National University of Singapore
School of Computing
CS2109S: Introduction to AI and Machine Learning
Semester I, 2023/2024

Tutorial 3
Adversarial Search and Local Search

These questions will be discussed during the tutorial session. Please be prepared to answer them.

Summary of Key Concepts

In this tutorial, we will discuss and explore the following key learning points/lessons from Lecture:

1. Adversarial Search
   (a) Minimax Algorithm
   (b) α-β Pruning
2. Local Search

Problems

1. Consider the game tree for Tic-Tac-Toe shown in Figure 1. The Tic-Tac-Toe search space can actually be reduced by means of symmetry. This is done by eliminating those states which become identical with an earlier state after a symmetry operation (e.g. rotation). Figure 2 shows a reduced state space for the first three levels with the player making the first move using "x" and the opponent making the next move with "o". Assume that the following heuristic evaluation function is used at each leaf node n:

   $$Eval(n) = P(n) - O(n)$$

   where $P(n)$ is the number of possible winning lines for the player while $O(n)$ is the number of possible winning lines for the opponent. A possible winning line for the player is a line (horizontal, vertical or diagonal) that either contains nothing or "x". For the opponent, it is either nothing or "o" in the line. Thus, for the leftmost leaf node in Figure 2, $Eval(n) = 6 - 5 = 1$.

   (a) Use the minimax algorithm to determine the first move of the player, searching 2-ply deep search space shown in Figure 2.

   The player will place his "x" in the centre as shown in Figure 2.

   (b) Assume that the "x" player places his first move in the centre and the opponent has responded with an "o". Compute the evaluation function for each of the filled leaf nodes and determine the second move of the "x" player in Figure 3 (searching 2-ply deep).

   The player will place his "x" in either the top left or the bottom left as shown in Figure 3.

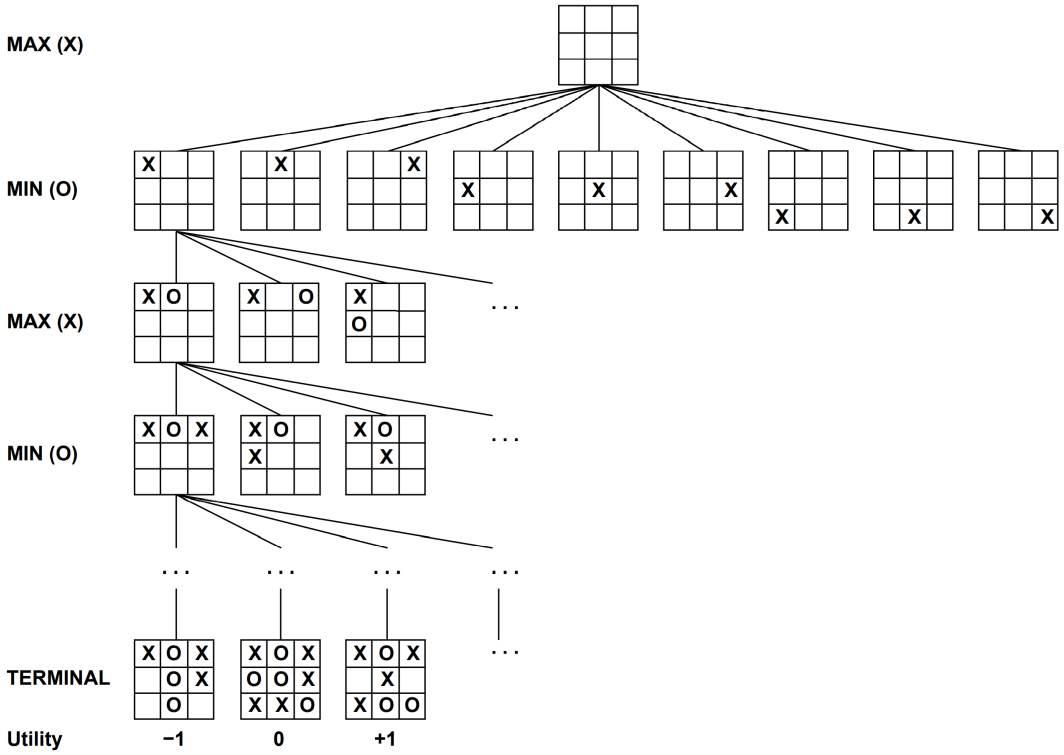


Figure 1: Search space for Tic-Tac-Toe.

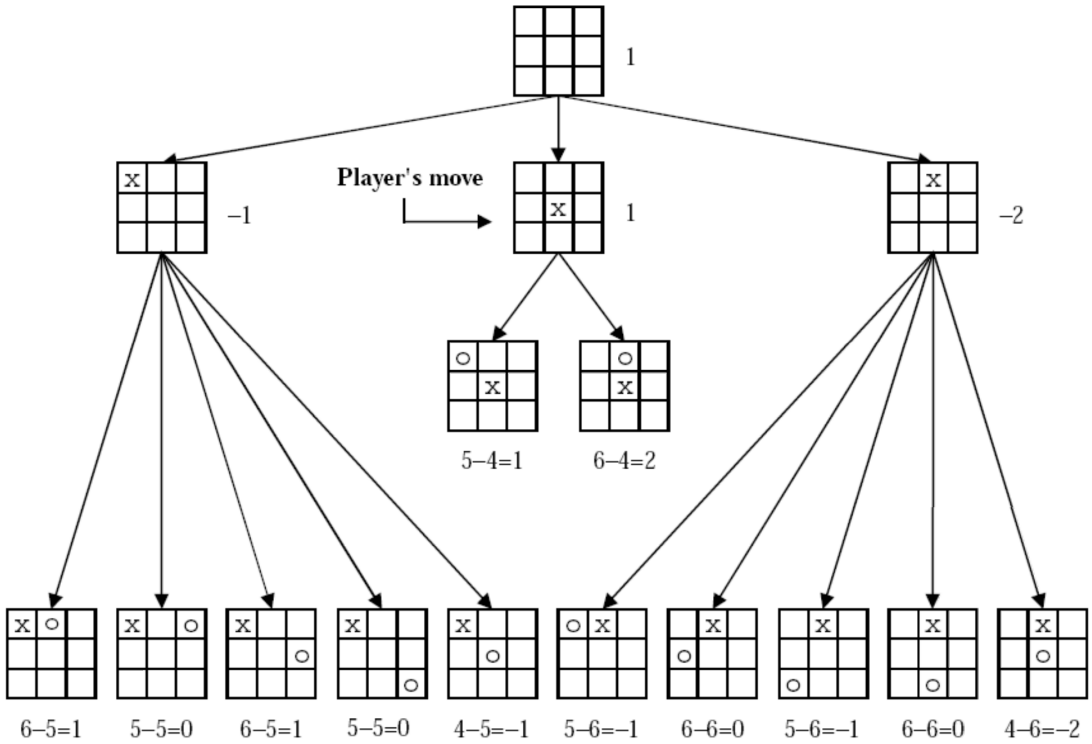


Figure 2: First move 2-ply deep search space solution

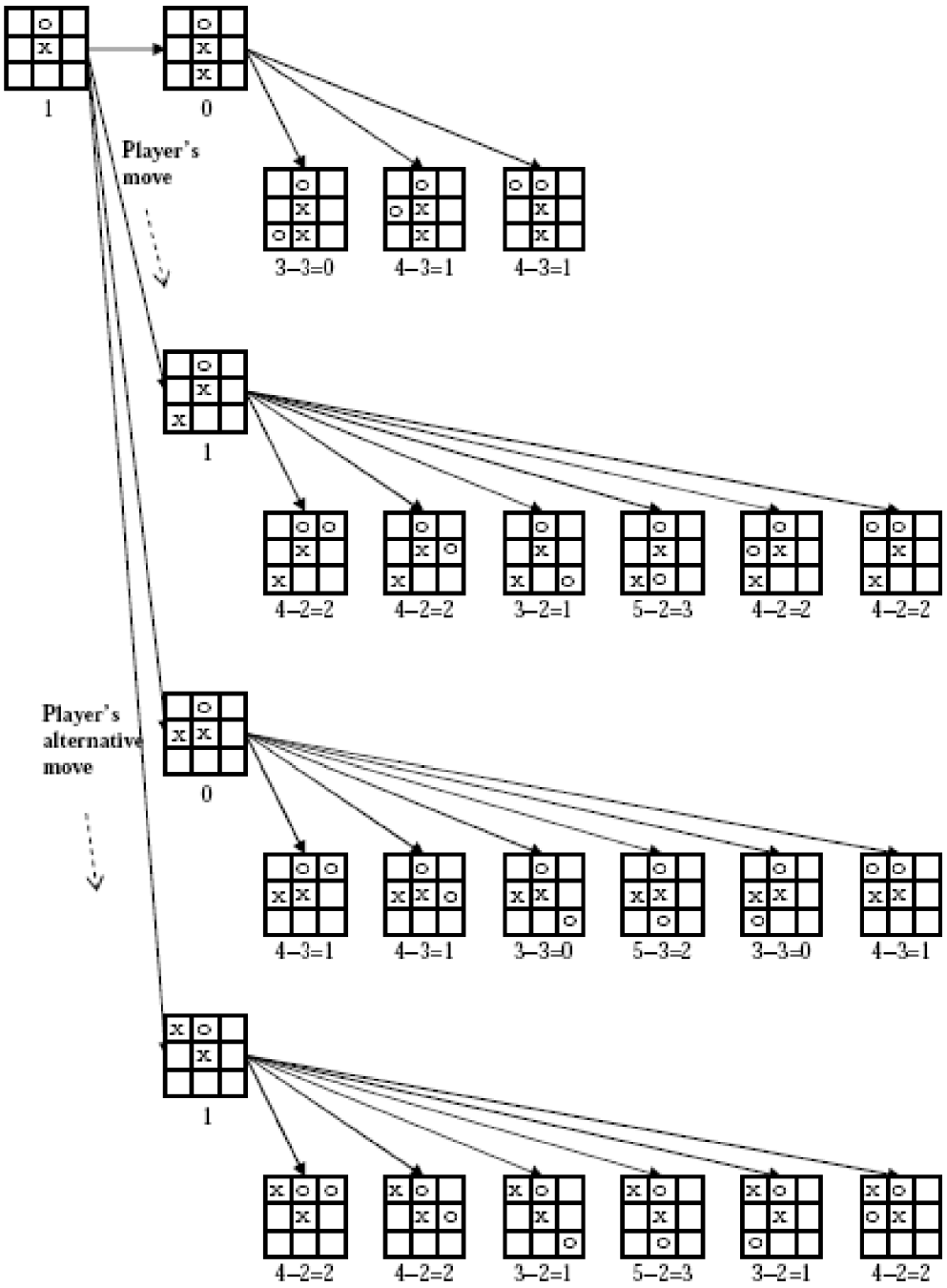


Figure 3: Second move 2-ply deep search space solution

2. Consider the minimax search tree shown in Figure 4. In the figure, black nodes are controlled by the MAX player, and white nodes are controlled by the MIN player. Payments (terminal nodes) are squares; the number within denotes the amount that the MIN player pays to the MAX player (an amount of 0 means that MIN pays nothing to MAX). Naturally, MAX wants to maximize the amount they receive and MIN wants to minimize the amount they pay.

   Consider the minimax tree given in Figure 4.

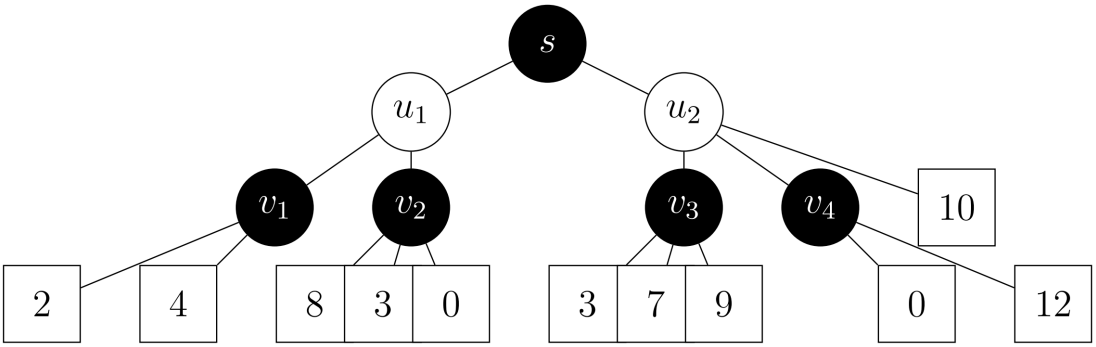


Figure 4: Minimax search tree

Suppose that we use the α-β Pruning algorithm reproduced in Figure 5.

```
function ALPHA-BETA-SEARCH(game, state) returns an action
  player ← game.TO-MOVE(state)
  value, move ← MAX-VALUE(game, state, −∞, +∞)
  return move

function MAX-VALUE(game, state, α, β) returns a (utility, move) pair
  if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
  v ← −∞
  for each a in game.ACTIONS(state) do
    v2, a2 ← MIN-VALUE(game, game.RESULT(state, a), α, β)
    if v2 > v then
      v, move ← v2, a
      α ← MAX(α, v)
    if v ≥ β then return v, move
  return v, move

function MIN-VALUE(game, state, α, β) returns a (utility, move) pair
  if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
  v ← +∞
  for each a in game.ACTIONS(state) do
    v2, a2 ← MAX-VALUE(game, game.RESULT(state, a), α, β)
    if v2 < v then
      v, move ← v2, a
      β ← MIN(β, v)
    if v ≤ α then return v, move
  return v, move
```

Figure 5: α-β Pruning algorithm

(a) **Assume that we iterate over nodes from right to left**; mark with an 'X' all **arcs** that are pruned by α-β pruning, if any.

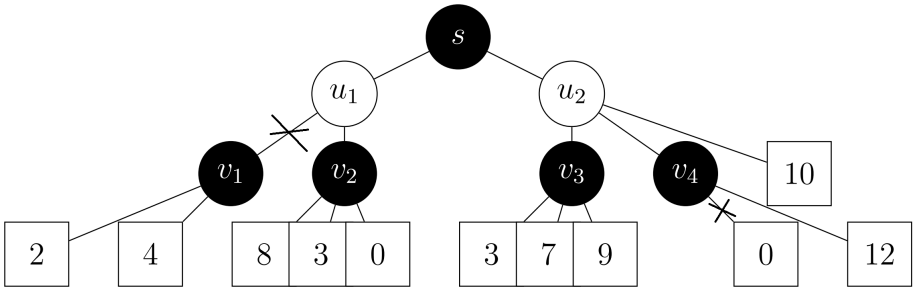


Figure 6: Minimax search tree pruned from right to left

At the start, the MAX player explores u_2 first. MIN player explores the terminal node 10 and sets $\beta(u_2) = 10$. Next, MIN player explores v_4, which leads to MAX player exploring the terminal node 12, setting $\alpha(v_4) = 12$. Since $12 \geq \beta(u_2) = 10$, there is no point exploring the rest of this subtree (intuition: since MIN player already has a better move which guarantees a score of 10 and for this subtree the MAX player will at least get a score of 12 regardless of what comes after) so the arc that leads to terminal node 0 is pruned.

Next, MIN player explores v_3, which leads to MAX player exploring the terminal node 9 setting $\alpha(v_3) = 9$. Since $9 < \beta(u_2) = 10$, we continue exploring this subtree (this move could possibly be better than the other moves we have already seen). None of the rest of the nodes in this subtree are more than $\beta(u_2)$, so none of them are pruned. With a new-found minimum of 9 from the subtree of v_3, we update $\beta(u_2) = 9$.

Now that we are done with u_2, we update $\alpha(s) = 9$ and then explore u_1, which leads to the MIN player exploring v_2. We explore the whole subtree and set $v(v_2) = 8$, and then $\beta(u_1) = 8$. Since $8 \leq \alpha(s) = 9$, there is no point exploring the rest of this subtree (intuition: since MAX player already has a better move which guarantees a score of 9 and for this subtree the MIN player will get a score of at most 8 regardless of what comes after) so the arc that leads to v_1 is pruned.

(b) Show that the pruning is different when we instead iterate over nodes from **left to right**. Your answer should clearly indicate all nodes that are pruned under the new traversal ordering.

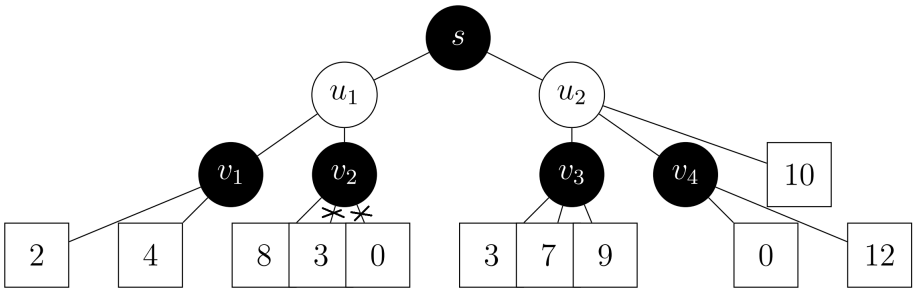
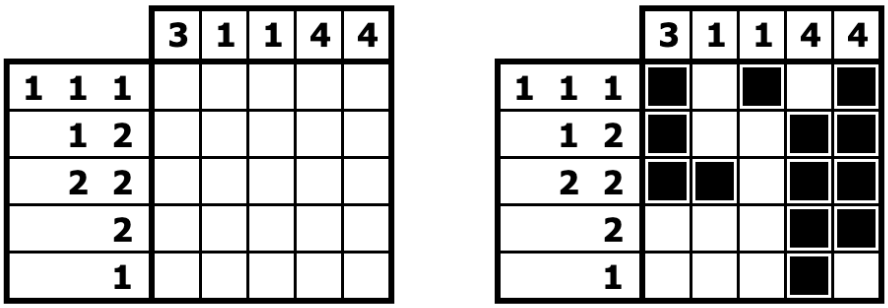


Figure 7: Minimax search tree pruned from left to right

Similarly, tracing the algorithm will result in Figure 7.

3. Nonogram, also known as *Paint by Numbers*, is a logic puzzle in which cells in a grid must be colored or left blank according to numbers at the side of the grid. Usually, the puzzles are colored either in black or white, and the result reveals a hidden pixel-art like picture.

   Nonograms can come in different grid sizes. In this example, we will take a look at a 5×5 Nonogram. The game starts with an empty 5×5 grid. In the row and column headings, a series of numbers indicate the configuration of colored cells in that particular row and column. For example, on the fourth row, the "2" in the row header indicates that there must be 2 *consecutive* black cells in that row. In a more complicated case, such as in the second row, "1 2" indicates that there must be 1 black cell, followed by at least one blank cell, then 2 consecutive black cells. Similarly, in the first row, there must be 1 black cell, followed by at least one blank cell, another black cell, then at least one blank cell, and finally the last black cell. The rules apply similarly column-wise. For example, in the last column, there must be 4 consecutive black cells.



(a) Initial Nonogram (b) Solved Nonogram

Figure 8: Nonogram

   To familarise yourself with the game, you may wish to play on this site.

   Given an empty $n \times n$ Nonogram with specified row and column color configurations, the challenging part of this game is to find out how you can color the cells in a way that satisfies **all** the constraints of the rows and columns.

   (a) Having learnt both informed search and local search, you think that local search is more suitable for this problem. Give 2 possible reasons why informed search might be a bad idea.

   First, the search space is very huge. A loose upper bound on the size of the search space is 2^{n^2}, since there are n^2 cells in an $n \times n$ grid, and each cell can be either colored or not colored. Thus, using informed search could take a very long time. Second, we are not interested in obtaining the solution path, but rather, reaching the goal state.

   Note that in this problem, we assumed that the provided $n \times n$ Nonogram is solvable. In general, local search can be helpful if the problem configuration is not necessarily solvable, and what we actually want is a configuration that minimizes the number of conflict violations.

   You decide to formulate the solution as a local search problem, which involves 3 steps:

i. Define a candidate solution

ii. Define a transition function to generate new candidate solutions

iii. Define a heuristic to evaluate the "goodness" of a candidate solution.

All 3 need to be designed in tandem because they have an impact on each other.

(b) Based on the description of the problem above, propose a state representation for the problem.

We can use an $n \times n$ boolean matrix, where each element is either true (if the corresponding cell is colored) or false (if the corresponding cell is not colored).

(c) What are the initial and goal states for the problem under your proposed representation?

The initial state is an $n \times n$ boolean matrix with m entries set to true, while the rest of the entries are set to false. Note that m can be obtained by summing the numbers in the row header (or in the column header). For example, in the provided example, $m = 3+1+1+4+4 = 13$, or, $m = (1+1+1)+(1+2)+(2+2)+2+1 = 13$.

To choose the positions of the m entries that are set to true, we can do so randomly, while ensuring that the entries adhere to the row constraints.

The goal state is defined as the set of representations that returns true for the goal test, which is only when the given $n \times n$ boolean matrix fulfils all the row and column constraints of the puzzle. Notice that unlike the Missionaries and Cannibals problem, or the pitchers problem, we cannot explicitly define the goal state. If not, the problem is already solved.

(d) Define a reasonable transition function to generate new candidate solutions.

Given the current candidate, we can pick a random row and generate the list of neighbours with the corresponding row permuted. Then, we can move to the neighbour with the lowest cost.

(e) Define a reasonable heuristic function to evaluate the "goodness" of a candidate solution. Explain how this heuristic can also be used as a goal test to determine that we have a solution to the problem.

A natural heuristic is the number of instances where the constraints on the column configurations are violated and we want to minimize this number. If there are no constraint violations, then we have a valid solution and the problem is solved.

Notice that we do not need to count the number of instances where the constraints on the row configurations are violated, because by design of how we generate candidate solutions, we have already ensured the adherance of the entries to the row constraints.

(f) Local search is susceptible to local minimas. Describe how you can modify your solution to combat this.

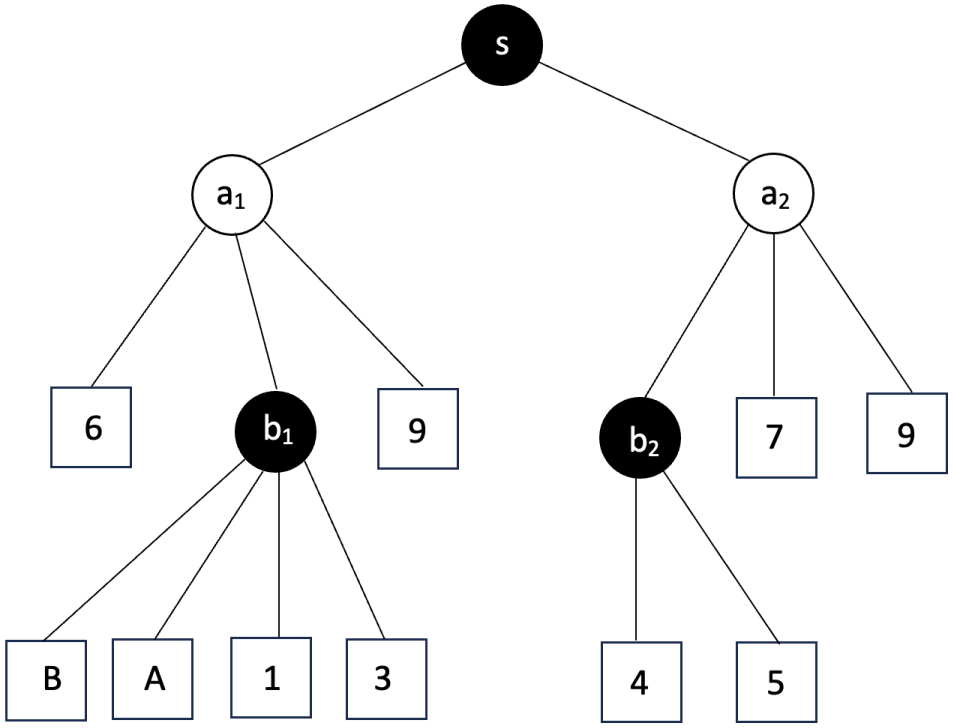
There are many possible ways in which we can improve the completeness of the local search algorithm. One way is to introduce random restarts by repeating local search from a random initial state. We can also use simulated annealing

search to accept a possibly "bad" state with a probability that decays over time or beam search to perform k hill-climbing searches in parallel.

Note: There are many possible solutions for parts (b), (c), and (d), but they need to be coordinated.

4. (Additional) In order for node B to NOT be pruned, what values can node A take on?

   NOTE: Assume that we iterate over nodes from right to left. Black node represents MAX player, white node represents MIN player.



   A less than or equals to 8.