Kevin Bacon 6 Degrees of Separation

Topic 1 Coding Assignment

Algorithms Review

Recall that searches begin by visiting the root node of the search tree, given by the initial state. Among other book-keeping details, three major things happen in sequence in order to visit a node:

1. First, we remove a node from the frontier set.
2. Second, we check the state against the goal state to determine if a solution has been found.
3. Finally, if the result of the check is negative, we then expand the node. To expand a given node, we generate successor nodes adjacent to the current node, and add them to the frontier set. Note that if these successor nodes are already in the frontier, or have already been visited, then they should not be added to the frontier again.

This describes the life-cycle of a visit, and is the basic order of operations for search agents in this assignment—(1) remove, (2) check, and (3) expand.

Please refer to lectures and Book for further details, and review the pseudocodes before you begin the assignments.

Description

Write a program that determines how many "degrees of separation" apart two actors are.

Introduction

According to the Six Degrees of Kevin Bacon game, anyone in the Hollywood film industry can be connected to Kevin Bacon within six steps, where each step consists of finding a film that two actors both starred in.

In this problem, we're interested in finding the shortest path between any two actors by choosing a sequence of movies that connects them. For example, the shortest path between Jennifer Lawrence and Tom Hanks is 2: Jennifer Lawrence is connected to Kevin Bacon by both starring in "X-Men: First Class," and Kevin Bacon is connected to Tom Hanks by both starring in "Apollo 13."

We can frame this as a search problem: our states are people. Our actions are movies, which take us from one actor to another (it's true that a movie could take us to multiple different actors, but that's okay for this problem). Our initial state and goal state are defined by the two people we're trying to connect. By using **breadth-first search**, we can find the shortest path from one actor to another.

Understanding The Environment

The distribution code contains two sets of CSV data files: one set in the large directory and one set in the small directory. Each contains files with the same names, and the same structure, but small is a much smaller dataset for ease of testing and experimentation.

Each dataset consists of three CSV files. A CSV file, if unfamiliar, is just a way of organizing data in a text-based format: each row corresponds to one data entry, with commas in the row separating the values for that entry.

Open up **small/people.csv**. You'll see that each person has a unique id, corresponding with their id in IMDb's database. They also have a name, and a birth year.

Next, open up **small/movies.csv**. You'll see here that each movie also has a unique id, in addition to a title and the year in which the movie was released.

Now, open up **small/stars.csv**. This file establishes a relationship between the people in people.csv and the movies in movies.csv. Each row is a pair of a **person_id** value and **movie_id** value. The first row (ignoring the header), for example, states that the person with id 102 starred in the movie with id 104257. Checking that against **people.csv** and **movies.csv**, you'll find that this line is saying that Kevin Bacon starred in the movie "A Few Good Men."

Next, take a look at **degrees.py**. At the top, several data structures are defined to store information from the CSV files. The "names" dictionary is a way to look up a person by their name: it maps names to a set of corresponding ids (because it's possible that multiple actors have the same name). The people dictionary maps each person's id to another dictionary with values for the person's name, birth year, and the set of all the movies they have starred in. And the movies dictionary maps each movie's id to another dictionary with values for that movie's title, release year, and the set of all the movie's stars. The **load_data** function loads data from the CSV files into these data structures.

The main function in this program first loads data into memory (the directory from which the data is loaded can be specified by a command-line argument). Then, the function prompts the user to type in two names (feel free to manually hardcode the path also to save time). The **person_id_for_name** function retrieves the id for any person (and handles prompting the user to clarify, in the event that multiple people have the same name). The function then calls the **shortest_path** function to compute the shortest path between the two people, and prints out the path.

The **shortest_path** function, however, is left unimplemented. That's where you come in!

Task Specification

Complete the implementation of the **shortest_path** function such that it returns the shortest path from the person with id source to the person with the id target.

- Assuming there is a path from the source to the target, your function should return a list, where each list item is the next **(movie_id, person_id)** pair in the path from the source to the target. Each pair should be a tuple of two ints.
  For example, if the return values of **shortest_path** were [(1, 2), (3, 4)], that would mean that the source starred in movie 1 with person 2, person 2 starred in movie 3 with person 4, and person 4 is the target.
- If there are multiple paths of minimum length from the source to the target, your function can return any of them.
- If there is no possible path between two actors, your function should return None.

Important: You will be given a set containing pairs of actors. Your task is to submit the lengths of the path between each pair of actors in Moodle. This is how we will grade your code.

Implementation Hints

- You may call the **neighbors_for_person** function, which accepts a person's id as input, and returns a set of **(movie_id, person_id)** pairs for all people who starred in a movie with a given person.
- You should not need to modify anything else in the file other than the **shortest_path** function, though you may write additional functions and/or import other Python standard library modules.
- While the implementation of search checks for a goal when a node is popped off the frontier, you can improve the efficiency of your search by checking for a goal as nodes are added to the frontier: if you detect a goal node, no need to add it to the frontier, you can simply return the solution immediately.
- We've already provided you with a file **util.py** that contains the implementations for Node, **StackFrontier**, and **QueueFrontier**, which you're welcome to use (and modify if you'd like).