enter next destination, enter ”end” to exit: 2
Enter next destination, enter ”end” to exit: 5
Enter next destination, enter ”end” to exit: end
Places 0 and 3 not connected.
Places 2 and 5 not connected.

Begin the search.!
Enter starting place, enter 'q' to quit: q
Thanks for using the software
```

<u>Test Case 3</u>

```
Enter the file name: map3.csv

Begin the search.!
Enter starting place, enter 'q' to quit: Ann Arbour
```

```
This place is not in the list.!
Enter starting place, enter 'q' to quit: Ann Arbor
Enter next destination, enter "end" to exit: Warren
Enter next destination, enter "end" to exit: Detroit
Enter next destination, enter "end" to exit: Detroit
This destination is not valid or is the same as the previous
destination!Enter next destination, enter "end" to exit: end
Your route is:
       Ann Arbor
       Warren
       Detroit
Total distance = 63

Begin the search.!
Enter starting place, enter 'q' to quit: Marquette
Enter next destination, enter "end" to exit: Flint
Enter next destination, enter "end" to exit: Lansing
Enter next destination, enter "end" to exit: Grand Rapids
Enter next destination, enter "end" to exit: end
places Marquette and Flint not connected.

Begin the search.!
Enter starting place, enter 'q' to quit: Lansing
Enter next destination, enter "end" to exit: Flint
Enter next destination, enter "end" to exit: Lansing
Enter next destination, enter "end" to exit: Grand Rapids
Enter next destination, enter "end" to exit: end
Your route is:
       Lansing
       Flint
       Lansing
       Grand Rapids
Total distance = 172

Begin the search.!
Enter starting place, enter 'q' to quit: Detroit
Enter next destination, enter "end" to exit: Kalamazoo
Enter next destination, enter "end" to exit: Traverse City
Enter next destination, enter "end" to exit: Kalamazoo
Enter next destination, enter "end" to exit: end
Your route is:
       Detroit
       Kalamazoo
       Grand Rapids
       Traverse City
       Grand Rapids
       Kalamazoo
Total distance = 523
```

```
Begin the search.!
Enter starting place, enter 'q' to quit: q
Thanks for using the software
```

<u>Test Case 4</u>

```
Enter the file name: map4.csv
Begin the search.!
Enter starting place, enter 'q' to quit: New York
Enter next destination, enter "end" to exit: Indiana
Enter next destination, enter "end" to exit: Arizona
Enter next destination, enter "end" to exit: Michigan
Enter next destination, enter "end" to exit: end
Your route is:
       New York
       Pennsylvania
       Ohio
       Indiana
       Illinois
       Missouri
       Oklahoma
       New Mexico
       Arizona
       New Mexico
       Oklahoma
       Missouri
       Illinois
       Michigan
Total distance = 3925

Begin the search.!
Enter starting place, enter 'q' to quit: Michigan
Enter next destination, enter "end" to exit: Washington
Enter next destination, enter "end" to exit: Alaska
Enter next destination, enter "end" to exit: end
places Washington and Alaska not connected.

Begin the search.!
Enter starting place, enter 'q' to quit: Michigan
Enter next destination, enter "end" to exit: Hawaii
Enter next destination, enter "end" to exit: Louisiana
Enter next destination, enter "end" to exit: Minnesota
Enter next destination, enter "end" to exit: end
places Michigan and Hawaii not connected.
places Hawaii and Louisiana not connected.

Begin the search.!
Enter starting place, enter 'q' to quit: Michigan
```

```
Enter next destination, enter "end" to exit: Idaho
Enter next destination, enter "end" to exit: Louisiana
Enter next destination, enter "end" to exit: Minnesota
Enter next destination, enter "end" to exit: end
Your route is:
       Michigan
       Minnesota
       North Dakota
       Montana
       Idaho
       Utah
       New Mexico
       Texas
       Louisiana
       Arkansas
       Missouri
       Iowa
       Minnesota
Total distance = 4358

Begin the search.!
Enter starting place, enter 'q' to quit: q
Thanks for using the software
```