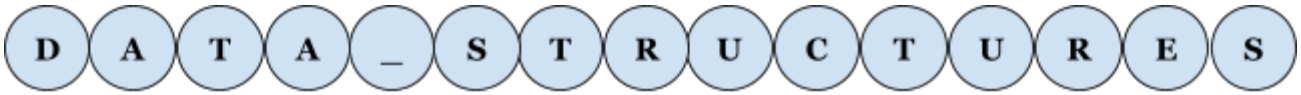
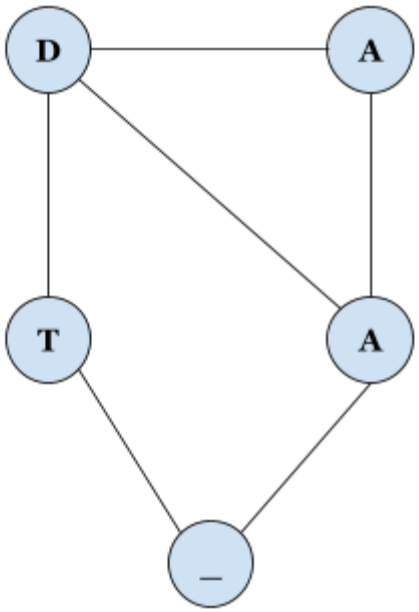
CS261 Data Structures

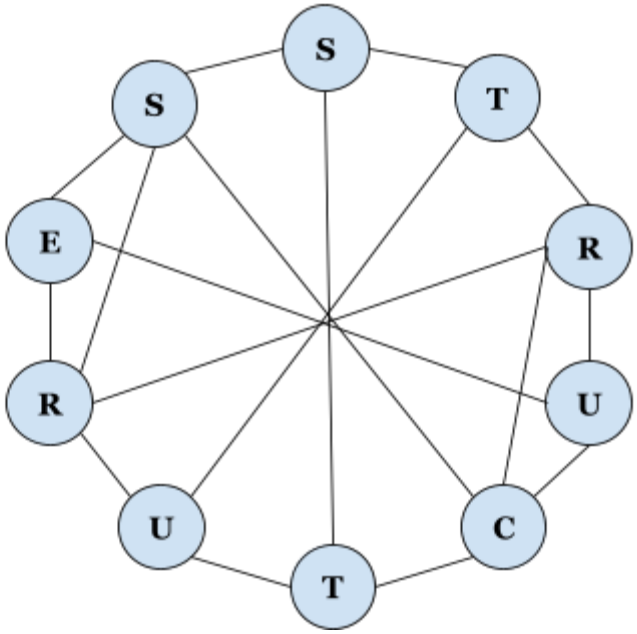
Assignment 6

v 1.10 (revised 8/2/2021)

Your Very Own Graphs







Contents

General Instructions

1. Programs in this assignment must be written in Python v3 and submitted to Gradescope before the due date specified in the syllabus. You may resubmit your code as many times as necessary. Gradescope allows you to choose which submission will be graded.

2. In Gradescope, your code will run through several tests. Any failed tests will provide a brief explanation of testing conditions to help you with troubleshooting. Your goal is to pass all tests.

3. We encourage you to create your own test programs and cases even though this work won't have to be submitted and won't be graded. Gradescope tests are limited in scope and may not cover all edge cases. Your submission must work on all valid inputs. We reserve the right to test your submission with more tests than Gradescope.

4. Your code must have an appropriate level of comments. At a minimum, each method should have a descriptive docstring. Additionally, put comments throughout the code to make it easy to follow and understand.

5. You will be provided with a starter "skeleton" code, on which you will build your implementation. Methods defined in skeleton code must retain their names and input / output parameters. Variables defined in skeleton code must also retain their names. We will only test your solution by making calls to methods defined in the skeleton code and by checking values of variables defined in the skeleton code.

   You can add more helper methods and variables, as needed. You also are allowed to add optional default parameters to method definitions.

   However, certains classes and methods cannot be changed in any way. Please see comments in the skeleton code for guidance. In particular, content of any methods pre-written for you as part of the skeleton code must not be changed.

6. Both the skeleton code and code examples provided in this document are part of assignment requirements. They have been carefully selected to demonstrate requirements for each method. Refer to them for the detailed description of expected method behavior, input / output parameters, and handling of edge cases. Code examples may include assignment requirements not explicitly stated elsewhere.

7. For each method, you can choose to implement a recursive or iterative solution. When using a recursive solution, be aware of maximum recursion depths on large inputs. We will specify the maximum input size that your solution must handle.

8. We will test your implementation with different types of objects, not just integers. We guarantee that all such objects will have correct implementation of methods __eq__, __lt__, __gt__, __ge__, __le__ and __str__.

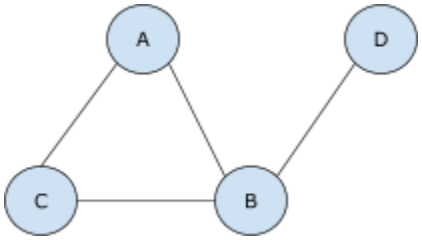
Part 1 - Summary and Specific Instructions

1. Implement the UndirectedGraph class by completing the provided skeleton code in the file `ud_graph.py`. UndirectedGraph class is designed to support the following type of graph: undirected, unweighted, no duplicate edges, no loops. Cycles are allowed.

2. Once completed, your implementation will include the following methods:

```
add_vertex(), add_edge()
remove_edge(), remove_vertex()
get_vertices(), get_edges()
is_valid_path(), dfs(), bfs()
count_connected_components(), has_cycle()
```

3. Undirected graphs should be stored as a Python dictionary of lists where keys are vertex names (strings) and associated values are Python lists with names (in any order) of vertices connected to the 'key' vertex. So for the graph pictured below:



self.adj_list = {'A': ['B', 'C'], 'B': ['A', 'C', 'D'], 'C': ['B', 'A'], 'D': ['B']}

4. The number of vertices in the graph will be between 0 and 900 inclusive. The number of edges will be less than 10,000.

5. RESTRICTIONS: For this assignment, you **ARE** allowed to use any built-in data structures and methods from the Python standard library. In addition, you are also allowed to add the following import statements:

```
import heapq
from collections import deque
```

   You are NOT allowed to import ANY other libraries / modules, especially those that provide functionality for working with graphs.

6. Variables in the UndirectedGraph class are not private. You ARE allowed to access and change their values directly.

add_vertex(self, v: str) -> None:

This method adds a new vertex to the graph. Vertex names can be any string. If a vertex with the same name is already present in the graph, the method does nothing (no exception needs to be raised).

add_edge(self, u: str, v: str) -> None:

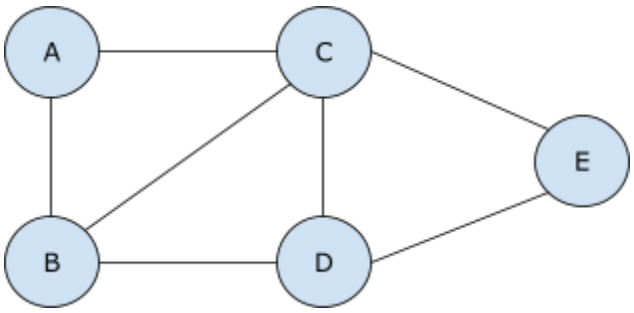
This method adds a new edge to the graph, connecting the two vertices with the provided names. If either (or both) vertex names do not exist in the graph, this method will first create them and then create an edge between them. If an edge already exists in the graph, or if `u` and `v` refer to the same vertex, the method does nothing (no exception needs to be raised).

Example #1:

```
g = UndirectedGraph()
print(g)

for v in 'ABCDE':
    g.add_vertex(v)
print(g)

g.add_vertex('A')
print(g)

for u, v in ['AB', 'AC', 'BC', 'BD', 'CD', 'CE', 'DE', ('B', 'C')]:
    g.add_edge(u, v)
print(g)
```

Output:

```
GRAPH: {}
GRAPH: {A: [], B: [], C: [], D: [], E: []}
GRAPH: {A: [], B: [], C: [], D: [], E: []}
GRAPH: {
  A: ['B', 'C']
  B: ['A', 'C', 'D']
  C: ['A', 'B', 'D', 'E']
  D: ['B', 'C', 'E']
  E: ['C', 'D']}
```



remove_edge(self, u: str, v: str) -> None:

This method removes an edge between the two vertices with provided names. If either (or both) vertex names do not exist in the graph, or if there is no edge between them, the method does nothing (no exception needs to be raised).

remove_vertex(self, v: str) -> None:

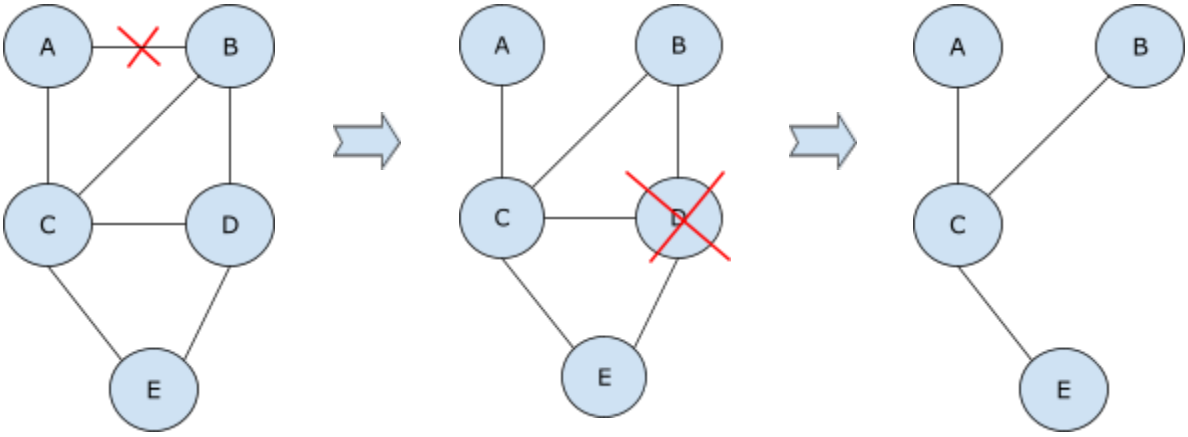
This method removes a vertex with a given name and all edges incident to it from the graph. If the given vertex does not exist, the method does nothing (no exception needs to be raised).

Example #1:

```
g = UndirectedGraph(['AB', 'AC', 'BC', 'BD', 'CD', 'CE', 'DE'])
g.remove_vertex('DOES NOT EXIST')
g.remove_edge('A', 'B')
g.remove_edge('X', 'B')
print(g)
g.remove_vertex('D')
print(g)
```

Output:

```
GRAPH: {
  A: ['C']
  B: ['C', 'D']
  C: ['A', 'B', 'D', 'E']
  D: ['B', 'C', 'E']
  E: ['C', 'D']}
GRAPH: {A: ['C'], B: ['C'], C: ['A', 'B', 'E'], E: ['C']}
```



get_vertices(self) -> []:

This method returns a list of vertices of the graph. The order of the vertices in the list does not matter.

get_edges(self) -> []:

This method returns a list of edges in the graph. Each edge is returned as a tuple of two incident vertex names. The order of the edges in the list or the order of the vertices incident to each edge does not matter.

Example #1:

```
g = UndirectedGraph()
print(g.get_edges(), g.get_vertices(), sep='\n')
g = UndirectedGraph(['AB', 'AC', 'BC', 'BD', 'CD', 'CE'])
print(g.get_edges(), g.get_vertices(), sep='\n')
```

Output:

```
[]
[]
[('A', 'B'), ('A', 'C'), ('B', 'C'), ('B', 'D'), ('C', 'D'), ('C', 'E')]
['A', 'B', 'C', 'D', 'E']
```

is_valid_path(self, path: []) -> bool:

This method takes a list of vertex names and returns True if the sequence of vertices represents a valid path in the graph (so one can travel from the first vertex in the list to the last vertex in the list, at each step traversing over an edge in the graph). An empty path is considered valid.

Example #1:

```
g = UndirectedGraph(['AB', 'AC', 'BC', 'BD', 'CD', 'CE', 'DE'])
test_cases = ['ABC', 'ADE', 'ECABDCBE', 'ACDECB', '', 'D', 'Z']
for path in test_cases:
    print(list(path), g.is_valid_path(list(path)))
```

Output:

```
['A', 'B', 'C'] True
['A', 'D', 'E'] False
['E', 'C', 'A', 'B', 'D', 'C', 'B', 'E'] False
['A', 'C', 'D', 'E', 'C', 'B'] True
[] True
['D'] True
['Z'] False
```

dfs(self, v_start: str, v_end=None) -> []:

This method performs a depth-first search (DFS) in the graph and returns a list of vertices visited during the search, in the order they were visited. It takes one required parameter, name of the vertex from which the search will start, and one optional parameter - name of the 'end' vertex that will stop the search once that vertex is reached.

If the starting vertex is not in the graph, the method should return an empty list (no exception needs to be raised). If the name of the 'end' vertex is provided but is not in the graph, the search should be done as if there was no end vertex.

When several options are available for picking the next vertex to continue the search, your implementation should pick the vertices in ascending lexicographical order (so, for example, vertex 'APPLE' is explored before vertex 'BANANA').

bfs(self, v_start: str, v_end=None) -> []:

This method works the same as DFS above, except it implements a breadth-first search.

Example #1:

```
edges = ['AE', 'AC', 'BE', 'CE', 'CD', 'CB', 'BD', 'ED', 'BH', 'QG', 'FG']
g = UndirectedGraph(edges)
test_cases = 'ABCDEGH'
for case in test_cases:
    print(f'{case} DFS:{g.dfs(case)} BFS:{g.bfs(case)}')
print('-----')
for i in range(1, len(test_cases)):
    v1, v2 = test_cases[i], test_cases[-1 - i]
    print(f'{v1}-{v2} DFS:{g.dfs(v1, v2)} BFS:{g.bfs(v1, v2)}')
```

Output:

```
A DFS:['A', 'C', 'B', 'D', 'E', 'H'] BFS:['A', 'C', 'E', 'B', 'D', 'H']
B DFS:['B', 'C', 'A', 'E', 'D', 'H'] BFS:['B', 'C', 'D', 'E', 'H', 'A']
C DFS:['C', 'A', 'E', 'B', 'D', 'H'] BFS:['C', 'A', 'B', 'D', 'E', 'H']
D DFS:['D', 'B', 'C', 'A', 'E', 'H'] BFS:['D', 'B', 'C', 'E', 'H', 'A']
E DFS:['E', 'A', 'C', 'B', 'D', 'H'] BFS:['E', 'A', 'B', 'C', 'D', 'H']
G DFS:['G', 'F', 'Q'] BFS:['G', 'F', 'Q']
H DFS:['H', 'B', 'C', 'A', 'E', 'D'] BFS:['H', 'B', 'C', 'D', 'E', 'A']
-----
B-G DFS:['B', 'C', 'A', 'E', 'D', 'H'] BFS:['B', 'C', 'D', 'E', 'H', 'A']
C-E DFS:['C', 'A', 'E'] BFS:['C', 'A', 'B', 'D', 'E']
D-D DFS:['D'] BFS:['D']
E-C DFS:['E', 'A', 'C'] BFS:['E', 'A', 'B', 'C']
G-B DFS:['G', 'F', 'Q'] BFS:['G', 'F', 'Q']
H-A DFS:['H', 'B', 'C', 'A'] BFS:['H', 'B', 'C', 'D', 'E', 'A']
```

count_connected_components(self) -> int:

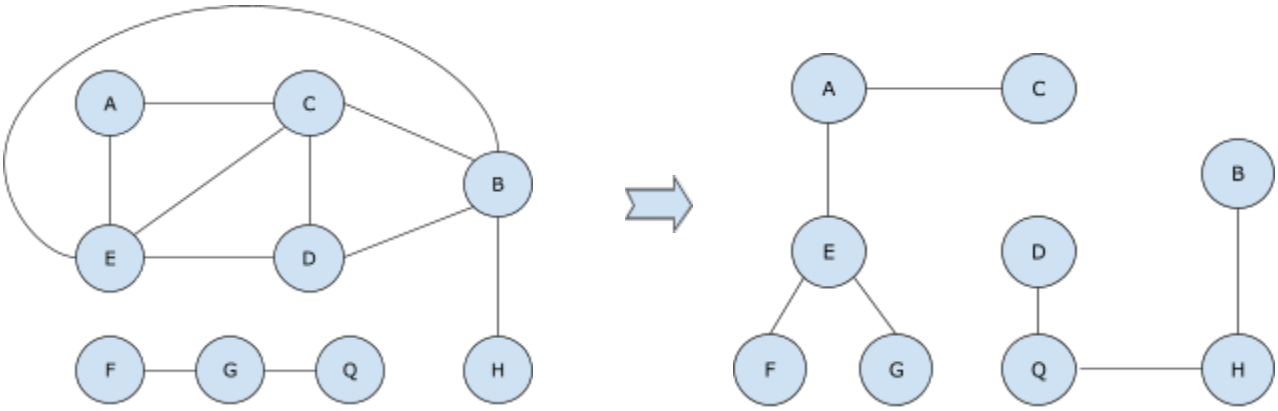
This method returns the number of connected components in the graph.

Example #1:

```
edges = ['AE', 'AC', 'BE', 'CE', 'CD', 'CB', 'BD', 'ED', 'BH', 'QG', 'FG']
g = UndirectedGraph(edges)
test_cases = (
    'add QH', 'remove FG', 'remove GQ', 'remove HQ',
    'remove AE', 'remove CA', 'remove EB', 'remove CE', 'remove DE',
    'remove BC', 'add EA', 'add EF', 'add GQ', 'add AC', 'add DQ',
    'add EG', 'add QH', 'remove CD', 'remove BD', 'remove QG')
for case in test_cases:
    command, edge = case.split()
    u, v = edge
    g.add_edge(u, v) if command == 'add' else g.remove_edge(u, v)
    print(g.count_connected_components(), end=' ')
```

Output:

```
1 2 3 4 4 5 5 5 6 6 5 4 3 2 1 1 1 1 1 2
```



BEFORE ALL OPERATIONS AFTER ALL OPERATIONS

has_cycle(self) -> bool:

This method returns True if there is at least one cycle in the graph. If the graph is acyclic, the method returns False.

Example #1:

```
edges = ['AE', 'AC', 'BE', 'CE', 'CD', 'CB', 'BD', 'ED', 'BH', 'QG', 'FG']
g = UndirectedGraph(edges)
test_cases = (
    'add QH', 'remove FG', 'remove GQ', 'remove HQ',
    'remove AE', 'remove CA', 'remove EB', 'remove CE', 'remove DE',
    'remove BC', 'add EA', 'add EF', 'add GQ', 'add AC', 'add DQ',
    'add EG', 'add QH', 'remove CD', 'remove BD', 'remove QG',
    'add FG', 'remove GE')
for case in test_cases:
    command, edge = case.split()
    u, v = edge
    g.add_edge(u, v) if command == 'add' else g.remove_edge(u, v)
    print('{:<10}'.format(case), g.has_cycle())
```

Output:

```
add QH     True
remove FG  True
remove GQ  True
remove HQ  True
remove AE  True
remove CA  True
remove EB  True
remove CE  True
remove DE  True
remove BC  False
add EA     False
add EF     False
add GQ     False
add AC     False
add DQ     False
add EG     True
add QH     True
remove CD  True
remove BD  False
remove QG  False
add FG     True
remove GE  False
```

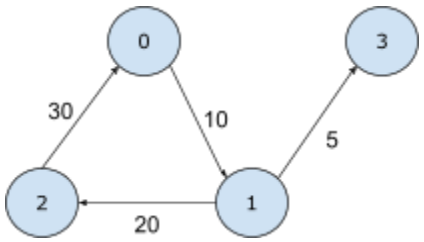
Part 2 - Summary and Specific Instructions

1. Implement the DirectedGraph class by completing the provided skeleton code in the file `d_graph.py`. DirectedGraph class is designed to support the following type of graph: directed, weighted (positive edge weights only), no duplicate edges, no loops. Cycles are allowed.

2. Once completed, your implementation will include the following methods:

```
add_vertex(), add_edge()
remove_edge(), get_vertices(), get_edges()
is_valid_path(), dfs(), bfs()
has_cycle(), dijkstra()
```

3. Directed graphs should be stored as a two dimensional matrix, which is a list of lists in Python. Element on the `i`-th row and `j`-th column in the matrix is the weight of the edge going from the vertex with index `i` to the vertex with index `j`. If there is no edge between those vertices, the value is zero. So for the graph pictured below:



self.adj_matrix = [[0, 10, 0, 0], [0, 0, 20, 5], [30, 0, 0, 0], [0, 0, 0, 0]]

4. The number of vertices in the graph will be between 0 and 900 inclusive. The number of edges will be less than 10,000.

5. RESTRICTIONS: For this assignment, you **ARE** allowed to use any built-in data structures and methods from Python standard library. In addition, you are also allowed to add the following import statements:

```
import heapq
from collections import deque
```

   You are NOT allowed to import ANY other libraries / modules, especially those that provide functionality for working with graphs.

6. Variables in the DirectedGraph class are not private. You ARE allowed to access and change their values directly.

add_vertex(self) -> int:

This method adds a new vertex to the graph. A vertex name does not need to be provided; instead the vertex will be assigned a reference index (integer). The first vertex created in the graph will be assigned index 0, subsequent vertices will have indexes 1, 2, 3 etc. This method returns a single integer - the number of vertices in the graph after the addition.

add_edge(self, src: int, dst: int, weight=1) -> None:

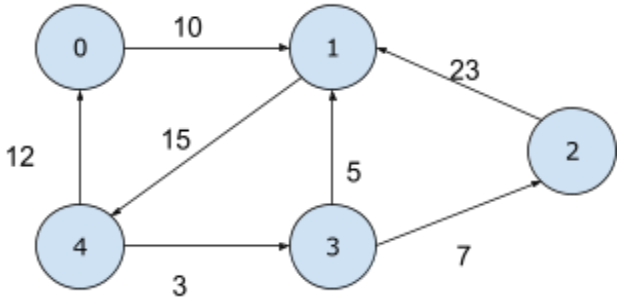
This method adds a new edge to the graph, connecting the two vertices with the provided indices. If either (or both) vertex indices do not exist in the graph, or if the `weight` is not a positive integer, or if `src` and `dst` refer to the same vertex, the method does nothing. If an edge already exists in the graph, the method will update its weight.

Example #1:

```
g = DirectedGraph()
print(g)
for _ in range(5): g.add_vertex()
print(g)
edges = [(0, 1, 10), (4, 0, 12), (1, 4, 15), (4, 3, 3),
         (3, 1, 5), (2, 1, 23), (3, 2, 7)]
for src, dst, weight in edges:
    g.add_edge(src, dst, weight)
print(g)
```

Output:

```
EMPTY GRAPH

GRAPH (5 vertices):
   | 0  1  2  3  4
------------------
 0 | 0  0  0  0  0
 1 | 0  0  0  0  0
 2 | 0  0  0  0  0
 3 | 0  0  0  0  0
 4 | 0  0  0  0  0

GRAPH (5 vertices):
   | 0  1  2  3  4
------------------
 0 | 0 10  0  0  0
 1 | 0  0  0  0 15
 2 | 0 23  0  0  0
 3 | 0  5  7  0  0
 4 |12  0  0  3  0
```



remove_edge(self, u: int, v: int) -> None:

This method removes an edge between the two vertices with provided indices. If either (or both) vertex indices do not exist in the graph, or if there is no edge between them, the method does nothing (no exception needs to be raised).

get_vertices(self) -> []:

This method returns a list of the vertices of the graph. The order of the vertices in the list does not matter.

get_edges(self) -> []:

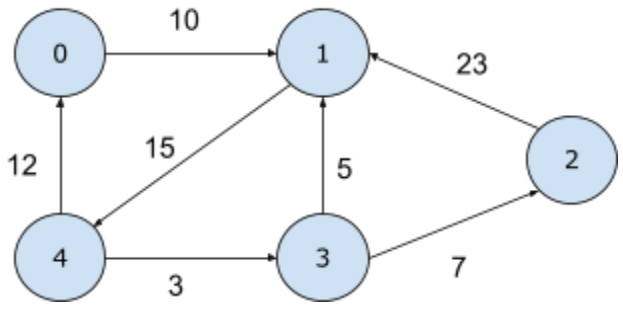
This method returns a list of edges in the graph. Each edge is returned as a tuple of two incident vertex indices and weight. The first element in the tuple refers to the source vertex. The second element in the tuple refers to the destination vertex. The third element in the tuple is the weight of the edge. The order of the edges in the list does not matter.

Example #1:

```
g = DirectedGraph()
print(g.get_edges(), g.get_vertices(), sep='\n')
edges = [(0, 1, 10), (4, 0, 12), (1, 4, 15), (4, 3, 3),
         (3, 1, 5), (2, 1, 23), (3, 2, 7)]
g = DirectedGraph(edges)
print(g.get_edges(), g.get_vertices(), sep='\n')
```

Output:

```
[]
[]
[(0, 1, 10), (1, 4, 15), (2, 1, 23), (3, 1, 5), (3, 2, 7), (4, 0, 12), (4, 3, 3)]
[0, 1, 2, 3, 4]
```



is_valid_path(self, path: []) -> bool:

This method takes a list of vertex indices and returns True if the sequence of vertices represents a valid path in the graph (one can travel from the first vertex in the list to the last vertex in the list, at each step traversing over an edge in the graph). An empty path is considered valid.

Example #1:

```
edges = [(0, 1, 10), (4, 0, 12), (1, 4, 15), (4, 3, 3),
         (3, 1, 5), (2, 1, 23), (3, 2, 7)]
g = DirectedGraph(edges)
test_cases = [[0, 1, 4, 3], [1, 3, 2, 1], [0, 4], [4, 0], [], [2]]
for path in test_cases:
    print(path, g.is_valid_path(path))
```

Output:

```
[0, 1, 4, 3] True
[1, 3, 2, 1] False
[0, 4] False
[4, 0] True
[] True
[2] True
```

dfs(self, v_start: int, v_end=None) -> []:

This method performs a depth-first search (DFS) in the graph and returns a list of vertices visited during the search, in the order they were visited. It takes one required parameter, the index of the vertex from which the search will start, and one optional parameter - the index of the 'end' vertex that will stop the search once that vertex is reached.

If the starting vertex is not in the graph, the method should return an empty list (no exception needs to be raised). If the 'end' vertex is provided but is not in the graph, the search should be done as if there was no end vertex.

When several options are available for picking the next vertex to continue the search, your implementation should pick the vertices by vertex index in ascending order (so, for example, vertex 5 is explored before vertex 6).

bfs(self, v_start: int, v_end=None) -> []:

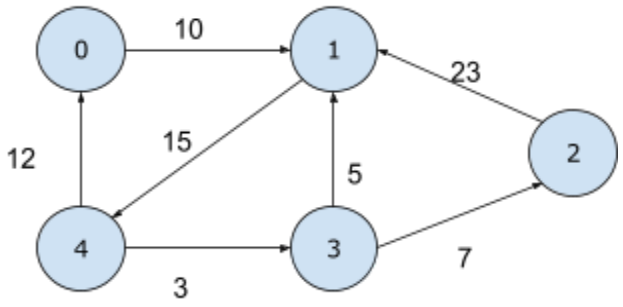
This method works the same as DFS above, except it implements a breadth-first search.

Example #1:

```
edges = [(0, 1, 10), (4, 0, 12), (1, 4, 15), (4, 3, 3),
         (3, 1, 5), (2, 1, 23), (3, 2, 7)]
g = DirectedGraph(edges)
for start in range(5):
    print(f'{start} DFS:{g.dfs(start)} BFS:{g.bfs(start)}')
```

Output:

```
0 DFS:[0, 1, 4, 3, 2] BFS:[0, 1, 4, 3, 2]
1 DFS:[1, 4, 0, 3, 2] BFS:[1, 4, 0, 3, 2]
2 DFS:[2, 1, 4, 0, 3] BFS:[2, 1, 4, 0, 3]
3 DFS:[3, 1, 4, 0, 2] BFS:[3, 1, 2, 4, 0]
4 DFS:[4, 0, 1, 3, 2] BFS:[4, 0, 3, 1, 2]
```



has_cycle(self) -> bool:

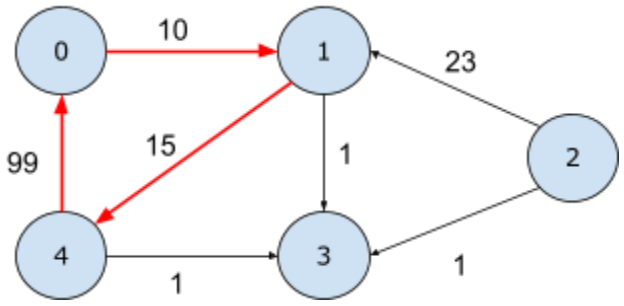
This method returns True if there is at least one cycle in the graph. If the graph is acyclic, the method returns False.

Example #1:

```
edges = [(0, 1, 10), (4, 0, 12), (1, 4, 15), (4, 3, 3),
         (3, 1, 5), (2, 1, 23), (3, 2, 7)]
g = DirectedGraph(edges)
edges_to_remove = [(3, 1), (4, 0), (3, 2)]
for src, dst in edges_to_remove:
    g.remove_edge(src, dst)
    print(g.get_edges(), g.has_cycle(), sep='\n')
edges_to_add = [(4, 3), (2, 3), (1, 3), (4, 0, 99)]
for src, dst, *weight in edges_to_add:
    g.add_edge(src, dst, *weight)
    print(g.get_edges(), g.has_cycle(), sep='\n')
print('\n', g)
```

Output:

```
[(0, 1, 10), (1, 4, 15), (2, 1, 23), (3, 2, 7), (4, 0, 12), (4, 3, 3)]
True
[(0, 1, 10), (1, 4, 15), (2, 1, 23), (3, 2, 7), (4, 3, 3)]
True
[(0, 1, 10), (1, 4, 15), (2, 1, 23), (4, 3, 3)]
False
[(0, 1, 10), (1, 4, 15), (2, 1, 23), (4, 3, 1)]
False
[(0, 1, 10), (1, 4, 15), (2, 1, 23), (2, 3, 1), (4, 3, 1)]
False
[(0, 1, 10), (1, 3, 1), (1, 4, 15), (2, 1, 23), (2, 3, 1), (4, 3, 1)]
False
[(0, 1, 10), (1, 3, 1), (1, 4, 15), (2, 1, 23), (2, 3, 1), (4, 0, 99), (4, 3, 1)]
True

GRAPH (5 vertices):
   | 0  1  2  3  4
------------------
 0 | 0 10  0  0  0
 1 | 0  0  0  1 15
 2 | 0 23  0  1  0
 3 | 0  0  0  0  0
 4 |99  0  0  1  0
```



dijkstra(self, src: int) -> []:

This method implements the Dijkstra algorithm to compute the length of the shortest path from a given vertex to all other vertices in the graph. It returns a list with one value per each vertex in the graph, where the value at index 0 is the length of the shortest path from vertex SRC to vertex 0, the value at index 1 is the length of the shortest path from vertex SRC to vertex 1 etc. If a certain vertex is not reachable from SRC, the returned value should be INFINITY (in Python, use float('inf')). You may assume that SCR will be a valid vertex.

Example #1:

```
edges = [(0, 1, 10), (4, 0, 12), (1, 4, 15), (4, 3, 3),
         (3, 1, 5), (2, 1, 23), (3, 2, 7)]
g = DirectedGraph(edges)
for i in range(5):
    print(f'DIJKSTRA {i} {g.dijkstra(i)}')
g.remove_edge(4, 3)
print('\n', g)
for i in range(5):
    print(f'DIJKSTRA {i} {g.dijkstra(i)}')
```

Output:

```
DIJKSTRA 0 [0, 10, 35, 28, 25]
DIJKSTRA 1 [27, 0, 25, 18, 15]
DIJKSTRA 2 [50, 23, 0, 41, 38]
DIJKSTRA 3 [32, 5, 7, 0, 20]
DIJKSTRA 4 [12, 8, 10, 3, 0]

 GRAPH (5 vertices):
   | 0  1  2  3  4
------------------
 0 | 0 10  0  0  0
 1 | 0  0  0  0 15
 2 | 0 23  0  0  0
 3 | 0  5  7  0  0
 4 |12  0  0  0  0

DIJKSTRA 0 [0, 10, inf, inf, 25]
DIJKSTRA 1 [27, 0, inf, inf, 15]
DIJKSTRA 2 [50, 23, 0, inf, 38]
DIJKSTRA 3 [32, 5, 7, 0, 20]
DIJKSTRA 4 [12, 22, inf, inf, 0]
```