Assignment 2: Basic Problem-Solving Techniques

CSE 415: Introduction to Artificial Intelligence
The University of Washington, Seattle, Autumn 2019

Due Friday, October 11 at 23:59 PM.

Introduction

This assignment begins a series of three assignments the cover state-space search in depth. We start by introducing a framework for the classical theory of problem solving. This will prepare the way to study the use of heuristics for state-space search in Assignment 3. Note that this assignment is an "individual work" assignment, and should not be done in partnerships or groups.

Procedure

1. Begin this assignment by reading the 1st chapter of "Applying AI in Problem Solving" (linked from the Readings page).

2. Next, download the starter code for this assignment. Un-tar it using a standard file archiving tool such as tar, WinZip, Windows Compressed Folders, etc.

3. Examine each of the two problem formulation files provided. One is the Missionaries and Cannibals problem. The other is the Towers of Hanoi problem. An additional file is provided, Farmer_Fox.py, and you will be writing one new problem formulation in that file.

In the given formulation files, pay particular attention to how the State class is defined and how the operators are established. We will be talking about this code in class.

4. An interactive solving client is provided. This is implemented in the file Int_Solv_Client.py. **Optional but recommended:** Try running the client with Towers of Hanoi by using the following command in a Linux, Darwin, or Cygwin command shell. To do this, issue the command:

```
python3 Int_Solv_Client.py TowersOfHanoi 3
```

If you are not comfortable working in Linux, Darwin, or Cygwin, you may wish to simply create a working folder with the Interactive Solving Client in it, and all the problem formulation files in it, and edit the importing code of the Interactive Solving Client so that it simply imports the problem formulation of your choice, and then run the program from IDLE, PyCharm, or other

IDE you might be using.

5. Create your own problem formulation for the "Farmer, Fox, Chicken, and Grain" problem covered in class. Then do either option 5a or 5b below to produce a session transcript for your formulation.
Option 5a: Using the interactive solving client, demonstrate the use of your formulation to create a solving-session log. You can simply capture your screen to show the log. If you capture the screen as a text file, name the file `FFCG-log.txt`, If you capture it as an image, name the file `FFCG-log.jpg`, `FFCG-log.png`, or `FFCG-log.gif`, depending on the image file format.
Option 5b: Use the starter-code program ItrDFS.py (described in item 6 below) to create output with your formulation. Capture that output and name the file as in option 5a.

6. Starting with the starter-code file ItrDFS.py, which implements a loop-based depth-first search method ("DFS" in the following), implement BFS (Breadth-First Search) as it is specified in the lecture slides. Make sure that these implementations keep track of "predecessor" links (also known as back links), and they can report a shortest path from start to goal, for whatever problem they are applied to. (You can implement these links using a hash table, i.e., dictionary, that maps each state other than the initial state to its parent state. The initial state should be mapped to a special value such as None, or -1, which is up to you.)

7. Compare DFS and BFS on the following problems: (i) Missionaries and Cannibals, (ii) Farmer, Fox, Chicken, and Grain, and (iii) 4-Disk Towers of Hanoi. For each combination of algorithm and problem, report the following: (a) the path found from start to goal, (b) the length of the path, (c) the number of nodes expanded (i.e., removed from OPEN and had successors generated). Create a file **BlindSearches.pdf** that contains a table showing these results in a clear, easy-to-read manner. Your Towers-of-Hanoi paths can be shown outside the table, since they can be relatively long. This file can be created with Microsoft Word, Google Docs, or similar tool.

8. Starting with the starter-code file `starter-file-for-UCS.py`, implement the Uniform-Cost Search algorithm. The starter file provides you with an implementation of a special kind of priority queue which supports not only the standard `insert` and `delete_min` operations, but also a method called `contains`, which returns True if a given element is in the priority queue. It also allows accessing and updating the priority value of any element in the priority queue. See the documentation for more information. This priority queue is not just adequate for implementing Uniform-Cost Search, but will be particularly helpful in Assignment 3, when it is time to implement the A* Search algorithm.
The main difference between Breadth-First Search and Uniform-Cost Search (UCS) is that UCS works with problem formulations that have different costs associated with different moves. In terms of graph search, there are edge costs, and they are not necessarily assumed to be all equal to 1 as they essentially are in Breadth-First Search. The starter code includes a new problem formulation for you to use with UCS; it's the French Cities network in which the problem is to find a shortest route from the initial-state city to the goal-state city. These are, by default, RENNES and AVIGNON. (These could easily be changed by editing the formulation file's line defining STARTING_CITY, etc.) This formulation is in the file `FranceWithCosts.py`.
The State class in this formulation file has a method `edge_distance(s2)`, which is used to get the

edge cost from the current state to s2. You'll be setting up your UCS.py program to use these cost values to determine the cost of the lowest-cost path from the starting state to any newly reached state.
When your UCS algorithm reaches a goal state, it should do a backtrace of "backlinks" (sometimes called predecessor links) to determine the actual lowest-cost path from the start state to the goal state found. The file `starting-file-for-UCS.py` includes a method `runUCS`. The staff is planning to use an autograder that imports your file and calls this function. It will then be able to check the value of the variable `SOLUTION_PATH` to determine whether your implementation is finding correct answers. For the default initial state and goal state, the results should be

```
['Rennes', 'Nantes', 'Limoges', 'Lyon', 'Avignon']
```

for the `SOLUTION_PATH` variable and

```
1041.0
```

for the `TOTAL_COST` variable. The autograder may also change the initial state and goal state by modifying certain methods of the problem formulation. Thus, your program should be able to find a shortest path for any pair of cities in the given network.
Although there is a Python implementation of Uniform-Cost Search inthe reading, your implementation for this assignment will be different. The code you write will need to fit in with the starter code provided in the file `starter-file-for-UCS.py`. This implementation will then be compatible with the problem formulation files we use in the course, and it will be a very helpful stepping stone to the implementation of A* Search, which you'll get to do in Assignment 3.

Keep in Mind

Here are some ideas to keep in mind for this assignment. The problem format we are using not only permits the "manual solving" that you do with the Interactive Solving Client. It also permits automatic solving, such as with DFS, and BFS. Thus it is important that you follow the problem structure correctly. Certain methods are required in the State class, including __init__, __eq__, __hash__, __str__, and copy. The operators must have the correct structure, as well. The nice thing about the Interactive Solving Client is that it provides a pretty good means to test whether your formulation will be OK.

Even if you are familiar with Python, there may be some constructs in the formulation files that you are not familiar with. You can read up on list comprehensions (p.31 of the first reading) and lambda expressions (pp. 47-52 of that reading). We'll also cover them in class.

What to Turn In

Turn in five files, but NOT zipped up, because that will interfere with the grading workflow. (As an incentive for compliance, the staff will deduct 2 points if the files are zipped.) The five files are the following: (a) `Farmer_Fox.py`, (b) `FFCG-log.txt`, or `FFCG-log.jpg`, or `FFCG-log.png`, or `FFCG-log.gif`, (c) `BFS.py`, (d) `BlindSearches.pdf` and (e) `UCS.py`

Each of the Python files should start with a multiline comment in the following format with your

own information rather than the sample information given here.

```
'''Farmer_Fox.py
by John Nathan Smith
UWNetID: jnsmith98
Student number: 9876543

Assignment 2, in CSE 415, Autumn 2019.

This file contains my problem formulation for the problem of
the Farmer, Fox, Chicken, and Grain.
'''
```

The turn-in method is UW Canvas.

Updates and Corrections

Oct. 9: The "do not zip your files" requirement was added, due to problems with the grading of A1.

Oct. 7: The spec was edited to reflect a change to the starter code for this quarter, in which all problem formulation files and Python implementations of solving methods are in the same, flat folder, rather than in a 2-level folder hierarchy as in the past.

If needed, additional updates and corrections will be posted here, and/or in ED.