Informatics 2D. Tutorial 1
Search

Kobby Nuamah, Michael Rovatsos

Week 2

1 Agents and Environments

Consider the following agents:

1. A robot vacuum cleaner which follows a preset route around a house and vacuums if it senses dirt, using a camera on the robot's underside. When it has vacuumed the whole house it returns to a recharging station and waits until the next day to vacuum again.
2. A chess-playing agent which tries to checkmate the opposing player.
3. A robot football player which must help its team score goals and prevent the opposition from scoring, as well as conserve energy and avoid damaging itself.

Question 1.a) Classify these agents according to the following types: simple-reflex based, model-reflex based, goal based or utility based.

Question 1.b) Answer the following questions for each agent, giving a short justification for your answer:

1. What is the agent's environment?
    - Fully or partially observable?
    - Deterministic or Stochastic?
    - Discrete or Continuous?
    - Single Agent or Multi-Agent?
    - Episodic or Sequential?
    - Static or Dynamic?
2. What are its percepts?

3. What are its actions?
4. What are its internal models, if any?
5. What are its goals, if any?
6. What are its performance measures, if any?

1.1 Tutor's Guide

For this question there may be alternative answers that I haven't considered. As long as students can provide a justification based upon the definitions of the terms they use then their answers are ok.

Question 1.a):

1. Model-based reflex agent since it has a model of its environment (to know when it has vacuumed the whole house) and it only vacuums based upon its immediate sensory evidence. It does no planning so it is not a goal or utility based agent.
2. Goal-based agent since it only has the one goal of trying to checkmate the opponent and it plans actions to achieve this.
3. Utility-based agent since it has to deal with multiple goals at the same time and find the best way to simulaneously achieve them in a stochastic domain.

Question 1.b):

For the vacuum agent:

1. The vacuum environment is:
    - partially observable (sensors only report what is directly under the agent),
    - deterministic or stochastic (depending upon whether movement and suck actions always have their predicted effects),
    - discrete or continuous (depending upon the level of detail returned by the sensors),
    - single agent,
    - episodic (each piece of carpet is a different episode),
    - static (assuming that the carpet doesn't become dirty, or clean, while the agent is sensing it).
2. Dirt sensor
3. Movement and suck actions

4. Internal model of whether the whole house has been cleaned during the current day.
5. None (note that it doesn't have a goal of cleaning the house, it is just designed to behave in such a way that the house is kept cleaned)
6. None

For the chess agent:

1. Chess board is:
    - Fully observable
    - Deterministic
    - Discrete
    - Multi-agent
    - Sequential
    - Static
2. Agent can sense the current board state.
3. Piece movement actions.
4. Agent has an internal model of the board and its history.
5. Agent has the goal of trying to checkmate its opponent.
6. None

For the robot football player:

1. Football field is:
    - Partially observable (agent has limited sensors).
    - Stochastic (kicking the ball might not send it in the direction the agent wants).
    - Continuous (e.g. continuous battery level).
    - Multi-agent
    - Sequential (using battery power now means having less to use later).
    - Dynamic (another player might take the ball when the agent is deciding what to do with it).
2. Sensors for damage, battery level, location, balance, ... (my robotics knowledge is limited feel free to add more detail).

3. Movement, ball kicking and communication actions .
4. Agent has a (partial) model of the football field and its status.
5. Agent has goals of scoring, or helping other players to score, preventing opposing team scoring, avoiding damage, conserving battery, etc.
6. Agent has battery level, number of goals scored, and damage level performance measures (you can probably think of more).

2 The Sticks Problem

Consider the following puzzle[1]. There are 8 sticks lined up as shown in Fig. 1 (a). By moving exactly 4 sticks, you are to achieve the configuration shown in Fig. 1 (b). Each move consists of picking up a stick, jumping over exactly two other sticks, and then putting it down onto a fourth stick. For example, in Fig.1 (c), stick d could be moved onto stick a or g passing over sticks c and b or e and f, respectively. It could not be moved anywhere else. In Fig. 1 (d) stick d cannot be moved at all. Note that when there is a stick x on top of another stick y, it is illegal to put another stick z on top of y.

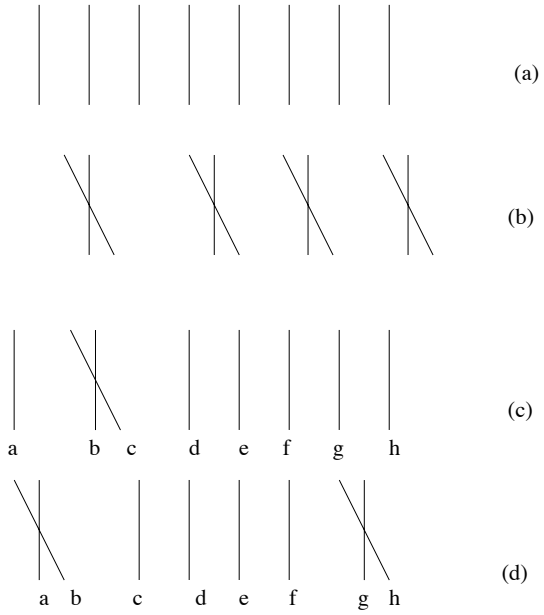

Figure 1: The Sticks Problem

[1]Originally set by John Hallam

The search space for this problem is shown on page 7. States which are essentially identical due to symmetry are not duplicated.

Here are your tasks for this tutorial:

1. Decide on a notation for representing states, which you could use in state space search.
2. Using this notation, show the initial state and the goal state.
3. Give the details of all the operators. For each, describe its preconditions, and give an example (i.e. apply the operator to a state and give the new state).
4. Apply the uninformed search algorithms, depth-first and breadth-first search, to the search tree. Is any one obviously the best in reaching the goal? How many nodes do they expand? How many nodes do they keep in the frontier at one time?

2.1 Tutor's Guide: The Sticks Problem

1. The position could be represented in several ways; the most natural is a list with the numbers 1 and 2, denoting single or overlapping sticks . The main difficulty people have with representation in this problem is that they think the position of each stick must be represented rather than the pattern - they visualise each stick occupying a place which becomes empty when the stick is moved.
2. Initial state: `[1,1,1,1,1,1,1,1]`
   Goal state: `[2,2,2,2]`
3. The operators can be described as rewrite rules. There are three of them (X and Y stands for variables that can march some arbitrary pattern):

   (a) $X1111Y \Rightarrow X211Y$

   (b) $X1111Y \Rightarrow X112Y$

   (c) $X121Y \Rightarrow X22Y$

   The first two move a single stick over two single sticks to the left and to the right, respectively; the last one moves a single stick over a pair of crossed sticks.

   The precondition for each operator is that we can match the pattern on the left of the arrow to some part of the current state representation.
4. Using left to right node expansion, Depth-first search expands 30 nodes, and Breadth-first search 49 nodes (or 70 nodes *without* the tweak described in R&N §3.4.1). So depth-first search gets to the goal first given the way we have ordered the search tree. Point out to the

students that depth-first search is better when there are many terminal states quite deep in the search tree, whereas breadth-first search is better when there are few solutions at a shallow depth.

Depth-first search has a maximum of 11 nodes in its frontier, whereas breadth-first search has a maximum of 32. In general depth-first will have fewer nodes, it depends upon the structure of the search tree.

Just to remind you, the general graph search algorithm is given in Figure 3. The depth-first search algorithm appends successors to the front of the frontier. The breadth-first search algorithm appends successors to the back of the frontier, but also checks if one the child nodes is the goal when expanding them (see Figure 4).

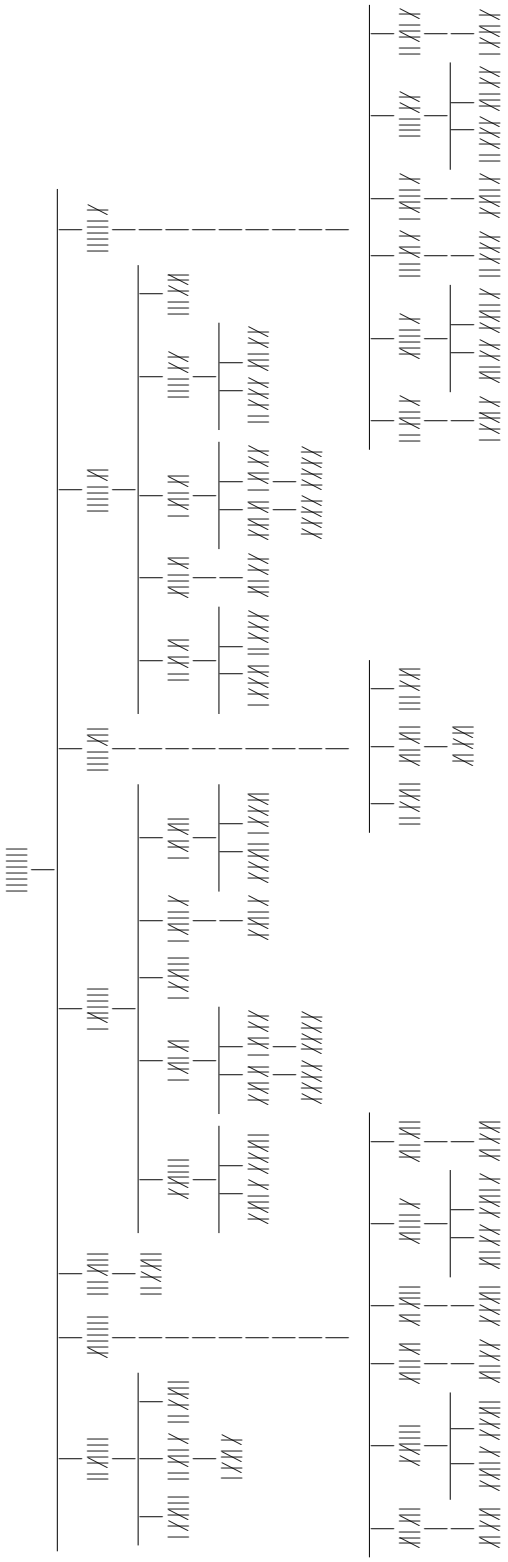

Figure 2: Search tree for the Sticks Problem

```
function GRAPH-SEARCH(problem) returns a solution, or failure
  initialize the frontier using the initial state of problem
  initialize the explored set to be empty
  loop do
    if the frontier is empty then return failure
    choose a leaf node and remove it from the frontier
    if the node contains a goal state then return the corresponding solution
    add the node to the explored set
    expand the chosen node, adding the resulting nodes to the frontier
      only if not in the frontier or explored set
```

Figure 3: The Graph search algorithm

```
function BREADTH-FIRST-SEARCH(problem) returns a solution, or failure
  node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
  if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
  frontier ← a FIFO queue with node as the only element
  explored ← an empty set
  loop do
    if EMPTY?(frontier) then return failure
    node ← POP(frontier)  /* chooses the shallowest node in frontier */
    add node.STATE to explored
    for each action in problem.ACTIONS(node.STATE) do
      child ← CHILD-NODE(problem, node, action)
      if child.STATE is not in explored or frontier then
        if problem.GOAL-TEST(child.STATE) then return SOLUTION(child)
        frontier ← INSERT(child, frontier)
```

Figure 4: The Breadth-First search algorithm