Project 9: Graphs Part 1

Due: Thursday, April 14 @ 10:00 pm

This is not a team project, do not copy someone else's work.

Project based on contributions by Adam Kasumovic, Andrew Haas, Brooke Osterkamp, Andrew McDonald, Tanawan Premsri, and Andy Wilson.

Assignment Overview

Graphs are particularly useful data structures for modeling connections and relationships among objects. In fact, you've likely used an application that relies on graphical modeling today - a few examples include:

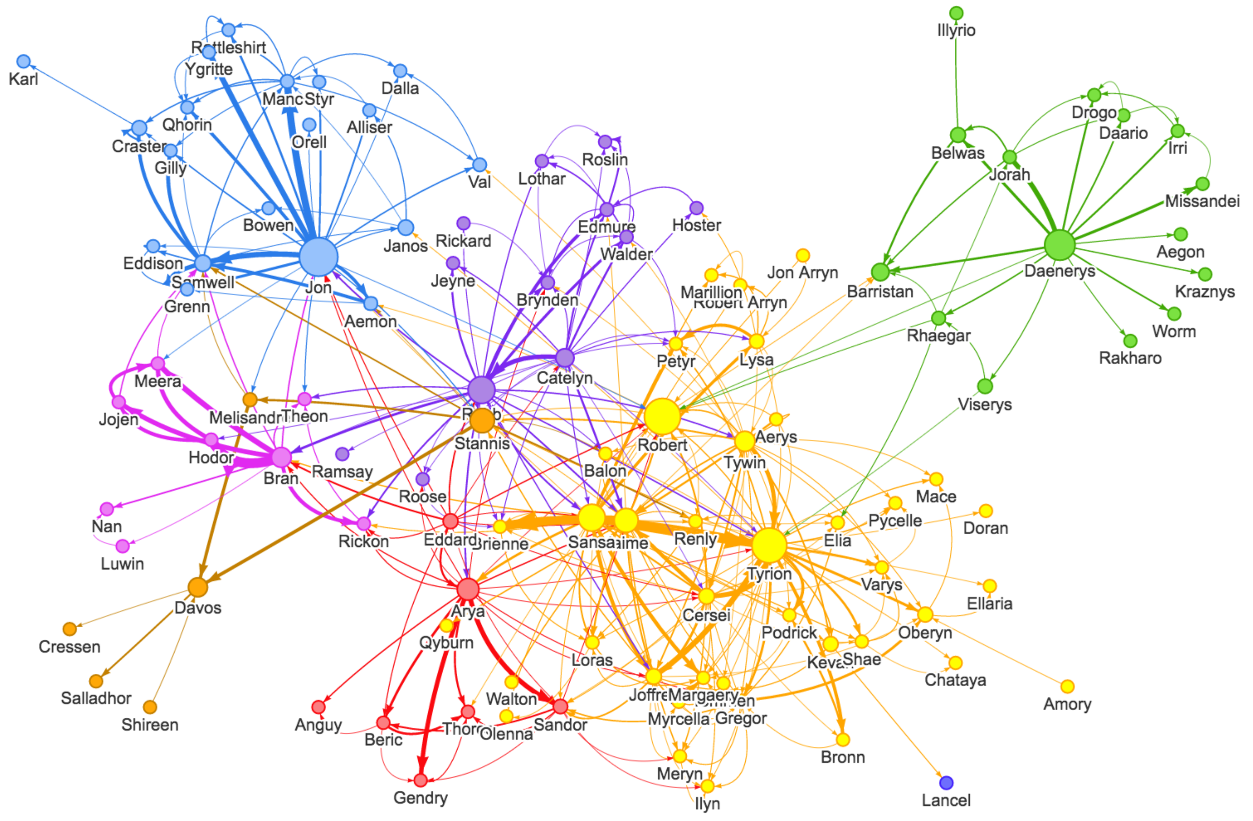
- **Facebook / Twitter / Instagram**
  - Users are modeled as vertices storing posts, photos, videos, etc.
  - Edges are modeled as "friendships," "followers," "likes," "favorites," etc.
- **Google / Apple / Bing Maps**
  - Intersections, cities, and other points of interest are modeled by vertices
  - Road segments between intersections are modeled by weighted edges, where weights represent the relative speed/traffic of the road segment

You will be implementing a **directed, weighted** Graph ADT using the **adjacency map** design, in which a graph object consists of a map (ordered dictionary) of vertices, and each vertex holds its own map (ordered dictionary) of adjacent vertices, i.e. vertices to which that vertex is connected by an outgoing edge.

For more depth information on Graphs make sure to check D2L-Week-10-11-12, lecture slides and Graph ADT source code is all uploaded on D2L.

In some ways, this project (along with Project 10) also serves as a capstone to the course; in completing it one utilizes recursion, queues, two-dimensional arrays, hash maps, dynamic programming, and more. You may also notice that trees, linked lists, and even heaps are special cases of the general graph structure and that many graph algorithms can be applied to these earlier structures without modification. To highlight this inheritance, consider the inorder traversal algorithm we applied to AVL trees; really, it was nothing more than a graph depth-first search with a tendency to go left before right.

The goal of this project (along with Project 10) is to introduce the versatile and flexible nature of graphs, along with the operations and search algorithms that make them so useful.



Assignment Notes

- A plotting function is provided to help you visualize the progression of various search algorithms
  - Be sure to read the specs explaining **plot()**
  - If you don't want to use it, just comment out the related import statements and **plot()** function
- Python allows representation of the value infinity using **float('inf')**
- No negative edge weights will ever be added to the graph
  - All edge weights are numeric values greater than or equal to zero
- Time complexities are specified in terms of *V* and *E*, where *V* represents the number of vertices in the graph and *E* represents the number of edges in a graph
  - Recall that *E* is bounded above by *V*^2; a graph has *E* = *V*^2 edges if and only if every vertex is connected to every other vertex
- Recall that **list.insert(0, element)** and **list.pop(0)** are both *O(N)* calls on a Python list
  - Recall that python's 'lists' are not lists in the more common sense of the word: linked lists. They are dynamically managed tuples, stored in memory as contiguous arrays of pointers to elements elsewhere in memory. This allows indexing into a 'list' in constant time. The downside of this, however, is that adding to a python ' list' at a specific index, *i*, requires shifting the pointer to every element past *i* by one in the underlying array: a linear operation.
  - Be careful when implementing **bfs and dfs** to ensure you do not break time complexity by popping or inserting from the front of a list when reconstructing a path
  - Instead of inserting into / popping from the front of the list, simply append to or pop from the end, then reverse the list *once* at the end
    - If you have N calls to **list.insert(0, element)**, that is *O(N^2)*

    - If you instead have N calls to **list.append(element)**, followed by a single call to **list.reverse()**, that is *O(N)*
    - Both methods will result in the same list being constructed, but the second is far more efficient

Assignment Specifications

class Vertex

Represents a vertex object, the building block of a graph.

DO NOT MODIFY the following attributes/functions

- **Attributes**
  - **id:** A string used to uniquely identify a vertex
  - **adj:** A dictionary of type **{other_id : number}** which represents the connections of a vertex to other vertices; the existence of an entry with key **other_i****d** indicates connection from this vertex to the vertex with id **other_id** by an edge with weight **number**
    - Note that as of Python 3.7, insertion ordering in normal dictionaries is guaranteed and ensures traversals will select the next neighbor to visit deterministically
  - **visited:** A boolean flag used in search algorithms to indicate that the vertex has been visited
  - **x:** The x-position of a vertex (defaults to zero)
  - **y:** The y-position of a vertex (defaults to zero)
- **__init__(self, idx: str, x: float=0, y: float=0) -> None:**
  - Constructs a Vertex object
- **__eq__(self, other: Vertex) -> bool:**
  - Compares this vertex for equality with another vertex
- **__repr__(self) -> str:**
  - Represents the vertex as a string for debugging
- **__str__(self) -> str:**
  - Represents the vertex as a string for debugging
- **__hash__(self) -> int:**
  - Allows the vertex to be hashed into a set; used in unit tests

IMPLEMENT the following functions

- **deg(self) -> int:**
  - Returns the number of outgoing edges from this vertex; i.e., the outgoing degree of this vertex
  - *Time Complexity: O(1)*
  - *Space Complexity: O(1)*
- **get_outgoing_edges(self) -> Set[Tuple[str, float]]:**
  - Returns a **set** of tuples representing outgoing edges from this vertex
  - Edges are represented as tuples **(other_id, weight)** where
    - **other_id** is the unique string id of the destination vertex
    - **weight** is the weight of the edge connecting this vertex to the other vertex
  - Returns an empty set if this vertex has no outgoing edges
  - *Time Complexity: O(degV)*
  - *Space Complexity: O(degV)*

- **euclidean_dist(self, other: Vertex) -> float:**
  - Returns the euclidean distance [based on two-dimensional coordinates] between this vertex and vertex **other**
  - Used in Project 10
  - *Time Complexity: O(1)*
  - *Space Complexity: O(1)*
- **taxicab_dist(self, other: Vertex) -> float:**
  - Returns the taxicab distance [based on two-dimensional coordinates] between this vertex and vertex **other**
  - Used in Project 10
  - *Time Complexity: O(1)*
  - *Space Complexity: O(1)*

class Graph

Represents a graph object

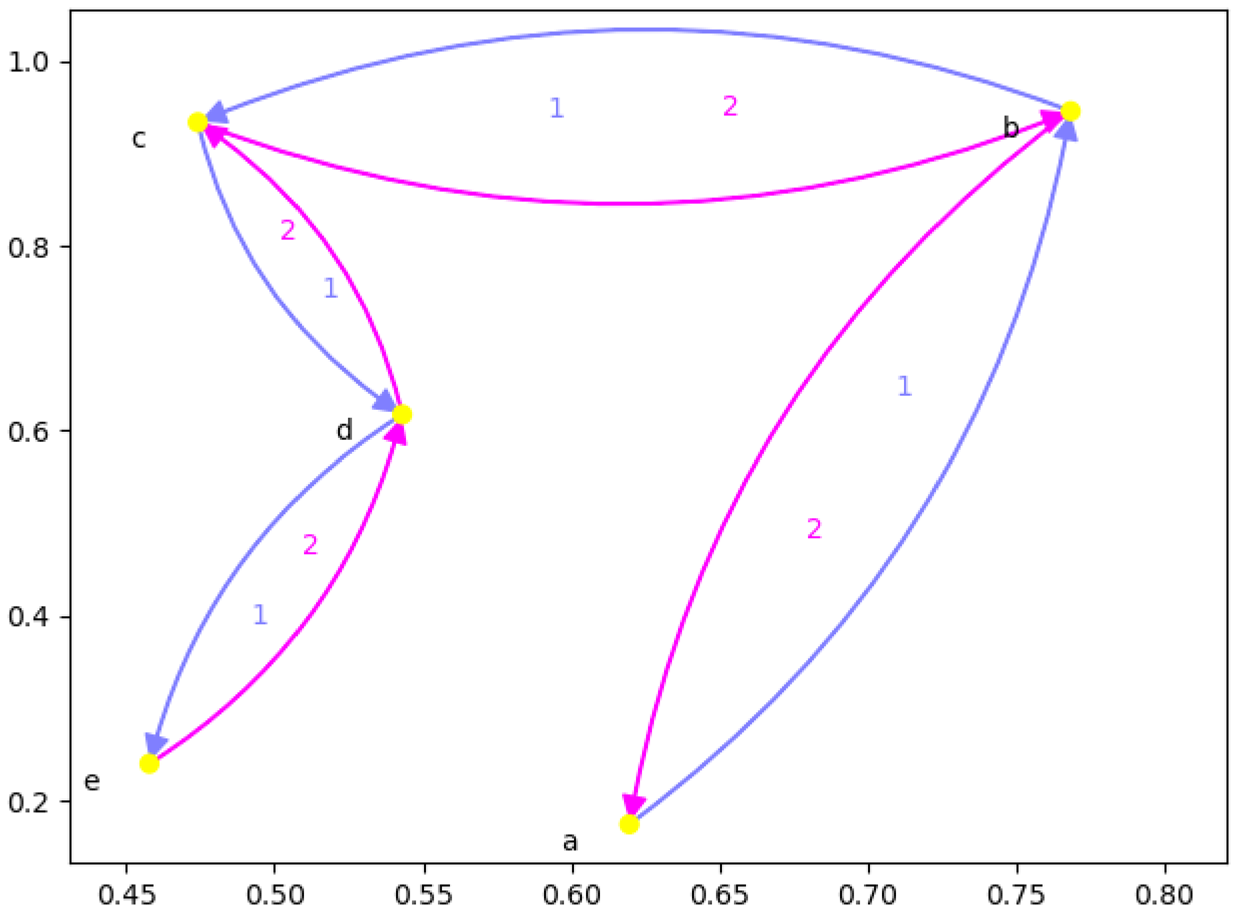
DO NOT MODIFY the following attributes/functions

- **Attributes**
  - **size:** The number of vertices in the graph
  - **vertices:** A dictionary of type **{id : Vertex}** storing the vertices of the graph, where **id** represents the unique string id of a **Vertex** object
    - Note that as of Python 3.7, insertion ordering in normal dictionaries is guaranteed and ensures **get_edges(self)** and **get_vertices(self)** will return deterministically ordered lists
  - **plot_show**: If true, calls to **plot()** display a rendering of the graph in matplotlib; if false, all calls to **plot()** are ignored (see **plot()** below)
  - **plot_delay**: Length of delay in **plot()** (see **plot()** below)
- **__init__(self, plt_show: bool=False) -> None:**
  - Constructs a Graph object
  - Sets **self.plot_show** to False by default
- **__eq__(self, other: Graph) -> bool:**
  - Compares this graph for equality with another graph
- **__repr__(self) -> str:**
  - Represents the graph as a string for debugging
- **__str__(self) -> str:**
  - Represents the graph as a string for debugging
- **add_to_graph(self, start_id: str, dest_id: str=None, weight: float=0) -> float:**
  - Adds a vertex / vertices / edge to the graph
    - Adds a vertex with id **start_id** to the graph if no such vertex exists
    - Adds a vertex with id **dest_id** to the graph if no such vertex exists and **dest_id** is not None
    - Adds an edge with weight **weight** if **dest_id** is not None
  - If a vertex with id **start_id** or **dest_id** already exists in the graph, this function will not overwrite that vertex with a new one
  - If an edge already exists from vertex with id **start_id** to vertex with id **dest_id**, this function will overwrite the weight of that edge
- **matrix2graph(self, matrix: Matrix) -> None:**
  - Constructs a graph from a given adjacency matrix representation

    - **matrix** is guaranteed to be a square 2D list (i.e. list of lists where # rows = # columns), of size **[V+1]** x **[V+1]**
        - **matrix[0][0]** is None
        - The first row and first column of **matrix** hold string ids of vertices to be added to the graph and are symmetric, i.e. **matrix[i][0] = matrix[0][i]** for i = 1, ..., n
        - **matrix[i][j]** is None if no edge exists from the vertex **matrix[i][0]** to the vertex **matrix[0][j]**
        - **matrix[i][j]** is a **number** if an edge exists from the vertex **matrix[i][0]** to the vertex **matrix[0][j]** with weight **number**
- **graph2matrix(self) -> None:**
    - Constructs and returns an adjacency matrix from a graph
    - The output matches the format of matrices described in **matrix2graph**
    - If the graph is empty, returns **None**
- **graph2csv(self, filepath: str) -> None:**
    - Encodes the graph (if non-empty) in a csv file at the given location

USE the following function however you'd like

- **plot(self) -> None:**
    - Renders a visual representation of the graph using matplotlib and displays graphic in PyCharm
        - Follow this tutorial to install matplotlib and numpy if you do not have them , or follow the tooltip auto-suggested by PyCharm
    - Provided for use in debugging
    - If you call this in your searches and **self.****plot_show** is true, the search process will be animated in successive plot renderings (with time between frames controlled by **self.plot_delay**)
    - Not tested in any testcases
        - All testcase graphs are constructed with **self.plot_show** set to False
    - If vertices have (x,y) coordinates specified, they will be plotted at those locations
    - If vertices do not have (x,y) coordinates specified, they will be plotted at a random point on the unit circle
    - To install the necessary packages (matplotlib and numpy), follow the auto-suggestions provided by PyCharm
    - Vertices and edges are labeled; edges are color-coded by weight
        - If a bi-directional edge exists between vertices, two color-coded weights will be displayed



IMPLEMENT the following functions

- **unvisit_vertices(self) -> None:**
  - Resets visited flags to False of all vertices within the graph
  - Used in unit tests to reset graph between tests
  - *Time Complexity: O(V)*
  - *Space Complexity: O(V)*
- **get_vertex_by_id(self, v_id: str) -> Vertex:**
  - Returns the Vertex object with id **v_id** if it exists in the graph
  - Returns None if no vertex with unique id **v_id** exists
  - *Time Complexity: O(1)*
  - *Space Complexity: O(1)*
- **get_all_vertices(self) -> Set[Vertex]:**
  - Returns a **set** of all **Vertex objects** held in the graph
  - Returns an empty set if no vertices are held in the graph
  - *Time Complexity: O(V)*
  - *Space Complexity: O(V)*
- **get_edge_by_ids(self, begin_id: str, end_id: str) -> Tuple[str, str, float]:**
  - Returns the edge connecting the vertex with id **begin_id** to the vertex with id **end_id** in a tuple of the form **(begin_id, end_id, weight)**
  - If the edge or either of the associated vertices does not exist in the graph, returns **None**
  - *Time Complexity: O(1)*
  - *Space Complexity: O(1)*
- **get_all_edges(self) -> Set[Tuple[str, str, float]]:**

  - Returns a **set** of tuples representing all edges within the graph
  - Edges are represented as tuples **(begin_id, end_id, weight)** where
    - **begin_id** is the unique string id of the starting vertex
    - **end_id** is the unique string id of the destination vertex
    - **weight** is the weight of the edge connecting the starting vertex to the destination vertex
  - Returns an empty set if the graph is empty
  - *Time Complexity: O(V+E)*
  - *Space Complexity: O(E)*
- **build_path(self, back_edges: Dict[str, str], begin_id: str, end_id: str) -> Tuple[List[str], float]:**
  - Given a dictionary of back-edges (a mapping of vertex id to predecessor vertex id), reconstructs the path from start_id to end_id and computes the total distance
  - Helper function to **bfs** and **MUST BE CALLED THERE!**
  - Used in Project 10
  - Returns tuple of the form **([path], distance)** where
    - **[path]** is a list of vertex id strings beginning with **begin_id**, terminating with **end_id**, and including the ids of all intermediate vertices connecting the two
    - **distance** is the sum of the weights of the edges along the **[path]** traveled
  - Handle the cases where no path exists from vertex with id **begin_id** to vertex with **end_id** or if one of the vertices does not exist in **bfs** (see below for more info).
  - *Time Complexity: O(V)*
  - *Space Complexity: O(V)*
- **bfs(self, begin_id: str, end_id: str) -> Tuple[List[str], float]:**
  - Perform a breadth-first search beginning at vertex with id **begin_id** and terminating at vertex with id **end_id**
  - **MUST CALL build_path!**
  - **As you explore from each vertex, iterate over neighbors using vertex.adj (not vertex.get_edges()) to ensure neighbors are visited in proper order**
  - Returns tuple of the form **([path], distance)** where
    - **[path]** is a list of vertex id strings beginning with **begin_id**, terminating with **end_id**, and including the ids of all intermediate vertices connecting the two
    - **distance** is the sum of the weights of the edges along the **[path]** traveled
  - If no path exists from vertex with id **begin_id** to vertex with **end_id** or if one of the vertices does not exist, returns tuple **([],0)**
  - Guaranteed that **begin_id != end_id** (since that would be a trivial path)
  - Because our adjacency maps use insertion ordering, neighbors will be visited in a deterministic order, and thus you do not need to worry about the order in which you visit neighbor vertices at the same depth
  - Use the SimpleQueue class to guarantee O(1) pushes and pops on the queue
  - *Time Complexity: O(V+E)*
  - *Space Complexity: O(V)*
- **dfs(self, begin_id: str, end_id: str) -> Tuple[List[str], float]:**
  - Wrapper function for **dfs_inner**, which MUST BE CALLED within this function
    - The majority of the work of dfs should be done in **dfs_inner**
    - This function makes it simpler for client code to call for a dfs
    - This function makes it possible to avoid inserting vertex ids at the front of the path list on path reconstruction, which leads to sub-optimal performance (see Assignment Notes)

        - Hint: construct the path in reverse order in **dfs_inner**, then reverse the path in this function to optimize time complexity
    - Hint: call **dfs_inner** with **current_id** as **begin_id,** then reverse the path here and return it
    - *Time Complexity: O(V+E) (including calls to dfs_inner)*
    - *Space Complexity: O(V) (including calls to dfs_inner)*
- **dfs_inner(self, current_id: str, end_id: str, path: List[str]=[]) -> Tuple[List[str], float]**
    - Performs the recursive work of depth-first search by searching for a path from vertex with id **current_id** to vertex with id **end_id**
    - **MUST BE RECURSIVE**
    - **As you explore from each vertex, iterate over neighbors using vertex.adj (not vertex.get_edges()) to ensure neighbors are visited in proper order**
    - Returns tuple of the form **([path], distance)** where
        - **[path]** is a list of vertex id strings beginning with **begin_id**, terminating with **end_id**, and including the ids of all intermediate vertices connecting the two
        - **distance** is the sum of the weights of the edges along the **[path]** traveled
    - If no path exists from vertex with id **current_id** to vertex with **begin_id** or if one of the vertices does not exist, returns tuple **([],0)**
    - Guaranteed that **begin_id != end_id** (since that would be a trivial path)
    - Because our adjacency maps use insertion ordering, neighbors will be visited in a deterministic order, and thus you do not need to worry about the order in which you visit neighbor vertices
    - *Time Complexity: O(V+E)*
    - *Space Complexity: O(V)*
- **topological_sort(self) -> List[str]:**
    - Performs topological sort on the graph, returning a possible topological ordering as a list of vertex ids.
    - Recall that there can be multiple correct orderings following topological sort, and the testcases will allow for any correct ordering.
    - Guaranteed that when this function is tested the graph is a connected DAG (Directed Acyclic Graph), although you may find it useful to ensure that this function properly "attempts" to sort cyclic connected directed graphs (i.e. this function does not crash when given a cyclic graph and just returns an invalid topological ordering).
    - You are provided with an **optional** empty **topological_sort_inner** function should you choose to use it in a recursive solution, although it is not required. You may remove it if you do not use it, and you may also modify its function signature.
    - Hint: There is a solution that is quite similar to your recursive implementation of **dfs** above which uses **topological_sort_inner** to traverse the graph in the same fashion as **dfs_inner,** except this time you do not stop on an end_id.
    - *Time Complexity: O(V+E) (including calls to topological_sort_inner)*
    - *Space Complexity: O(V) (including calls to topological_sort_inner)*

Application Problem

BRAVE TARNISHED, RISE, AND BECOME ELDEN LORD!

We salute you, worthy Tarnished for persevering through hardships to reach this point, but now you must embark on (Part 1) of your final journey.

Elden Ring is a game known for its high levels of difficulty, and like other games in its own special genre, much of this difficulty comes from fighting challenging bosses. Unfortunately, this extreme difficulty is off-putting for a lot of players. As a new recruit to FromSoftware 's team of game programmers, you've been tasked with rebalancing the game. Miyazaki has made it very clear that no changes to the actual bosses will be made. Instead, your teammates have proposed possible boss reorderings in the form of a connected, directed graph.

In such graphs, the vertices represent bosses, while the edges represent pairwise orderings in which the two bosses must be defeated. For example, an edge ***from*** Margit ***to*** Godrick i.e. ('Margit', 'Godrick') means that Margit must be defeated ***before*** Godrick. In other words, Godrick will not be available until Margit has been defeated. Another way to think of it is that before visiting a vertex (defeating a boss), one must visit (defeat) *all* vertices (bosses) with edges incoming to said vertex. Examples are provided below. We define the game with its boss ordering graph as _ beatable_ if and only if it is possible to defeat all bosses given by the graph while following the ordering rule denoted by the edges in the graph.

Your goal is to write a function for your Graph ADT that tells us if the boss ordering given by the graph produces a beatable game. Note that boss orderings may be partial (not include all bosses) and that there may be more than one order to defeat bosses in order to beat the game.

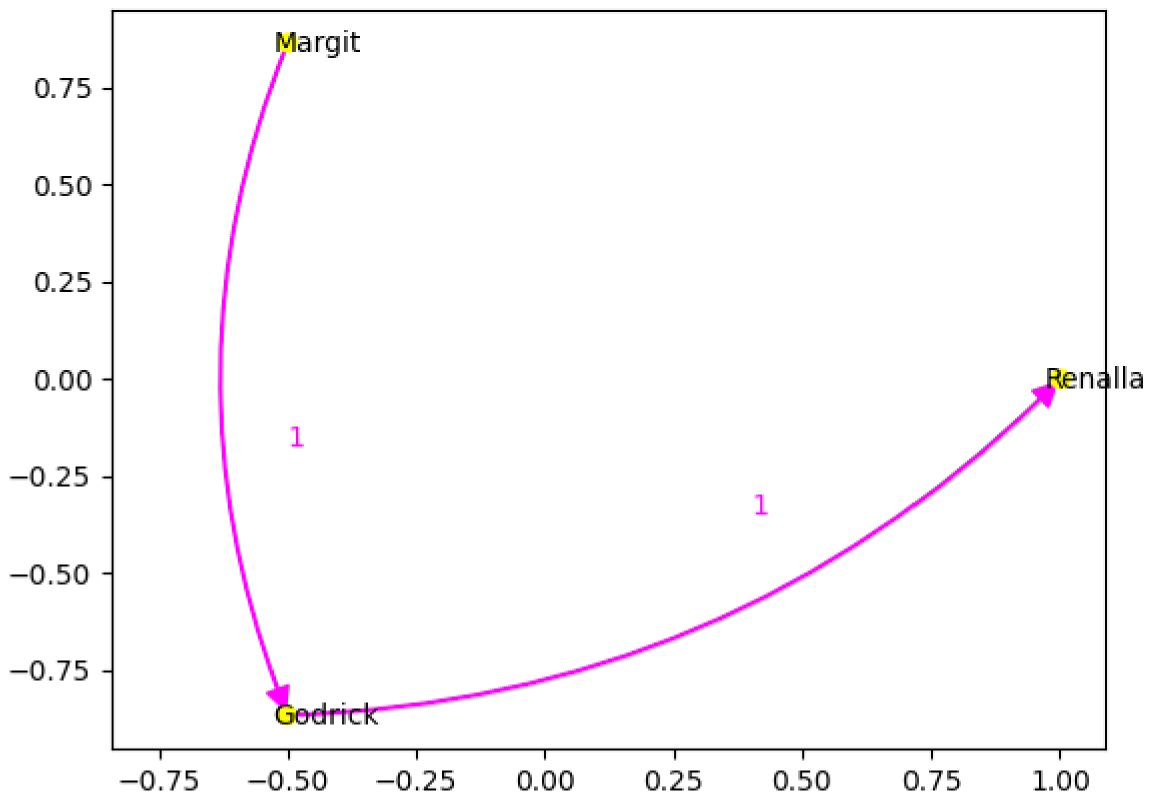
Your Task

Implement the application solution function on your Graph ADT according to the following specifications

- **boss_order_validity(self) -> bool:**
  - Returns **True** if the game be beaten with the boss ordering represented by this graph, otherwise returns **False**
  - Be careful with time and space complexity, they are easy to violate here.
  - The testcases should give a good idea of when the game can be beaten or not.
  - Guaranteed that the graph is connected.
  - Consider an empty graph (no bosses) as already trivially beatable (return **True**).
  - Hint: You may find it useful to use your **topological_sort** function to solve this problem, but it is not required.
  - *Time Complexity: O(V+E)*
  - *Space Complexity: O(V)*

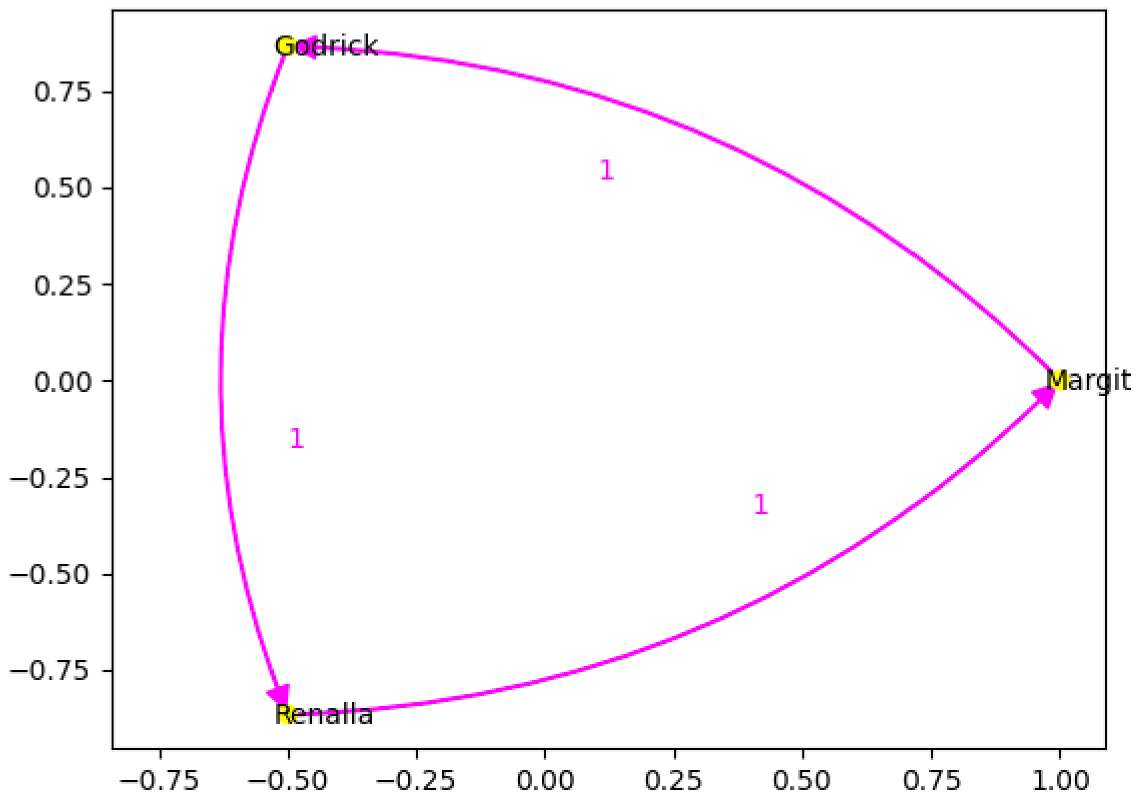
Examples

Given the following graph (all plots made using the plot() function we provide you),



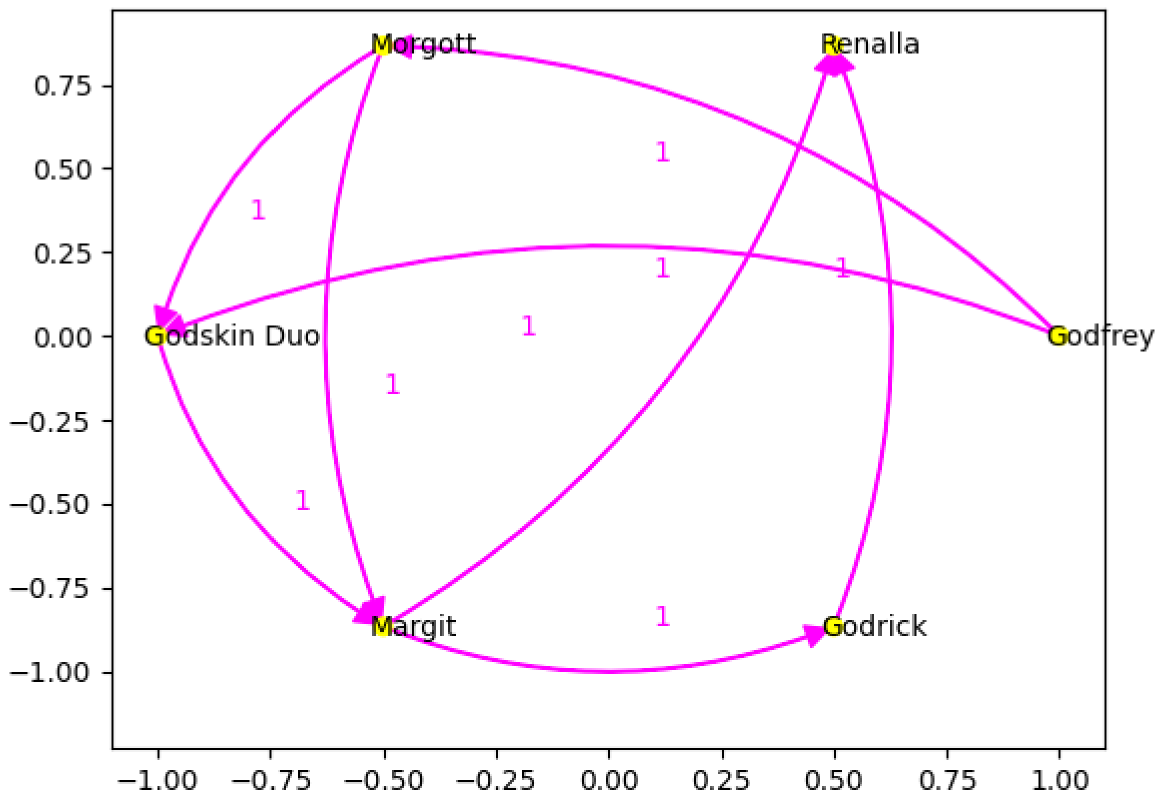
- **boss_order_validity()** would return **True**
  - Players can defeat Margit, then Godrick, and then Renalla.

Given the following graph,



- **boss_order_validity()** would return **False**
  - In order to defeat Renalla, Godrick must be defeated. In order to defeat Godrick, Margit must be defeated. However, in order to defeat Margit, Renalla must be defeated. It is impossible to beat the game!

Given the following graph,



- **boss_order_validity()** would return **True**
  - The bosses can be defeated in the following order: Godfrey, Morgott, Godskin Duo, Margit, Godrick, Renalla. Like the first case, there is only one order to defeat the bosses. (There may be multiple).

Submission

Deliverables

Be sure to upload the following deliverables in a .zip folder to Mimir by 10:00 pm ET, on Thursday, April 14.

Project09.zip

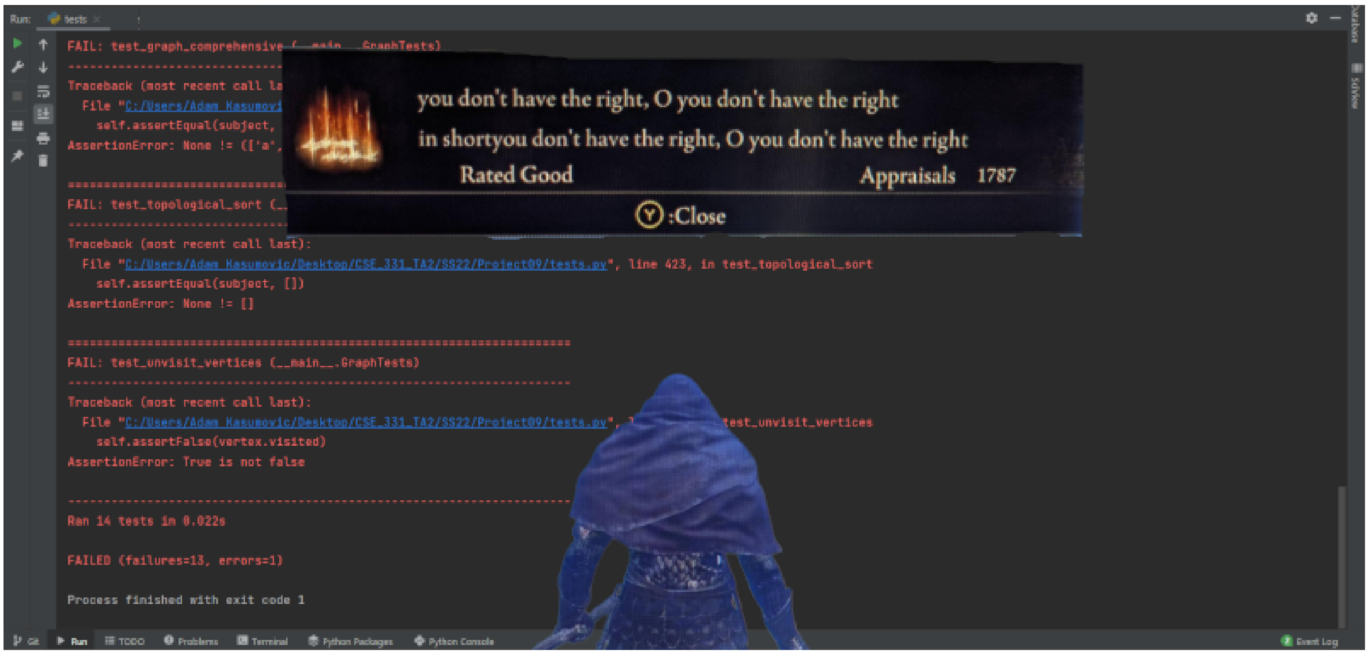
```
    |— Project09/
        |— solution.py        (contains your solution code)
        |— feedback.xml       (for project feedback)
        |— __init__.py        (for proper Mimir testcase loading)
```

Grading

- Tests (70)
  - Coding Standard: __/3
  - feedback.xml Validity Check: __/3
  - Vertex: __/4

        - deg: __/1
        - get_outgoing_edges: __/1
        - distances: __/2
    - Graph: __/45
        - unvisit_vertices: __/1
        - get_vertex_by_id: __/2
        - get_all_vertices: __/2
        - get_edge_by_ids: __/2
        - get_all_edges: __/2
        - build_path: __/4
        - bfs: __/8
        - dfs: __/8
        - topological_sort: __/8
        - comprehensive: __/8
    - Application: __/15
        - boss_order_validity: __/15
- Manual (30)
    - Loss of 1 point per missing docstring (max 3 point loss)
    - Loss of 2 points per changed function signature (max 20 point loss)
    - Time and space complexity points are **all-or-nothing** for each function. If you fail to meet time **or** space complexity in a given function, you do not receive manual points for that function.
    - Not using recursion when we tell you to (dfs_inner) will result in a loss of half of all points associated with that function!
    - Not using build_path will result in a loss of half of all points associated with bfs!
    - Time & Space Complexity (30)
        - M1 - deg, get_outgoing_edges, distances __/3
        - M2 - unvisit_vertices, get_vertex_by_id, get_all_vertices, get_edge_by_ids, get_all_edges __/5
        - M3 - build_path __/2
        - M4 - bfs __/3
        - M5 - dfs __/5
        - M6 - topological_sort __/6
        - M7 - boss_order_validity __/6
    - You must pass a given set of functions' automated tests to be eligible for manual points associated with that set of functions.
        - For example, to be eligible for complexity points on M2, you must pass all automated tests associated with get_vertex_by_id, get_all_vertices, get_edge_by_ids, get_all_edges
    - Any use of networkx or similar prewritten graph libraries in a function will result in a zero for all automated and manual points associated with that function.

Extra



```
FAIL: test_graph_comprehensive (__main__.GraphTests)
----------------------------------------------------------------------
Traceback (most recent call la
  File "C:/Users/Adam_Kasumovi
    self.assertEqual(subject,
AssertionError: None != (['a',

======================================================================
FAIL: test_topological_sort (__
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:/Users/Adam_Kasumovic/Desktop/CSE_331_TA2/SS22/Project09/tests.py", line 423, in test_topological_sort
    self.assertEqual(subject, [])
AssertionError: None != []

======================================================================
FAIL: test_unvisit_vertices (__main__.GraphTests)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:/Users/Adam_Kasumovic/Desktop/CSE_331_TA2/SS22/Project09/tests.py",          test_unvisit_vertices
    self.assertFalse(vertex.visited)
AssertionError: True is not false

----------------------------------------------------------------------
Ran 14 tests in 0.022s

FAILED (failures=13, errors=1)

Process finished with exit code 1
```

you don't have the right, O you don't have the right

in shortyou don't have the right, O you don't have the right

Rated Good Appraisals 1787

You can do it! *Ahem*, I mean,

something incredible ahead

therefore don't give up!

Some music to listen to while working on this project 😃