Assignment 4: Quokka Survival Strategies

All submitted work must be *done individually* without consulting someone else's solutions in accordance with the University's Academic Dishonesty and Plagiarism policies.

Story

A small quokka colony in Western Australia is migrating and since these creatures are vulnerable, we've been tasked with helping them make their way to their new home. For safety reasons, the quokka colony travels together at all times and can only travel between specific locations. We model these locations as the vertices of an undirected graph and two vertices are connected by an edge if the colony can travel from one location to the other.

Since the quokkas need to eat every so often, every vertex has the additional information whether sufficient food is available at a certain location. The colony can move through a number of locations before needing food and we assume that both their starting location and intended destination have food. All other locations may or may not have food.

Please help the quokkas find a path from their current home to their destination such that they have sufficient food along the way.

Informally, our implementation should support the following operations:

`find_path(s, t, k)`

The main operation consists of finding a path from vertex `s` to vertex `t` such that from any location with food along this path we reach the next location with food in at most `k` steps.

Implement `find_path(s, t, k)` returning the list of vertices used in the path, or `None` if no path exists.

`fix_edge(u, v)`
`block_edge(u, v)`

We may need to update the graph sometimes when a certain edge is blocked or becomes accessible again. Implement the `block_edge(u, v)` and `fix_edge(u, v)` functions that removes an existing edge (if exists), or adds a new edge to the graph (if exists).

`exists_path_with_extra_food(s, t, k, x)`

We also want to help the quokkas by placing some extra food ourselves. Implement a method `exists_path_with_extra_food(s, t, k, x)` that returns whether it is possible for the quokkas to make it from s to t along a path where from any location with food we reach the next location with food in at most k steps, by placing food at at most x new locations. In other words, is it possible

to place food at x new locations such that afterwards `find_path(s, t, k)` can find a path?

`find_location_of_extra_food(s, t, k, x)`

Knowing whether we can place food at at most `x` locations such that the quokkas can make it is fine and all, but this doesn't actually help the quokkas yet. Implement a method `find_locations_of_extra_food(s, t, k, x)` that returns the locations where we need to place the extra food such that such that afterwards `find_path(s, t, k)` can find a path. If no such locations can be found you should return `None`.

`minimize_extra_food(s, t, k)`

In order to influence nature as little as possible, we want to place food at as few locations as possible. Implement a method `minimize_extra_food(s, t, k)` that returns the smallest `x` such that placing food at `x` new locations ensures that `find_path(s, t, k)` can find a path. In addition, you should return a list of these `x` locations.

Unmarked challenge: The correct value of `x` can be determined in `O(log x)` Values. Can you figure out how to do this?

Note

This assignment was inspired by Hamster Obstacle Mazes and Quokkas. No quokkas or hamsters were harmed in the production of this assignment. Credit to Andre for the assignment creation.

Visual Example

Given a graph:

```
        *       *
A---B---C---D---E
```

The `find_path(A, E, 2)` is a valid, achievable path, because between `A` and `C` there are 2 hops, and `C` and `E` are two hops, so the Quokkas will survive. This will then return: `[A, B, C, D, E]`.

Whereas `find_path(A, E, 1)` cannot be reached, because `A` to `B` is 1 hop, and the Quokkas will need food. So this returns `None`.

`exists_path_with_extra_food(A, E, 1, 3)` returns `True`, because adding 3 extra food to (at least) `B`, and `D` will make the path `[A, B, C, D, E]` achievable as there is food at least 1 hop away for each vertex.

ORDER MATTERS IN THE PATH RETURNS, IT SHOULD BE A SEQUENCE.

About the code

You are asked to implement 2 major files, `vertex.py` and `graph.py`.

vertex.py

The `Vertex` class provides the information of the vertex in the graph.

Properties

- `has_food` - [Boolean] indicates whether the vertex has food or not.
- `edges` - [List[Vertex]] the list of vertices connected to this vertex, forming edges in the graph.

Functions

`add_edge(v)`

Adds an edge between this vertex and the Vertex `v`.

`rm_edge(v)`

Removes the edge between this vertex and the Vertex `v`.

graph.py

This `QuokkaMaze` class provides the implementation of the graph for the Quokkas to traverse.

Properties

- `vertices` - [List[Vertex]] the list of vertices in the graph.

Functions

`add_vertex(v) -> bool`

Adds the Vertex `v` to the graph, returning `True` if the operation was successful, `False` if it was not, or it was invalid.

`fix_edge(u, v) -> bool`

Fixes an edge between two vertices, `u` and `v`. If an edge already exists, or the edge is invalid, then this operation should return `False`. Else, if the operation succeeds, return `True`.

Example: If an edge between `u` and `v` already exists or is invalid, `fix_edge(u, v)` should return False.

`block_edge(u, v) -> bool`

Blocks the edge between two vertices, u and v. Removes the edge if it exists, and returns `True` if the operation was successful. If the edge does not exist or is invalid, it should be unsuccessful an return `False`.

`find_path(s, t, k) -> List[Vertex] or None`

Find a **SIMPLE PATH** between Vertex `s` and Vertex `t` such that from any location with food along this path we reach the next location with food in at most `k` steps.

This function returns: The list of vertices to form the simple path from `s` to `t` which satisfies the condition, or, `None` if there is no path that exists in the graph.

If there are invalid aspects (invalid path, invalid input), then this function returns `None`.

`exists_path_with_extra_food(s, t, k, x) -> bool`

Determines whether it is possible for the quokkas to make it from Vertex `s` to Vertex `t` along a **SIMPLE path** where from any location with food we reach the next location with food in at most `k` steps, by placing food at at most `x` new locations.

This function returns `True` if we can complete the simple path with at most `x` additional food, else it returns `False`.

If there are invalid aspects (invalid path, invalid input), then this function returns `False`.

`find_location_of_extra_food(s, t, k, x) -> List[Vertex] or None`

Returns the (at most) `x` locations where we can add food, such that afterwards there exists a path from Vertex `s` to Vertex `t` such that from any location with food, we reach the next location with food in at most `k` steps.

This function returns a List of Vertices denoting the location to add the food (in any order). Or, `None` if no path can exist with `x` added food.

`minimize_extra_food(s, t, k) -> List[Vertex] or None`

Returns the smallest `x` locations such that adding food at these `x` locations guarantees that there exists a path from Vertex `s` to Vertex `t` such that from any location with food, we reach the next location with food in at most `k` steps.

This function returns a List of Vertices denoting the location to add the food (in any order). Or, `None` if no path can exist.

IMPORTANT INFORMATION

- We will be performing minor adversarial testing, which means:
  - CHECK YOUR PARAMS, for example:
    - `k` should always be $>= 0$ for `find_path()` and `exists_path_with_extra_food()`.
    - `x` should always be $>= 0$ for `find_path()` and `exists_path_with_extra_food()`.
  - You must be careful for **ALL** functions.

- DO NOT MODIFY THE `has_food` PROPERTY OF THE VERTICES - we are running multiple tests on the same graph, and if you corrupt your graph by modifying the vertices then you may achieve an unwanted result and fail the tests.
- BE CAREFUL WITH `copy` and `deepcopy` - we are using equality on the vertices to check paths, this does a direct memory comparison with the node object, so any modifications and copies will be incorrect.
- List of Vertices as edges in a vertex are **unordered**, but when we fix an edge between two vertices, it should update both!
- The list of vertices returned by path functions (such as `find_path`) **IS ORDERED**, which means you return the sequence to form the path.