50.021 – AI

Kwan Hui

Week 02: Search

[The following notes are compiled from various sources such as textbooks, lecture materials, Web resources and are shared for academic purposes only, intended for use by students registered for a specific course. In the interest of brevity, every source is not cited. The compiler of these notes gratefully acknowledges all such sources.]

Due: 4th June, 11:59pm
Submission: via eDimension

1 Coding: Word Ladders

In this problem we will be writing a Problem specification that we can use with our search code to solve Word Ladder puzzles `https://en.wikipedia.org/wiki/Word_ladder`.

In a word ladder puzzle, you are given a start word and a goal word. You must transform the start word into the goal word by transforming one letter at a time, and all intermediate words must be valid as well. In the following example, you start with the word cold and must reach the word warm:

- cold
- cord
- card
- ward
- warm

Notice that states in this search space are represented by strings, e.g. "oatmeal".

You will need a function `is_valid_word(w)`, which returns True if w is a valid English word, and False otherwise. This function is provided below. You may also assume that all characters are lower case letters (no spaces, no capital letters, no punctuation, etc.).

You can download the file words2.txt to your working directory, and paste the following code into your work file:

```
WORDS = set(i.lower().strip() for i in open('words2.txt'))

def is_valid_word(word):
    return word in WORDS
```

You may find the constants and functions in Python's string module helpful; you need to import string if you want to use it.
Here are some simple test cases:

- "cars" to "cats"
- "cold" to "warm"
- "best" to "math"

2 Coding: Flight Itinerary

Our goal in these exercises will be to make a flight search engine ... just like skyscanner or so, just a bit smaller. A flight is specified with a starting city and time and an ending city and time. Cities are represented as strings and times as integers.

We want to be able to answer queries that are specified with a starting city, starting time, final destination city and deadline time. The objective is to find a sequence of flights, starting in the starting city, leaving sometime at or after the starting time, and arriving at the final destination before (or at) the deadline.

We will formulate it as a search problem.

Part 1: State

What is a good choice of state in this problem?

- Current city
- Current city and destination city
- Current city and current time
- Current city and deadline

Here is the skeleton of a 'Flight' class:

```
class Flight:
    def __init__(self, start_city, start_time, end_city, end_time):
        self.start_city = start_city
        self.start_time = start_time
        self.end_city = end_city
        self.end_time = end_time

    def __str__(self):
        return str((self.start_city, self.start_time))+' -> '
        + str((self.end_city, self.end_time))
    __repr__ = __str__
```

Here is a database of fights:

```
flightDB = [Flight('Rome', 1, 'Paris', 4),
            Flight('Rome', 3, 'Madrid', 5),
            Flight('Rome', 5, 'Istanbul', 10),
            Flight('Paris', 2, 'London', 4),
            Flight('Paris', 5, 'Oslo', 7),
            Flight('Paris', 5, 'Istanbul', 9),
            Flight('Madrid', 7, 'Rabat', 10),
            Flight('Madrid', 8, 'London', 10),
            Flight('Istanbul', 10, 'Constantinople', 10)]
```

Part 2: Matches

Write a method matches contained within the Flight class, that takes a pair (city,time) as an argument and returns a boolean based on whether or not the city and time specified "match" those of the flight, in the sense that the flight leaves from the same city, at a time later than time.

Part 3: Flight Itinerary

Define a procedure `find_itinerary` that returns a plan, in the form of a sequence of (city, time) pairs, that represents a legal sequence of flights (found in FlightDB) from start_city to end_city before a specified deadline.

```
    def find_itinerary(start_city, start_time, end_city, deadline):
      pass
```

Part 4: Going Further

Ben Bitdiddle wants to find a way to get from Rome at time 1 to Istanbul at the earliest time possible. He proposes to start with a deadline argument (to `find_itinerary` of 1, and then increase it, one-by-one, calling `find_itinerary` each time, until it successfully finds a path.

Will this strategy find the path that arrives soonest, given that we start at time 1?

Imagine that if we use this strategy to solve a problem whose shortest path (shortest time) is length 100, it takes an amount x of calls on find_itinerary to solve. Roughly how many calls to `find_itinerary` will it take to solve a problem whose shortest path is length 200?

Write a procedure `find_shortest_itinerary` that implements this strategy.

Your procedure should take two strings, representing a starting location and an ending location, and should return a list of (location,time) tuples representing the shortest path between the two locations. You may assume that there is at least one path connecting the two locations.

As an additional challenge, try to minimize the number of times your procedure calls `find_itinerary`!