CAB203 Project: Bork (Hardcore)

Harrison Leach: n11039639

1 Introduction

The text adventure game, *Bork* has an incredibly convoluted multi-dimensional map that pushes the bounds of space-time. It is of great interest to be able to record all locations of this map, hence the purpose of this problem. The character (hereafter reffered to as Bork) that the player controls, has many actions during the exploration of the map. In the Python interface, these are represented with the following methods:

- bork.restart() - restarts the game, returning Bork to the starting location
- bork.description() - returns a string of Bork's current location
- bork.exits() - returns a Python set of all possible exits from the current location (given as directions)
- bork.move(exit) - moves Bork to the given exit (direction)
- bork.save() - saves the current state of the game
- bork.restore(savegame) - restores a saved game

As well as these functionalities, there are promises and restrictions to this problem:

- It is always possible to return to the starting location from any point of the map
- It is impossible for Bork to use the previous path it just took
- The map does not change throughout the duration of the game

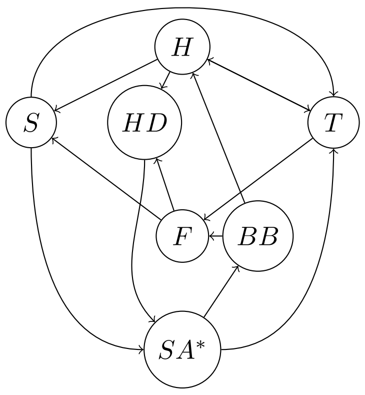
The intention of the solutions program is to output a map that correctly records all locations, and paths on the map. The format of this map was chosen to be a dictionary:

```
exitsMap = {
    'Home': { 'East': 'Swamp', 'West': 'Town' },
    'Town': { 'South': 'Home', 'North': 'Forest' },
    ...
    'Heat death of the universe': { 'Past': 'Sagitarius A*' }
}
```

The remainder of this report will describe the problem, solution, and implementation of the *Bork* problem.

2 Problem

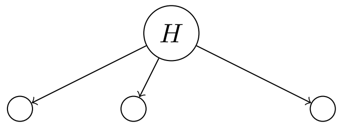
The python interface of *Bork*, provides the map of the game. Given the player cannot take the previous path just taken, this map can be interpreted as a directed graph. It can be visually represented like so:



NOTE: Names of locations have been shortened for simplicity, e.g. H = Home.

At first glance, this seems quiet complex to understand, however, with the use of mathematics, the problem can be solved. This graph can be defined as $G = (V, E)$. With the vertices, V, as the set of all locations, and the edges, E, represented by the set of all connecting paths.

Bork's functionalities brought upon many limitations and obstacles for this problem. Bork is only aware of the name of the current location it is situated in. It only knowns the direction of the exits from said location, not even the names of the destinations. On top of this, this problem is in hardcore mode, depriving the player of the ability to .save() and .restore(savegame). Here is Bork's perspective from the location, *Home*:



This issue can be overcome if the player notes where they've been. This allows Bork to check locations that have been visited but not fully explored by referring to the *map in progress*.

With the goal of traversing the entire map in mind, the question remains: How to traverse the map when the path just taken cannot be the succeeding path?
In graph theory, there's 2 common ways of traversal, Breadth-First-Search (BFS) & Depth-First-Search (DFS). Although very effective in most cases, neither BFS or DFS will work in it's classic form. Whilst, there would likely be a solution using a DFS inspired method, a hybrid version of BFS will be implemented to solve the map.

3 Solution

The approach to this solution is heavily inspired by BFS - to explore all exits from a location before moving to the next location. The map that Bork uses to record locations begins as an empty set, $\emptyset$. When the game starts, Bork immediately finds itself at Home, this is great as Home has 3 possible options to explore. At this point Bork would update the map by appending Home to it. The map now looking like so:

$$\{Home : \{\}\}$$

This is a dictionary, where the location is the key, and the values (that are currently empty) will become the directions and corresponding destinations. Bork takes one of the paths and arrives at a new location, noting the direction it took and the name of where its arrived. For example, say Bork travelled East to the Swamp. Having found a new location, Swamp and a path from home, the map is updated to:

$$\{Home : \{East : Swamp\}, Swamp : \{\}\}$$

The temptation would now be to continue exploring from the Swamp, however, that would be the process a DFS traversal would take. In this solution Bork, must use the restart function to return home. With the knowledge that Bork has already travelled East, it chooses West or Future as it's next direction and then repeats the steps above for the remaining locations. At this point, Home has been completely explored and the map has been updated to look like:

$$\{Home : \{East : Swamp, West : Town, Future : Heat\ Death\ of\ the\ Universe\},$$
$$Swamp : \{\}, Town : \{\}, Heat\ Death\ of\ the\ Universe : \{\}\}$$

Bork will now have a look at what places haven't been fully explored on the map and choose an arbitrary location. For example, say Swamp has been chosen as the location to fully explore, Bork will restart back to home and need to use the map to get to the Swamp and begin exploring. Reiterating the same process that was done above however this time after restarting Bork must use the map to navigate to the Swamp everytime.

Where the resemblance to BFS begins to dissipate is because distance is not considered when selecting the next location. Imagine Bork finished exploring the Swamp and in the process found Sagitarius A*. Sagitarius A*, has been added to the map as a location to explore and very well could be explored next, even though Town and Heat Death of the Universe haven't yet. Distance isn't considered in this solution, Bork only cares about what hasn't been fully explored.

When traversing the map, it is inevitable that a location previously visited will be arrived upon via another path. It is important to check if this location has been fully explored before appending the location to a map with an empty set of directions, as this would overwrite the discoveries made previously for that area.

Finally, the map has been completely traversed when every location on Bork's map has been discovered and there are no locations with empty sets.

4 Implementation

The main function is `traverseBork` (or `traverseBorkHardCore`), it simply does the following:

- Creates a variable, `map` that is initially an empty set, $\emptyset$
- Uses bork.restart() to confirm starting at Home
- Creates a variable `init_location`, that is assigned bork.description()
- Updates the map for the first time by adding Home as a location

Then finally, the function `exploreR(bork, init_location, map)` is called which will be explained shortly.

The implementation of this solution uses six helper functions: `allowed_exits`, `travel_paths`, `travel`, `next_to_explore`, `explore` & `exploreR`.

The function `allowed_exits(bork, map, location)`, simply returns a set of exits that haven't been visited yet for a given location. By checking the exits already visited by using the index, `set(map[location].keys())`, the remaining exits can be returned.

`travel_path(start, end, map)`, creates a set of vertices and edges, using the map in its current position:

```
V = set(map.keys())
```

```
E = { (u, Nu[v]) for u, Nu in map.items() for v in Nu }
```

Then with assistance from the `findPath` function from `digraphs.py`, a path can be obtained to the location that is going to be explored. With the use of for loops, an array of directions is returned to arrive at the desired area.

`travel(bork, given_path)` uses the array given from `travel_path` and uses the bork.move(exit) function repeatedly to go to the destination.

`next_to_explore(map)`, is a function that checks what places have an empty set, $\emptyset$, as their directions. This makes them candidates for the next place to be chosen in accordance to Section 3.

`explore(bork, start, end, map)`, performs the bulk of the exploration process. Bork travels to the desired location using functions from earlier, `travel(bork, travel_path(start, end, map))`. An exit is chosen using the `arbitraryElement` function from `digraphs.py` and `allowed_exits`:

```
exit = digraphs.arbitraryElement(allowed_exits(bork, map, location))
```

It then moves to this exit and assigns the name of the location to a variable called destination:

```
destination = bork.description()
```

It then updates the map to include the direction and where this direction led to:

```
map[location].update({ exit: destination })
```

Then it checks to see if this destination is a new location the map hasn't seen. If so add it as a key to the dictionary with an empty set, otherwise do nothing.

The function repeats this algorithm for the exact number of exits the location has.

`exploreR(bork, init_location, map)`, is a recursive function. To check if the map has been explored it uses `next_to_explore(map)`. If the result is false, it runs `explore(bork, start, end, map)` and checks again by calling itself. Otherwise, the function ends and the completed map is returned.