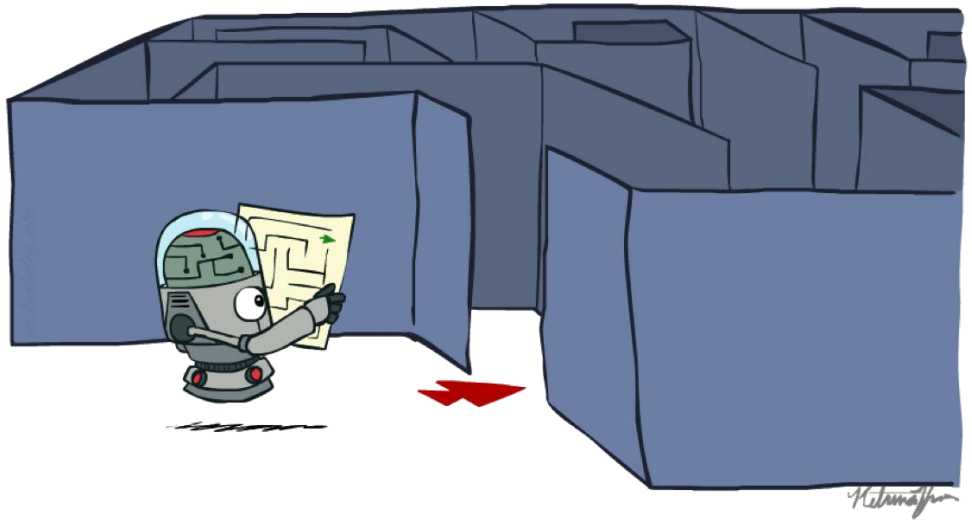
CSE 3521: HW1_Programming: Search Algorithms



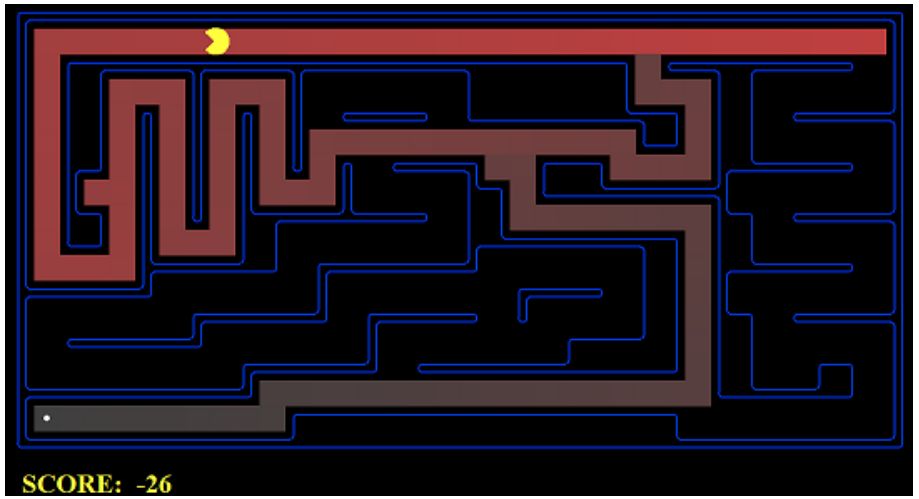
THE OHIO STATE UNIVERSITY

General

- Please note that this is NOT a programming course. Thus, the TA and I are not to help step-by-step debug your code (e.g., syntax or semantic errors). We may provide suggestions for how to get familiar with Python and help point out which part of your code might go wrong, but it is your job to implement the search algorithms, and the implementation itself involves debugging. It is important that you first go through the lecture slides or suggested reading (on the website) to understand the concepts of the search algorithms and the graph search before implementation.

Introduction

- In this homework, you are to implement four search algorithms --- DFS, BFS, UCS, and A* --- so that a Pac-Man planning agent can complete the search problem. You are to implement the "graph" search rather than the "tree" search version.



  - Please note the difference between the “search process (nodes that are expanded)” and the “solution (a path from the start state to the goal state)”
  - What to return is a solution (i.e., a sequence of actions)
  - There is an autograder that you can run

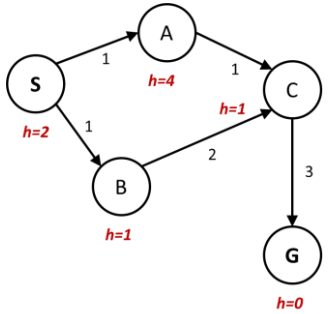
Introduction

- All the search algorithms are the same except for fringe/exploration strategies. Thus, focus on implementing the first algorithm (say DFS) and the others will become easier.

- A search problem
  - Start state
  - Goal states
  - Successor functions: with actions, costs, and next states

`problem.getStartState()`
`problem.isGoalState(state)`
`problem.getSuccessors(state)`

- What you need to return:
  - A list of actions!



Example: [“go north-east”, “go south-east”, “go-south”]

Introduction (in search.py)

You don’t need to implement any of these on the right-hand side or bottom side. Just read through them and understand the input and output of these functions.

```python
def nullHeuristic(state, problem=None):
    """
    A heuristic function estimates the cost from the current state to the nearest
    goal in the provided SearchProblem.  This heuristic is trivial.
    """
    return 0
```

```python
class SearchProblem:
    """
    This class outlines the structure of a search problem, but doesn't implement
    any of the methods (in object-oriented terminology: an abstract class).

    You do not need to change anything in this class, ever.
    """

    def getStartState(self):
      """
      Returns the start state for the search problem
      """
      util.raiseNotDefined()

    def isGoalState(self, state):
      """
      state: Search state

      Returns True if and only if the state is a valid goal state
      """
      util.raiseNotDefined()

    def getSuccessors(self, state):
      """
      state: Search state

      For a given state, this should return a list of triples,
      (successor, action, stepCost), where 'successor' is a
      successor to the current state, 'action' is the action
      required to get there, and 'stepCost' is the incremental
      cost of expanding to that successor
      """
```

Other notes

- Please carefully read the pseudo-code of the "graph" search in the slides. It is very important where you:
  - (a) check the closed-set
  - (b) insert states into the closed-set
- I have seen some earlier cases where you may do (a) when you add a node into the fringe but not when you expand that node. There is a big difference between them.
- If you are not sure what the differences are, create some toy examples yourselves (on the paper), and go through them. This is what I always do to find out some cases that I may miss considering.

An example template (no need to follow this!)

Template code for DFS (you can extend the idea to other algorithms with suitable changes):

```
def depthFirstSearch(problem):
 """
 Search the deepest nodes in the search tree first.
 Your search algorithm needs to return a list of actions that reaches the goal. Make sure to implement a graph search algorithm.
 To get started, you might want to try some of these simple commands to
 understand the search problem that is being passed in:
 print("Start:", problem.getStartState())
 print("Is the start a goal?", problem.isGoalState(problem.getStartState()))
 print("Start's successors:", problem.getSuccessors(problem.getStartState()))
 """
 closedset = []
 openset = [problem.getStartState()] # openset starts with starting state parents = {}
 while len(openset) > 0:
  state = openset.pop() # get most-recently-added element from openset
  # ...
  if # ...
   print("Found goal!")
   # retrieve series of steps that brought us here (use the parents map) actions = []
   while state != problem.getStartState():
    # ...
   print(actions) # just to see the resulting actions
   return actions
  else:
   for (next_state, action, cost) in problem.getSuccessors(state):
    # next_state is something like (4, 2) (coordinates)
    # action is something like WEST
    # cost is a number (integer or float), not used for depth-first search
    # ...
```

Solution Path

- Having found a goal state (goal test returned true), how do we construct the path used to reach it?
- As presented, open set has no information about path(s) that generated contents
- Answer: Augment state to contain this information
- What information is needed?
  - Predecessor: State that this state was reached from. I.e., state that this state is *successor* of.
  - Action: What action was performed to move from the predecessor to this state
  - (Can also add cost for UCS, etc.)

Augmented States

- Wrapper object, contains state plus additional information
- A Python tuple (,) is convenient for this:
  - `(state,predecessor,action)`
    - state: The original (successor) state we are wrapping up
    - predecessor: The wrapper object (this is important!) of the predecessor state
    - action: The action performed to reach successor from predecessor state
- (Note, for UCS and A*, you can extend this to include other information you need, e.g., costs.)

Augmented States

- For example:

```
openset = [ (problem.getStartState(),None,None) ]
#The starting state has no predecessor!

#...

pred_wrapper = openset.pop()
state,predecessor,action = pred_wrapper

#...

for next_state,action,cost in problem.getSuccessors(state):
  #...
  next_wrapper=(next_state,pred_wrapper,action)
  #Add the wrapper to open set, as appropriate
  #...
```

Solution Path (cont'd)

- To generate solution path:
  - Follow chain of predecessors
    - Goal state
    - Goal state's predecessor
    - Goal state's predecessor's predecessor
    - Etc.
  - Stop when reach initial state
    - Equivalently: until predecessor==null
- Note: Solution path starts with initial state, need to reverse order!

Solution Path (cont'd)