Programming Fundamentals Using Python

2018

Problem Set 11

Most recent updated: July 15, 2018

Objectives

1. Graph
2. Breadth First Search

Note: Solve the programming problems listed using your favorite text editor. Make sure you save your programs in files with suitably chosen names, **and try as much as possible to write your code with good style (see the style guide for python code)**. In each problem find out a way to test the correctness of your program. After writing each program, test it, debug it if the program is incorrect, correct it, and repeat this process until you have a fully working program. Show your working program to one of the cohort instructors.

Problems: Cohort sessions

1. *Graph* See Figure 1 and answer the following:

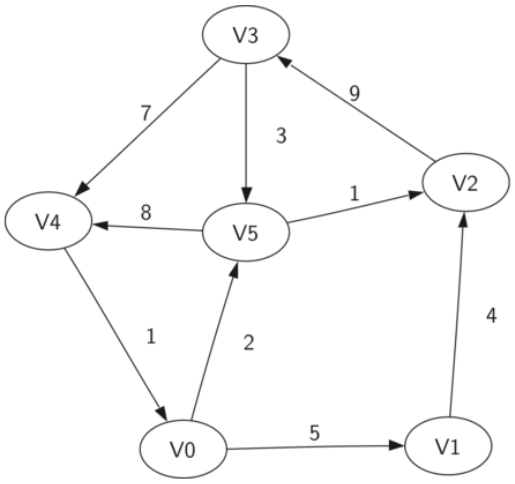

Figure 1: Example of a Graph

   (a) How many vertex are there?

   (b) How many edges are there?

   (c) What is weight of the edge from vertex 5 to vertex 4?

   (d) What are the paths from vertex 1 to vertex 4?

   (e) Give one cycle in the graph.

2. *Adjacency Matrix* Draw the graph from the Adjacency Matrix given in Figure 2.

3. *Graph* Implement a Graph ADT. The class has the following attributes:

   (a) `vertices`: a list of vertices.

   (b) `num_vertices`: to give the number of vertices in the graph.

   It should support the following methods:

   (a) `add_vertex(vertex)`: To add vertex into the graph.

   (b) `add_edge(from_v, to_v)`: To add an edge from one vertex to another vertex.

   (c) `add_edge(from_v, to_v, weight)`: To add a weighted edge from one vertex to another vertex.

   (d) `get_vertex(key)`: to find a vertex with a given key.

   (e) `get_vertices()`: to return a list of all vertices in the graph.

	V0	V1	V2	V3	V4	V5
V0		5				2
V1			4			
V2				9		
V3					7	3
V4	1					
V5			1		8	

Figure 2: Adjacency Matrix of a Graph.

(f) `__contains__(key)`: to find if a vertex with a given key is in the graph.

(g) `__iter__()`: to iterate over all the vertices in the graph.

The vertex is defined in class `Vertex`, which has the following attributes:

(a) `key`: the key or id of the vertex.

(b) `connected_to`: a dictionary where the keys are the vertices that this Vertex is connected to and the values are the weight of that edge.

It should support the following methods:

(a) `add_neighbour(vertex, weight = 0)`: to add an neighbour to the current vertex with a given weight at the edge.

(b) `get_connections()`: to return a list of all the vertices that the current vertex is connected to.

(c) `get_key()`: to return the key or id of the current vertex.

(d) `get_weight(neighbour_v)`: to return the weight of an edge connecting the current vertex to the given neighbour.

(e) `__str__()`: to return a string representation that describes the vertex. E.g. "Vertex 2 is connected to 3, 4, 5."

4. *Building Graph* Let's implement the Word Ladder puzzle. In this puzzle we are to transform the word FOOL into the word SAGE. In a word ladder puzzle you must make the change occur gradually by changing one letter at a time. At each step you must transform

one word into another word, you are not allowed to transform a word into a non-word. The word ladder puzzle was invented in 1878 by Lewis Carroll, the author of Alice in Wonderland. The following sequence of words shows one possible solution to the problem posed above.

- FOOL
- POOL
- POLL
- POLE
- PALE
- SALE
- SAGE

You are to represent the relationships between the words as a graph. Words that has one letter different from one another are connected by an edge. Write a function that takes in a list of words and return a graph that describes the above relationship. Use the template provided that loads the list of words from a pickle object. Note that when you build the Graph, make sure you ignore the case, i.e. cat and Cat are the same word. You can first convert all words to upper case letter when building the graph.

5. *VertexSearch* Do the following tasks:

   (a) Create a class `VertexSearch` which is a child class of `Vertex`. The class `VertexSearch` has three additional properties: `distance`, `predecessor`, and `color`. Make sure that its setter and getter work with these properties. Implement also the `__lt__(self, other)` in VertexSearch— in such a way that the vertex is compared based on its key. This will be useful if we need to sort the vertices.

   (b) Create also a new class `GraphSearch` that uses `VertexSearch` instead of `Vertex`. Implement the class `GraphSearch` as a child class `Graph`.

   (c) Create a child class of `GraphSearch` called `GraphSearchUndirected` to represent undirected graph. In this child class, override the method `add_edge(v1, v2, weight=0)` so that v1 is connected to v2 and v2 is connected to v1.

6. *Breadth First Search* Create a function `bfs(graph, start_vert)` that modifies `graph` attributes (distance, predecessor, and color) as it does breadth first search from the starting vertex.

7. *Traversing Node* Write a function `traverse(key)` that finds the path coming to the vertex `key` from a starting vertex. The starting vertex is set when calling the function `bfs()`. This function should return a list from the starting vertex to the vertex of a given key.

End of Problem Set 11.