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 \(y\).

The search space for this problem is shown on page 6. 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?

\(^1\)Originally set by John Hallam
1.1 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 2. 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 3).

```
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 2: The Graph search algorithm

2 Adversarial Search

This exercise was taken from R&N Chapter 5.

Consider the two-player game shown in Figure 4

1. Draw the complete game tree, using the following conventions:
   - Write each state as \((S_A, S_B)\) where \(S_A\) and \(S_B\) denote token locations
   - Put each terminal state in square boxes and write its game value in a circle
• Put loop states (states that already appear on the path to the root) in double square boxes. Since it is not clear how to assign values to loop states, annotate each with a “?” in a circle.

2. Now mark each node with its backed-up minimax value (also in a circle). You will have to think of a way to assign values to the loop states.

3. Explain why the standard minimax algorithm would fail on this game tree and briefly sketch how you might fix it, drawing on your answer to item (2) above. Does your modified algorithm give optimal decisions for all games with loops?

Figure 4: The starting position of a simple game. Player A moves first. The two players take turns moving, and each player must move their token to an open adjacent space in either direction. If the opponent occupies an adjacent space, then the player may jump over the opponent to the next available space. For example, if A is on 3 and B is on 2, then A may move back to 1. The game ends when one player reaches the opposite end of the board. If player A reaches space 4 first, then the value of the game to A is +1; if player B reaches space 1 first the value of the game to A is −1.

2.1 Tutor’s Guide: Adversarial Search

1. The game tree, complete with annotations is shown in Figure 5
Figure 5: The game tree for the four-square game in Exercise 6.3. Terminal states are in single boxes, loop states in double boxes. Each state is annotated with its minimax value in a circle.

2. The “?” nodes can be assigned a value of 0, so choosing the loop state will not benefit either player. If there is a win for a player from a loop state then they will choose the moves that lead to the win, rather than end up in the loop state.

3. Standard minimax is depth-first and would go into an infinite loop. It can be fixed by comparing the current state against the stack; and if the state is repeated, then return a “?” value. Propagation of “?” is handled as above. Although it works in this case, it does not always work. For example, it is not clear how to compare “?” with a drawn position: a drawn position is not better or worse for either player, but might at least bring an end to the game, so assigning 0 to “?” states will not work there.
Figure 6: Search tree for the Sticks Problem