Task 6 - Lab: Graphs, Search & Rules

Summary

The purpose of this lab is to give you experience representing games as discrete states. A graph of states allows us to explore the outcome of a game by applying actions, and to search for game results using graph-based search.

In this document, key result steps you need to do have a check box ☐ symbol. Git commit as you go!

We will use simple random search. In later work we will use better, non-random, approaches to search game graphs. The emphasis at this stage is on the usefulness of representing games as sequences of states, and using the selection of game actions to change states, and understanding the limits of random search.

Download the task code from the unit website. It contains two python files, one for the Towers of Hanoi problem (`towers_of_hanoi.py`), and one for a Water Jugs problem (`water_jug_problem.py`). The code is written in a relatively simple procedural style, with output presented as basic ASCII text.

Step 1. Towers of Hanoi - Approach

Read the block comment in the header section of the file (at the top) which describes the game, and the rules. (No point in repeating that here!). If you run the code nothing should happen.

The "state" of the game is the position of all the disks on the three poles. Although we could physically place disks in any order on any pole, the game has rules and so not all positions are valid "states". There is only one "action" for this game, implemented in the `move_disk` function shown below.

```
def move_disk(poles, src, dest):
    '''Move a single disk from the src pole to the dest pole.
    - src or dest = pole index, i.e. 0, 1, 3
    - Does check if there is a piece to move, returns None if not.
    - Does NOT check if it is a valid state, just does the move.
    - Does NOT modify poles - returns a copy with change
    '''
    #check if it's a valid move
    if poles[src]:
        result = copy_state(poles)
        result[dest].append(result[src].pop(-1))
        return result
    else:
        return None
```

Note how this code makes a copy of the state (stored in poles), and returns the new altered state if the move was possible, but it does not check if it breaks the game rules.

☐ Find the "`### 1.`" comment. Change the Boolean flag to True, run, and make sure you see the output and understand what is happening. Have a close look at the `is_valid_state` function. Be ready to explain it to your tutor!

Step 2. Towers of Hanoi - Sequence of Moves

Move on to "`### 2`" and enable it (change the `False` to `True`). This code (shown below) creates a game with three disks (n=3), and then performs the "`move_disk`" action using a pre-set sequence of moves. The sequence is stored as a list of (`src`, `dest`) tuples stored in the "`moves`" variable.

```
    ### 2. Perform a sequence of moves
    if False:
        # do a sequence of moves
        print("Running coded sequence ...")
        n = 3
        s = init_poles(n)
        print_poles_as_text(s, n)
        ### 3. Complete the sequence of moves to solve the game
        moves = [(0, 2), (0, 1)]
        for src, dest in moves:
            print('> Moving from', src, 'to', dest)
            s = move_disk(s, src, dest)
            print_poles_as_text(s, n)
            #print(is_valid_state(s))
            print_poles_as_state(s, test_valid=True)
        print('Done.')
```

☐ Modify the moves sequence (at "`### 3`") and add the additional moves required to complete the n=3 game. Check your answer by looking at the displayed output.

It is easy to do moves that are not valid (break the rules) and result in an invalid game state "s", so be careful and check the tested state!

Because n=3, it should take you seven (7) moves to finish the game optimally. However, we don't really care about "optimal" today … let's solve it using random search! ☺

Step 3. Towers of Hanoi - Random Search

Now that you have solved a simple n=3 problem by explicitly "hard coding" seven moves, let's use random search to create moves, test the result, and see if we can find a valid sequence that solves the problem.

☐ Enable the call to the aptly named "`attempt_using_random_moves`" function (`### 4`) and run. This attempt will fail! Why? Because it always tries to move a disk from pole 0 to 0 until it hits the limit.

☐ In the function, identify the code where the `src` and `dest` are being set to 0 and replace it with the following "`sample`" approach. The sample function is from the `random` module - read the docs!

```
#Raw random sample
src, dest = sample([0, 1, 2], 2)
```

Note that we avoid the pointless case of having `src` and `dest` be the same pole (no change in state) by using sample and not just the random "`choice`" function we've seen before.

See also the "`seed`" function and how it could be used (commented out in the code). Setting seed to a fixed value can be helpful to get repeatable behaviour when developing and debugging code, but not if we are trying to collect a sample of results (which we want to do next).

Run it! The code should be able, sometimes, to find a valid sequence of moves. But not always! You might want to increase the search attempt limit which is initially set to 100. Remember that you explicitly worked out the optimal sequence as a mere seven steps before, so the random search is not likely to be optimal!

☐ For a sample of at least 10 runs, record if the random search was successful and the length of the sequence. (Suggest modifying the code to do this for you, but you can do it by hand if you want!)

Step 4. Towers of Hanoi - Better Random Search?

Can we help the random search do better? In many ways! In general, we can use knowledge of the problem domain (the game rules and states) to avoid actions that don't help. We already avoid the following:

(a) A pointless move to and from the same pole (by using sample instead of choice).
(b) An invalid move that tries to move from a pole that doesn't have any disks.

☐ You may not have realised that (b) was being done. Identify where in the code this occurs.

What else could we avoid?

(c) A move that is simply a reverse of the last move done.
(d) A move that goes back to a state that we have already seen in the history (avoid loops).

Note that (d) is a general case for (c) that you can try as an extension. For now, let's deal with the simple case of (c).

```
#Random sample, but reject if it's a reversal
#-- (no point going backwards all the time ...)
_src, _dest = history[-1]
#this is python style do ... until loop.
while True:
    src, dest = sample([0, 1, 2], 2)
    if (src != _dest) or (dest != _src):
        break
```

☐ Replace the existing sample approach with this snippet of code to avoid last move reversal. Note that it uses the "history" of moves (which the provided code is already doing).

☐ Run the code and collect a sample of at least 10 runs (same as before), recording (again) if the search was successful and the length of the sequence.

☐ Does the change actually reduce the length of the sequences overall? Support your answer with evidence from the data you collect! Is the number of random guesses needed reduced?

As an extension, implement (d). Tip: Convert the state to a tuple, and test the cache to see if you have already seen it. If so, ignore it, else store it. (The cache is not to be confused with the move "history".)

Step 5. (Optional) Towers of Hanoi - Recursion!

As noted in the header document string of the python file, this problem is commonly used as a classic application for recursion to solve. For this lab using recursion is *optional* as it is not the main focus. However, as many students are interested in this approach, it is presented here for completeness.

In the code at "`### 5`" there is a call to the function named "`solve_using_recursion`" that has an incomplete internal function named "`move`" (which currently only has a "`pass`" command).

☐ Enable the call the `solve_using_recursion` function, and complete the `move` function using the snippet shown below.

```
#This is the recursively called "move" function
def move(n, src, dest, aux):
    if n > 0:
        move(n-1, src, aux, dest) #move src disk out-of-the way (using recusion)
        moves.append((src, dest))
        move(n-1, aux, dest, src) #move out-of-way back to dest (using recusions)
```

See how the "`move`" function calls itself - this is the key recursive behaviour and results in concise (elegant?) code. Recursion requires a "stack" in computer memory for it to work. So, for a large number of disks you might run out of "stack space" (a.k.a. the "stack overflow" problem).

☐ Run it! Check with a simple case of n=3 and see that it results in a sequence of 7 moves with a correct final result (state).
☐ Try with a greater number of disks and confirm that the result is valid.

What is going on here? In code, to solve using recursion for number of disks n, we are creating a sequence of move actions to perform which is stored in the "moves" list. The code does NOT actually require the moves to be done (implemented and tested) - it simply creates the sequence of what should be done.

In fact, this recursive sequence generator is ignorant of the game and its rules, but it is able to give us the required sequence. As a result, it works well for this specific case (optimal) but it is not a flexible approach that could be adapted to other games and rules.

Step 6. Water Jugs Problem

Now open the python file for the water jug problem, and again read the block comment in the header section of the file (at the top) which, very simply, describes the game and the rules. We will only spend time on the simple two jug version of the game of unit size 5 and 3, but there are many variations.

The state of the game in this case is the amount of water in each jug. Unlike the last game where we had only a single "`move_disk`" action, this time we have three different actions. We can:

(a) **fill** a specified jug,
(b) **empty** a specified jug, and
(c) **pour** from a specified jug into another specified jug.

The fill and empty functions both require, as parameters, the current state to work on (the jugs) and the target jug to alter. The pour function requires the state, source and destination jugs as parameters. Empty could be considered a "pour" action where the destination is not a jug (i.e. a value of None or similar), but it's nice semantically to keep it as a separate action.

A single "move" is still an action with specific "arguments". For example, a tuple of ("fill", 0) could represent a move of the fill action on "jug 1", and ("pour", 0, 1) the pour action from jug 1 to jug 2. However instead of splitting the two arguments, it is better to group the source and destination, so the tuple becomes ("pour", (0, 1)).

Like the last game, to play we create a sequence of moves to perform and the sequence of game states will result from those moves. (In the code we use a variable called "actions" to represent "moves".) We can use a pre-defined sequence of moves, or a search to go from a start state to a target (goal) state.

It is possible that the target state is unreachable, and it can be a challenge to test or prove this for a complex game graph, but that is an extension topic we will not cover here!

Also like the last game, we could perform pointless or invalid moves. For example, it is pointless to pour from an empty jug, pour into a jug that is already full, or pour from and into the same jug. We could either choose to prevent these moves (to avoid wasted moves) or allow them occur (to allow a very general search approach).

Find "`### 1`" in the code and enable it. Note how the global `JUG_CFG` is set with the jug sizes, as this is used as configuration details by some of the actions. Note how the basic testing is done using assert and print. We could have additional jugs, but that outside the scope of this lab.

☐ Run the code - some of the tests will fail. Fix the assert tests so that they all pass.

Step 7. Water Jugs Problem - Solve with a Sequence

Search for "`### 2`" and enable the block. At "`### 3`" you will see an incomplete "sequence 1". First, run this code and make sure you understand how it works. In particular understand that:

- the `action_calls` variable is a dictionary so that a string can be easily turned in to a function to call,
- the `actions` variable is a list, where each entry in the list is a tuple to represent the "move" that will be done, and
- each move tuple first has string of the function to call, and second a nested tuple of the arguments to use with the function call.

Below you can see the essential part of the action calling code. It includes a commented-out print statement that might assist you in debugging or understanding the code.

```
    # run sequence of actions
    s = setup_jugs()
    for fn, args in actions:
        #print('Calling...', fn, 'with', args, 'on', s)
        s = action_calls[fn](s, *args)
        print(s)
    print('Done')
```

Note how we get `fn` (a string) and `args` (a tuple) for each move in actions, and look up the function in `action_calls` and call it by passing in the current game state (the jugs) and the `*args`.

The "*" in front of the `args` tells python to expand it and use it as a list of arguments, so it will work for the fill and empty commands that expect one jug argument, as well as with the pour command that requires two jug arguments.

(In python there is also a double "*" syntax that will convert a dictionary argument, such as `**kwargs`, and expand it as named argument pairs. This can be very handy as a flexible configuration structure.)

☐ Sequence 1 only requires one more move to reach the goal state. Add it, and show that it works.
☐ Disable `### 2` and enable `### 4`. This is the start of an alternative sequence. Add to the list of actions to solve the problem using the alternative sequence (which is documented in the header of the file).

Step 8. Water Jugs Problem - Random Search

As before let's try a random search approach for this game. If we consider all possible moves (actions with arguments) it is clear that it is a finite set of only eight unique moves for a two-jug problem. If we exclude pointless actions (pour from 1 to 1 and pour from 2 to 2) there are actually only six unique actions.

We can declare these unique moves, and then simply make a random "choice" for each step of our search. See below how this is declared and how choice is used with it later. Note that the functions are being directly referenced in each move, not as a string (that we later map to a function as we did before).

```
    # there is a set of six unique actions to choose from
    actions = [
        #all possible fill's
        (fill, [0]),
        (fill, [1]),
        #all possible pour's
        (pour, [0, 1]),
        (pour, [1, 0]),
        #all possble empty's
        (empty, [0]),
        (empty, [1]),
    ]
```

```
    #select a random action to try
    fn, args = choice(actions)
    new_s = fn(s, *args)
```

To start with, we use this code to see if `new_s` is valid (not `None`) and if so, add it to the `history`.

```
    #if move outcome state is valid (not None) keep it
    if new_s:
        history.append((fn, args))
        if new_s == s_end:
            status = 'Success'
```

- ☐ Run it and check to see if it works! Does it succeed? Increase the limit size if you think it is needed.
- ☐ If the search is successful, how often does it find the goal state?
- ☐ How many moves are there (size of history) when a search is successful?

Step 9. Water Jugs Problem - Better Random Search?

From the last step you would have seen that, even with a very large limit, we rarely find the goal state. Also, if you look at the actual states generated (using `print(new_s)` for example) you will see many (0,0) states, and they are not affected by 4 of the 6 actions! The result state would be the same as the initial state for the pour and empty actions. Probability is against us - let's improve the odds!

Set of Goals! For starters, we are trying to find a single goal state, but our target state is "4" in the larger jug (in the Die Hard 3 movie example) and we don't care what is in the other jug. So this is actually a set of goal states (even though they might be impossible to get to?!). The set of goal states are (4,0), (4,1), (4,2), (4,3) and we can store them in a list to test against using the "in" test with a list.

- ☐ Before the search loop, define the list of valid `end_states`:

```
end_states = [(4,0), (4,1), (4,2), (4,3)]  #not all possible, but valid
```

- ☐ Now, instead of the simple `new_s == s_end` test, use a list "`in`" test.

```
    if new_s in end_states:
        status = 'Success!'
```

- ☐ Run this version.
- ☐ Does it succeed more often than the previous version?
- ☐ How many moves are there (size of history) when a search is successful?

Don't Do Nothing! If you look at the list of output states, you will see that we are still doing some moves that result in no change. Let's reject moves that don't help (no state change).

- ☐ Update the simple `if new_s`: test, and also comparing the `new_s` to the last state `s`, to make sure it is different. If not, we ignore the new move and don't keep it in the history. Run this version

```
if new_s and new_s != s:
    s = new_s
    history.append((fn, args))
```

- ☐ If it succeeds, is the solution path (the sequence of states in the history) smaller?
- ☐ If it succeeds, is the guess count (the amount of random guess work done) less?

Don't Do Loops! (Optional) Implement a cache of visited states. Use it to reject moves that take you back to a previous state. Test if this actually helps the frequency and / or quality of search results.

Save your work, commit and push. Upload a text file to Canvas of your commit history to indicate you are ready to present your working code to your tutor.