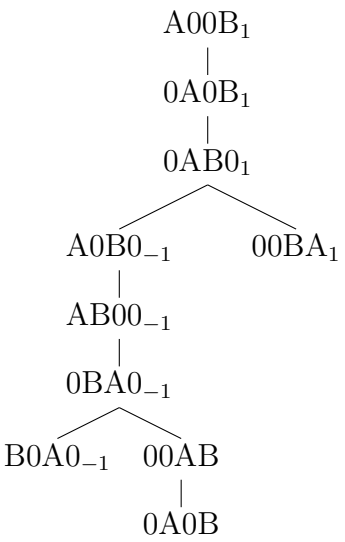
Intelligent Systems (Sheet #4)

Marius Gavrilescu

1. (a) 9! is an upper bound for the number of games, because at the first turn there's 9 possible moves, at the second turn there's 8 possible moves, and so on. Not all of these are reachable, because we stop the game when a player wins. Every reachable game is a node in the search tree. Each of these games is a valid game state.

   (c) We explore nodes in order of the heuristic Eval. The tree below is drawn in that order, and nodes with subscript p are pruned

2. (a)



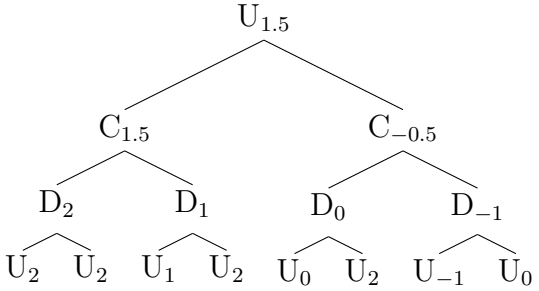
The problem is that the game tree is not a tree – there can be cycles. We can instead consider the depth-first traversal tree of this graph to be the game tree, and not assign any value to leaf nodes that are not terminal states. Then this node is ignored when computing the utility of its parent, and if all children of node X have no value then X also has no value.

(b) Key point: with no jumps, the parity of the distance between A and B will not change after two consecutive moves.

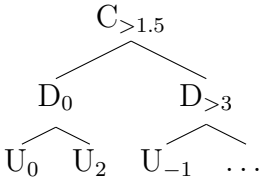
A's winning strategy for even n: keep moving right until A is adjacent to B. This will necessarily happen on B's turn (see key point above), so A will be able to jump over B. Now A has travelled one square further than B, and if it keeps moving to the right it will arrive there before B arrives to the left.

B's winning strategy for odd n: keep moving left until A is adjacent to B. This will necessarily happen on A's turn (as above), so B will be able to jump over A. By the same reasoning as becore B will win.

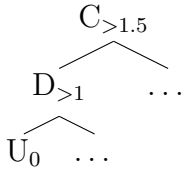
3. (a)



(b) Here we only prune the rightmost leaf. Once we are at the rightmost min-node, we know its value has to be at least 3 to get us utility more than 1.5 for its parent. It is a min-node and we see its left child has value -1, so we know it cannot have a value larger than 1.5 and we stop the search here.



(c) Here we will compute the left chance node to have utility 1.5. Now we are looking for something larger than 1.5 in the second chance node:



As the chance node's value is the mean of its two children, and each of its children is at most 2, we know each child has to be at least 1. Since the first child is a min-node and has a child with value 0, we know it can't be larger than 0 so we stop searching there. There is no point in looking at the right child of the chance node, as we already know we can't get more than 1 in this chance node.

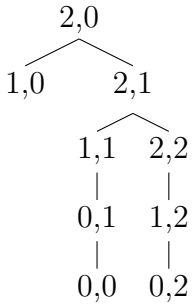
4. We will use a mutable solution map and we keep passing references to it from function to function. The only change to OR-SEARCH is that it receives a sol_map argument that it passes to calls to OR-SEARCH.

```
FUNCTION AND-OR-GRAPH-SEARCH(problem) RETURNS solution/failure
  RETURN OR-SEARCH(problem.INITIAL-STATE, problem, [], CREATE-EMPTY-MAP())

FUNCTION OR-SEARCH(state, problem, path, sol_map) RETURNS solution/failure
  IF problem.GOAL-TEST(state) THEN RETURN the empty plan
  IF state IS ON path THEN RETURN failure
  IF NOT (sol_map.contains(state)) THEN
    FOR EACH action IN problem.ACTIONS(state) do
      plan <-
        AND-SEARCH(RESULTS(state, action), problem, [state | path], sol_map)
      IF plan <> failure THEN
        sol_map.insert(state, [action | plan])
        BREAK
    sol_map.insert(state, failure)
  RETURN sol_map.get(state)
```

This can be further improved by also remembering results of AND-SEARCH in a similar manner.

6.



0,2 is the goal node, 2,0 is the start node.

The question is slightly suspicious: this is not an AND-OR search, because this is not a two-player game. The single agent can always choose to move in whatever direction he wants, so we only care if there exists one path to the destination or not.

5. The problem is unsolvable because for any sequence of actions there exists an initial state and set of random event results after which we are not in a goal state.

   To simplify the problem, let's consider only SuckLeft and SuckRight actions instead of Left, Right, Suck actions. Let there be a sequence of SuckLeft and SuckRight actions. Now let's construct a situation where this sequence fails to clean the rooms. First, we will only consider the situations where if we suck a dirty room we only clean that room and not the adjacent room.

   If the number of SuckLeft actions is zero, then we will take an initial state where the left room is dirty. After this sequence of actions we will still have the left room dirty.

   If the number of SuckLeft actions is nonzero, then we will take an initial state where the left room is dirty, and consider the case where every SuckLeft action does nothing to the already-clean room save for the last one, which deposits dirt in the room. After this sequence of actions we will have the left room dirty.

   We have shown that for every sequence of moves, there is an initial state and choice of random action results that results in a non-goal state. Therefore there is no solution to the problem if we know nothing about the initial state.