Lab 2: Uninformed and informed search

Objectives:

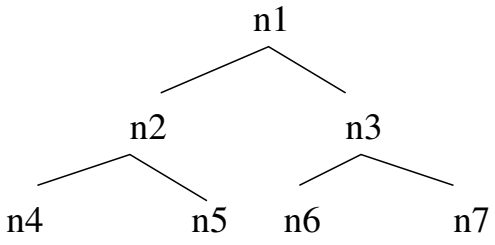
- **To review your understanding of the look-ahead search**
- **To guide you through the setup of your Python environment and the PyCharm IDE for the labs in this paper (if you haven't done it in Lab 1)**
- **To guide you through implementation of A* tree-search**
- **To test your understanding of A* with an exercise where you implement an A* graph-search.**
- **To demonstrate the importance of the choice of the heuristic for the speed of the A* search.**

Preparation:

- **If you're new to Python, familiarise yourself with the fundamentals;**
- **possible tutorials to start with: an introduction to Python and more on Python flow of control**
- **Answer the questions below on "Uniform search concepts"**

Uniform search concepts

1. What's a look-ahead search problem? Give an example.
2. How can a look-ahead search problem be formalised as a **state space**?
3. Consider the search-tree below.



What order would the nodes be encountered if the graph was searched

(a) depth first left-to-right?

(b) breadth-first right-to-left?

4. Alternative search strategies

   (a) Summarise the advantages of depth-first and breadth-first search.

   (b) How does iterative deepening search obtain the advantages of both methods while avoiding their disadvantages?

A quick introduction to Python 3 and PyCharm

In these labs we're going to be using Python 3.10 and PyCharm IDE for programming exercises. Hopefully you had already some exposure to this setup in Lab 1, but it was in a group setting, so let's go over some basics here on an individual basis.

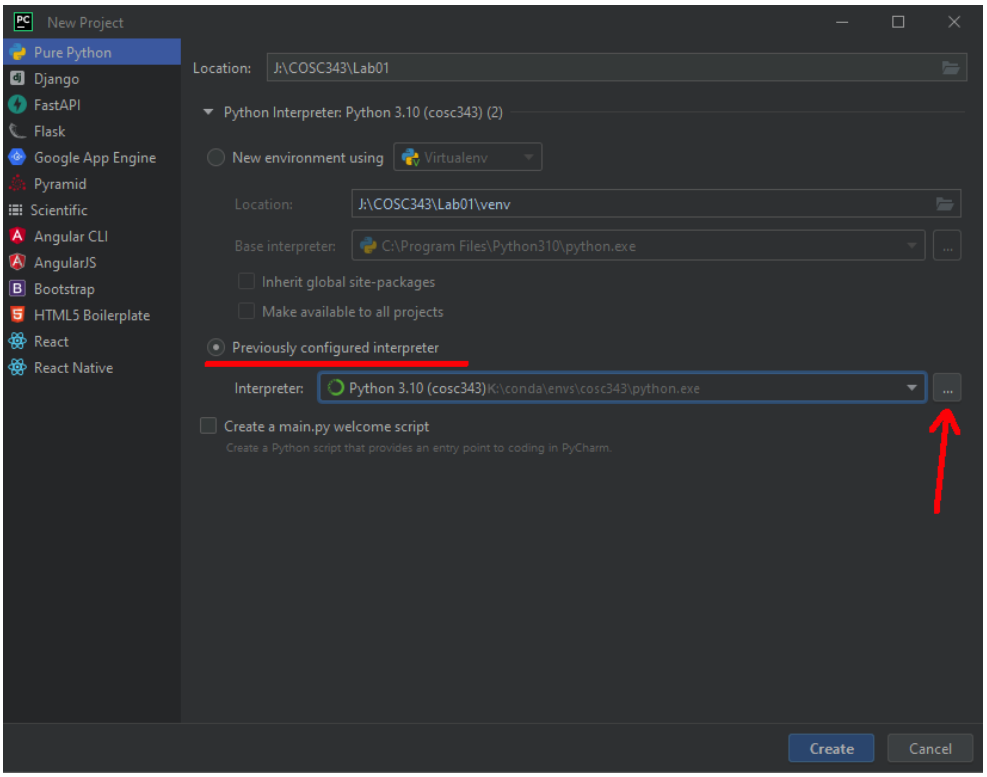
Python-based ecosystem is a good environment for implementing and testing AI algorithms, and visualising their performance. If you're new to Python it might be worthwhile to first go over an introduction to Python and more on Python flow of control.

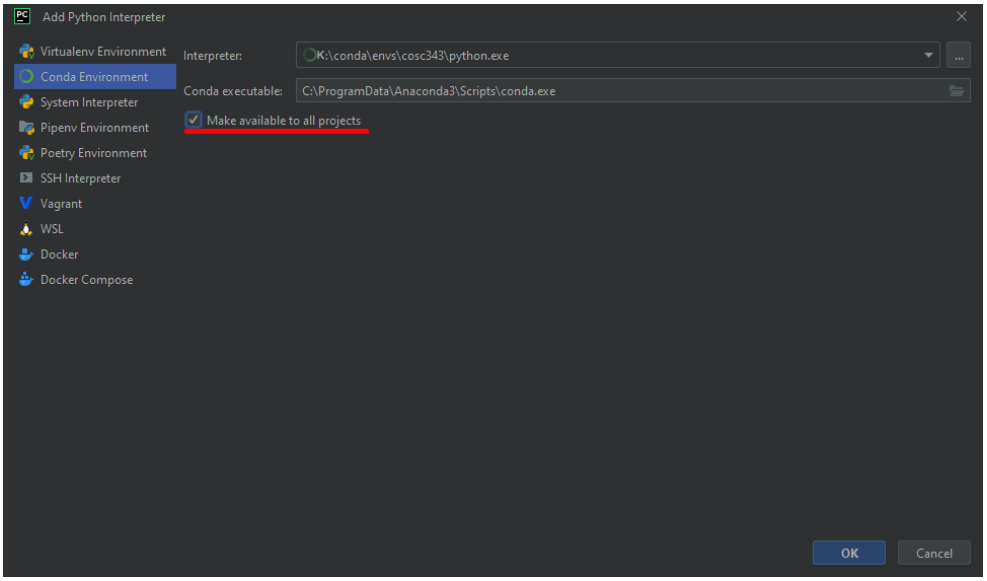
1. A virtual environment for Python has been created for you on all the lab machines. If you haven't done this in Lab 1, you need to add few things to your profile configuration in order for PyCharm to be able to pick it up. Follow the "First time PyCharm setup (in the Owheo lab)" instructions in the "Labs" section of the Blackboard page for AIML402/COSC343, before you proceed with the next step; follow "First time Python+PyCharm setup (on your own device)" if you're using your own device.

2. Create a new Pure Python PyCharm project. Make sure to select cosc343 Anaconda environment for the Project Interpreter. If you don't have the option of the interpreter with that path as shown in the screenshot below, click on the [...] button to the right and select "Add Local Interpreter". In the window that comes up, on the left-hand side select "Conda Environment" and browse to the python executable of the cosc343 anaconda environment:

   - on the lab machine this will be:
     `K:\conda\envs\cosc343\python.exe`
   - on your device this will be the location of the python executable of the cosc343 Anaconda environment, such as for instance:
     `/Users/lechszym/anaconda3/envs/cosc343/bin/python`

   Select "Make available to all projects" and press OK.





3. From the "Labs" section of the Blackboard page for COSC343 download `cosc343EightPuzzle.py` and save it to the directory of your project. This script provides a class with the 8-puzzle environment that will be used for the following exercise.

4. From the menu select `File->New`, and then select `Python File`. Name the file `exercise1.py` and type in the following code:

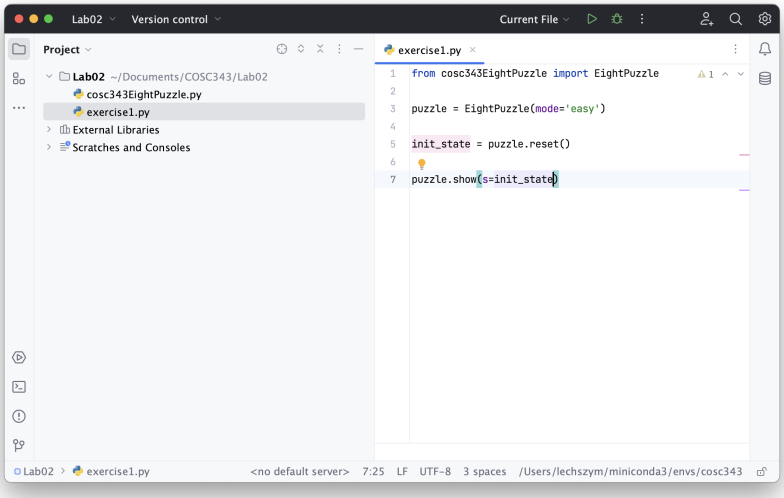
```
from cosc343EightPuzzle import EightPuzzle

puzzle = eightpuzzle(mode='easy')

init_state = puzzle.reset()

puzzle.show(s=init_state)
```

5. Your PyCharm window should look something like the screenshot below. On the left hand side you should see the project window showing all the scripts (if you don't see them, select `View->Tool Windows->Project` from the menu). The contents of your `exercise1.py` script should be displayed on the right-hand side.



6. Here's what the current code in `exercise1.py` does:

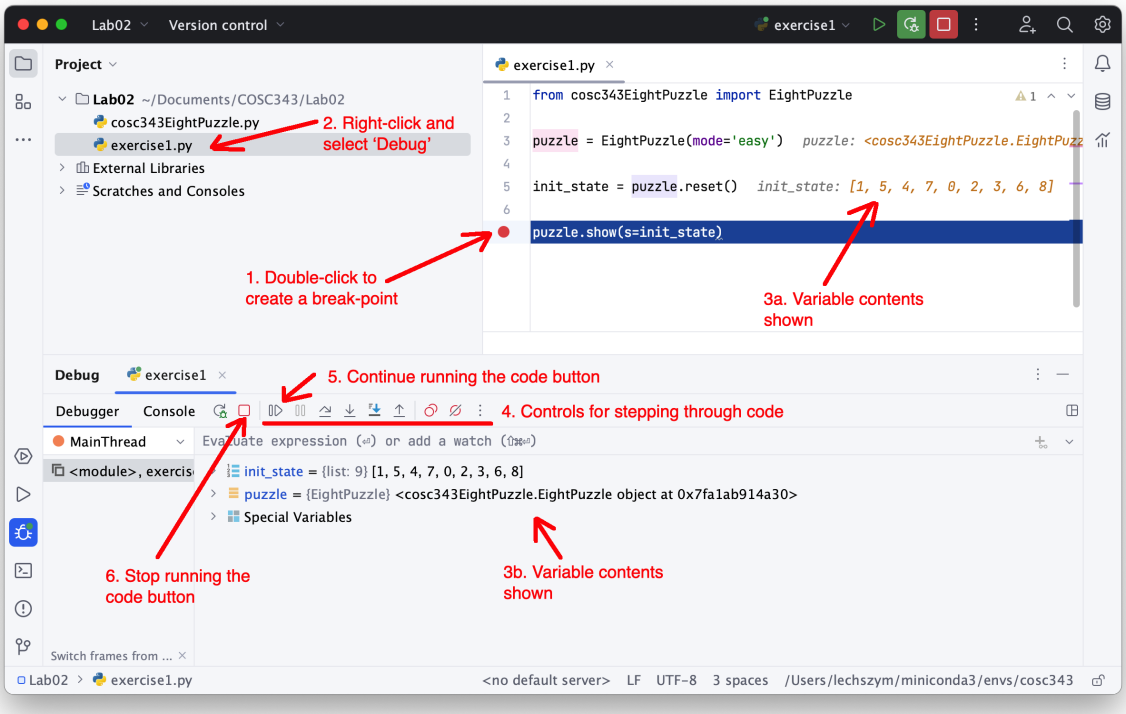
   - The line "`from cosc343EightPuzzle import EightPuzzle`" imports a class called `EightPuzzle` that has been defined inside the script `cosc343EightPuzzle.py`.
   - The line "`puzzle = EightPuzzle(mode='easy')`" instantiates a new instance of `EightPuzzle` class and stores a reference to that object in the variable called `puzzle`. The `EightPuzzle` class has a constructor that accepts an argument called `mode` and expects it to be a string from the following choices: 'easy', 'medium', or 'hard'; depending what string you pass into that constructor, the starting state of the puzzle will be different.
   - The `EightPuzzle` class implements a method called `reset`, which takes no arguments; this method resets internal state of the puzzle to the starting state and returns a description of that initial state as a list; the line "`init_state = puzzle.reset()`"

invokes that method on the object referenced by the `puzzle` variable and stores the returned representation of the initial state in the variable called `init_state`.

- The `EightPuzzle` class implements a method called `show`, which takes a variable number of arguments. This method will show a visualisation of the state passed into the method through the argument `s`. The line "`puzzle.show(s=init_state)`" invokes that method on the object reference by the `puzzle` variable and shows visualisation of the state stored in the `init_state` variable.

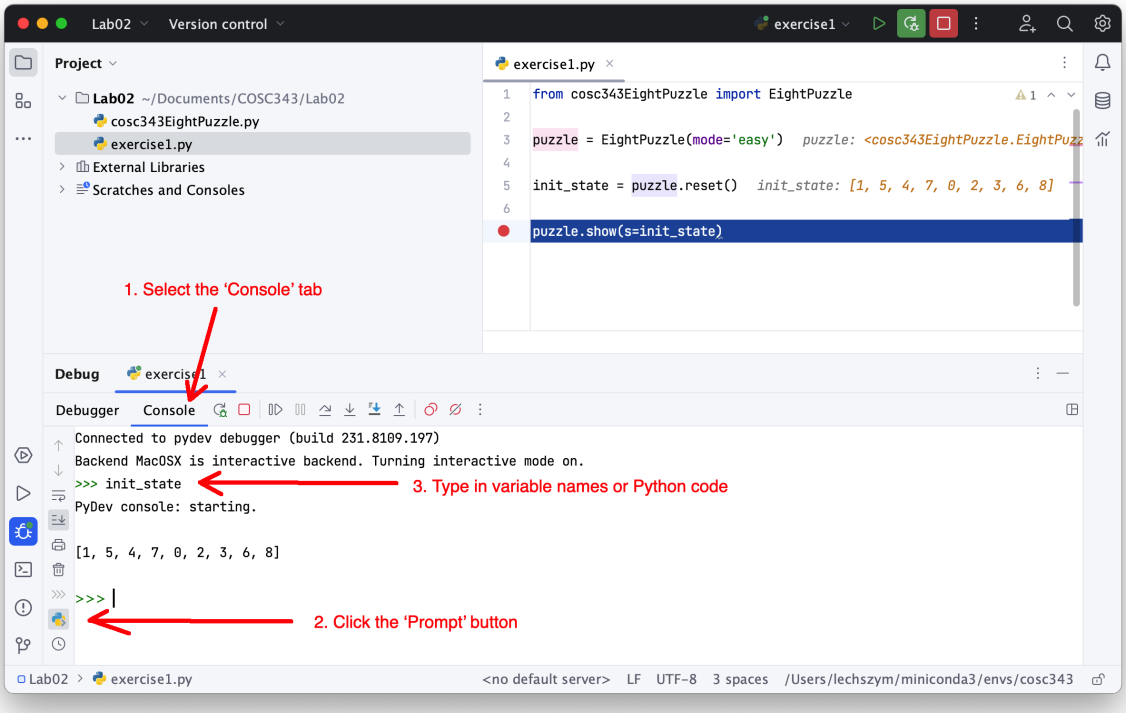
7. To run the `exercise1.py` script, right-click on it in the Project pane and select `Run 'exercise1'`. You might want to change the configuration of the figure display in your project in order for the visualisation to work properly – from the menu select `PyCharm->Preferences`, then select `Tools->Python Scientific` and de-select "Show plots in tool window".

8. PyCharm debugger makes it easy to step through and inspect your code as it is running. In the window, where the contents of `exercise1.py` are shown, double-click on the left-hand side of the line that invokes the `show` method of the `puzzle` (object near the line number) to create a break-point. Now, right-click on `exercise1.py` in the Project pane and select `Debug 'exercise1'`. The program will run and stop at the break-point.



9. You can inspect the contents of variables. For instance, the `init_state` variable should contain a list of numbers corresponding to a 3×3 board of buttons read left-to-right, top-to-bottom with 0 indicating the empty spot; you can step through the code/continue/or stop.

10. You can also interact with the variables through the python shell; select the `Console` tab in the bottom panel and click on the `Prompt` button; now you can type in variable names to see their contents and execute python statements in the shell.



Informed search

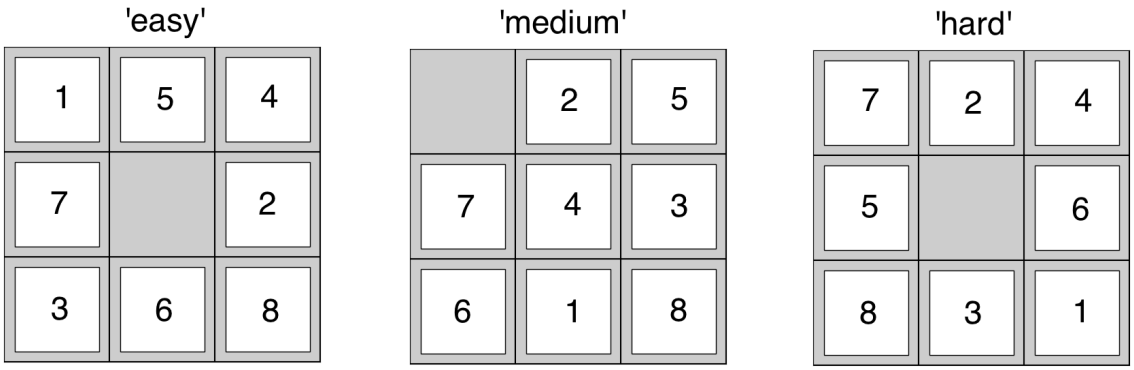
Exercise 1: In this exercise you will implement the A* tree search to solve the 8-puzzle from a given starting state. If you haven't done it already, create a new project in PyCharm (don't forget to set the interpreter to the cosc343 environment). Download `cosc343EightPuzzle.py` from Blackboard and place it inside your projects folder. The `cosc343EightPuzzle.py` script provides a class with the 8-puzzle environment that will help you with a number of aspects of the transition model, valid actions and visualisations. Create a new script and import the `EightPuzzle` class from the provided script.

```
from cosc343EightPuzzle import EightPuzzle

puzzle = EightPuzzle(mode='easy')
init_state = puzzle.reset()
```

The line "`puzzle = EightPuzzle(mode='easy')`" instantiates the `EightPuzzle` class in the 'easy' mode (the other modes to choose from are 'medium' and 'hard'). The difficulty mode relates to how many moves are required in order to solve the puzzle. It's a good idea to start in the easy mode before tackling the harder ones, which may require a bit of optimisation in order to find the solution in a reasonable amount of time.

The line "`init_state = puzzle.reset()`" resets the environment into its starting state and returns that state as a list of 9 numbers.



The state is expressed as a permutation of 9 digits (from 0 to 8) which correspond to the tile numbers (0 being the empty tile). The tiles are listed from left to right and top to bottom. And so, the three starting states from the figure above correspond to the the following states:

- 'easy' – `[1,5,4,7,0,2,3,6,8]`
- 'medium' – `[0,2,5,7,4,3,6,1,8]`
- 'hard' – `[7,2,4,5,0,6,8,3,1]`

To visualise a particular state, you can do

```
puzzle.show(s=current_state)
```

where `current_state` is a permutation of the 9 digits from 0 to 8. You might need to disable the "Show plots in tool window" option in Preferences→Tools→Python

Scientific in order for this Matplotlib figure to work. The call to the plotting method from your Python script blocks the execution, so when you run the program and get a figure, you need to close the figure window in order to continue running the script.

The point of this exercise is to implement a search that finds the sequence of actions that solves the puzzle from a given starting state. Normally, the lab script will explain the functionality of the provided scripts and require you to do the implementation on your own. However, given this is the first proper lab where we introduce Python, the solution for the 'easy' case will be provided, walking you through various parts of the Python syntax.

First let's create a helper class for a node that will allow us to build a tree of nodes. Type the following code into `exercise1.py`, right after the import statement, but before the line that instantiates the `EightPuzzle` object (you don't have to copy the comments).

```
# Definition of a class named 'Node'
class Node:

    def __init__(self, s, parent=None, g=0, h=0, action=None):
        self.s = s                  #State
        self.parent = parent        #Reference to parent node
        self.g = g                  #Cost
        self.f = g+h                #Evaluation function
        self.action = action        #Action taken from parent
                                    #nodes state to this node's
                                    #state
```

Python scripts are typically structured as follows: import statements at the top, followed by any definitions of classes and functions that are used by the script, and then the statements of the script itself. Essentially, the interpreter needs all the definitions of classes and functions (whether they come from other libraries or are implemented by your script) before running the script.

Note that in Python there are is no '{' and '}' to indicate code blocks, instead you rely on indentation for definition of scope. You have no choice, but use proper indentation for the program to work. In this case the class 'Node' contains only one method. The "`__init__(self,...)`" method is a specially reserved name for the constructor of the class – it's invoked automatically when an object of this class gets instantiated. The "**self**" keyword is a special keyword that refers to the object's instance (equivalent to **this** in Java). In Python "**self**" must be the first argument of every member method of a class. The constructor of the `Node` class takes several arguments: `state`,`parent`, `g`, `h` and `action`. The arguments that are assigned a default value (with the `=` sign) don't have to be specified explicitly when the method

is invoked. Since `state` has no default value, the constructor of the `eightpuzzle` requires at least this argument. The `Node` class doesn't do much – it just stores data from the arguments in the created object and computes the sum of `g` and `h`, which corresponds to the evaluation function $f(n) = g(n) + h(n)$ (refer to Lecture 4 notes if you don't know where this formula comes from). The "**`self.f`**" syntax means that "`f`" is a member variable specific to instance of the object of the class. Hence, each instance of the `Node` object will have its own copy of the value of `self.f`.

Next, after the definition of the `Node` class, but still before the instantiation of the `EightPuzzle` class, add the code given below. It implements the heuristic function that counts the number of tiles in the wrong place (as compared to the `goal` state). Make sure that the "**`def heuristic(current_state,goal_state):`**" IS NOT indented under the scope of the `Node` class:

```
# Function computing misplaced tiles
def heuristic(s,goal):

    h = 0
    # Walk through all tiles in the current state
    for i in range(len(s)):
        if s[i] != goal[i]:
            h += 1
    return h
```

This function expects two lists: one corresponding to the tiles in the current state and the other to the goal state. It calculates the number of misplaced tiles. Inside, there's a `for` loop that iterates over the items in the list. This loop will index i over a `range` of numbers. Python's built-in `range` function accepts either 1, 2 or three arguments. When just one argument is used, the `range` function interprets it as the highest (excluded) value in the range, and thus generates a range from 0 incrementing by one up to largest integers smaller that the highest value. The length of a list is computed using Python's built-in function `len`. Since the list will always contain 9 items, the loop will iterate `i` from 0 through to (and including) 8 (in increments of 1). The expression `s[i] != goal[i]` compares the i^{th} item in both lists. If they are not the same, 1 is added to the misplaced tile count.

OK, given definitions of then `Node` class and a `heuristic` function, we can start implementing A*.

To get the the goal state for the solution, you can use the provided method of the `EightPuzzle` class:

```
goal_state = puzzle.goal()
```

Place that code right after fetching the initial state from the `puzzle` object. Next, create the root node of the search tree like so:

```
root_node = Node(s=init_state,parent=None, g=0,
                 h=heuristic(s=init_state, goal=goal_state))
fringe = [root_node]
```

The first line creates a new object of `Node` type, calling its constructor and passing-in a bunch of arguments. The root node's state is the initial state, there is no parent (`None` in Python is a bit like `null` in other languages), there is a zero cost up to that point and the heuristic for the distance from the initial to the goal state is passed in directly from the output of the `heuristic` function.

The second line creates a new list called `fringe` with single item, the reference to the root note object.

Next, type in the following code:

```
solution_node = None
while len(fringe)>0:
    current_node = fringe.pop(0)
    current_state = current_node.s
```

Here we start a while loop that will keep going until the `fringe` is empty. First time around, the `fringe` list will have exactly 1 item, so the while loop is guaranteed to execute at least once. The line "`current_node = fringe.pop(0)`" removes the first item (one at position 0) from the `fringe` list using lists `pop`. The removed item is now referenced by variable `current_node`. Then, a reference to the node's state is created and stored into the variable `current_state`.

Now type the following if statement, making sure it's indented so that the entire block is within the while loop (the `if` being flush with the `current_state` above):

```
    if(current_state == goal_state):
        solution_node = current_node
        break
    else:
        available_actions = puzzle.actions(s=current_state)
        for a in available_actions:
            next_state = puzzle.step(s=current_state,a=a)
            new_node = Node(s=next_state,
                            parent=current_node,
                            g=current_node.g+1,
                            h=heuristic(s=next_state,
                                        goal=goal_state),
                            action=a)
            fringe.append(new_node)
        fringe.sort(key=lambda x: x.f)
```

The if statement checks if the current state is the same as the goal state – the `==` between two lists in Python just compares them item by item. If solution is found, the reference to the `current_node` is saved in the `solution_node` and the while loop is broken. If the `current_node` is not the same as the goal state, then possible actions from the current state are considered and the states resulting from taking those actions are added to the fringe.

The `EightPuzzle` class provides a method that gives you a list of actions that can be taken from the state passed in as argument. This method is used in the line "`available_actions = puzzle.actions(s=current_state)`". It determines all valid actions allowed from the passed-in state and returns these actions as a list of numbers. The encoding of actions is as follows:

- 0 - move a tile to the right into the empty space
- 1 - move a tile up into the empty space
- 2 - move a tile to the left into the empty space
- 3 - move a tile down into the empty space

The for loop that follows iterates over all actions in the `available_actions` list. Note that this loop does not iterate over a range of indexes according to the length of the list. The syntax of this for loop makes it operate like a `foreach` loop – the `a` will take on values of different items from the list.

To `EighPuzzle` class provides a method that returns the state that results from taking action `a`. This is what the line "`next_state = puzzle.step(s=current_state,a=a)`" does.

Then a new node is created with the new state, and this node is meant to be the child of the current node in the state tree. The new node takes the next state as its state and the current node as the parent. The passed-in cost is computed by adding 1 to the overall cost (cost corresponds to the total number of moves when solving the puzzle). The passed-in heuristic (that will get added to the cost for evaluation function) is computed between the next state and the goal.

The new node is then added to the `fringe` list by using the `append` method of the list object.

Once the new nodes from all possible actions are added, the `fringe` list is sorted. Lists in Python provide a `sort` method, but because the items on the `fringe` list are references to nodes, simple sorting would just rearrange the references to nodes, and not the nodes according to the value of their `f` member variable. The `sort` function can take a comparison function as an argument, and the special syntax "`key=lambda x: x.f`" is a code of an anonymous function that tells the `sort` to compare its items by their `.f` values. So, after this call, the node with the lowest `.f` value will find itself at the beginning of the `fringe` list, ready for popping and expanding on the next iteration of the while loop.

Finally, at the end of your script add the following code:

```
if solution_node is None:
    print("Didn't find a solution!!!")
else:

    action_sequence = []

    next_node = solution_node
    while True:
        if next_node == root_node:
            break

        action_sequence.append(next_node.action)
        next_node = next_node.parent

    action_sequence.reverse()
    print("Number of moves: %d" % solution_node.g)

puzzle.show(s=init_state, a=action_sequence)
```

The if in the above code is flush with the while loop above. Essentially, once the while loop breaks, the script checks if `solution_node` has been set. If not, it means the entire tree has been searched without finding a solution. However, if it has been

set, the code under the `else` condition walks the search tree back from the solution node to the root node recording all the node actions in a list. Once the root node is reached, that list gives the sequence of actions from the solution back to the start state. So, in order to give actions from the start state to the goal, i needs to be flipped. This is done using the list's method `reverse()`. Remember, lists and their method are built in Python. The `EightPuzzle` class and its methods all come from the `cosc343EightpPuzzle.py` script (feel free to inspect it).

The `show` method of the `EightPuzzle` class is implemented in such a way, that if you pass in the list of actions via the `a` argument it will show you the animation of the actions taken from the state given by the `s` argument.

As a final touch, import the time library and wrap your code between the calls to `time.time()` in order to measure how long your search takes:

```
# Other imports
import time

# Your definitions of node and heuristic function

start_time = time.time()
.
.
.
# Your search code here
.
.
.

elapsed_time = time.time() - start_time
print("Elapsed time: %.1f seconds" % elapsed_time)
```

How long is your search for the easy solution? It should be fairly fast for the 'easy' puzzle. Try the 'medium' one. Do you have the patience to wait for it to finish?

Exercise 2: The A* tree search implementation given above is pretty inefficient. Your second task is to modify it for graph-search so that it solves the 'hard' puzzle in around 2s.

Here are some hints for speeding up your A* algorithm:

- Is there a different heuristic that might work better/faster?
- Is your graph search implemented correctly?