CS3243: Introduction to Artificial Intelligence **Fall 2020**

Lecture 5: September 9

Lecturers: Prof. Kuldeep S. Meel *Scribe: Ang Zheng Yong*

5.1 Recap of Week 4

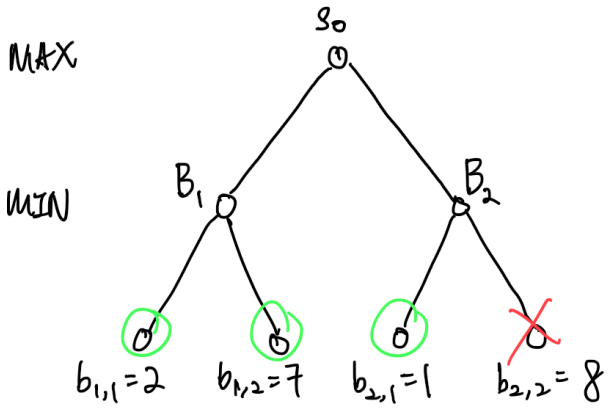

Figure 5.1: A game of bags and balls

We evaluated the balls from left to right in an orderly manner.

1. When the first ball $b_{1,1}$ is evaluated, player *MIN* knows that he is able to get a value of ≤ 2.
2. When the second ball $b_{1,2}$ is evaluated, player *MIN* would definitely select $b_{1,1} = 2$, and here player *MAX* knows that he is able to get a value of ≥ 2.
3. When the third ball $b_{2,1}$ is evaluated, player *MIN* knows that he is able to get a value of ≤ 1. Here, player *MAX* knows that it definitely does not want to be taking the right branch, since the left branch would give a greater utility ($2 \geq 1$).
    - Hence, we can ignore the last ball, and from the perspective of player *MAX*, we can treat it as if the right branch did not exist.

5.2 Alpha-Beta Pruning (Continued)

We begin this section by discussing the exercise provided last week.

5.2.1 Sample game tree

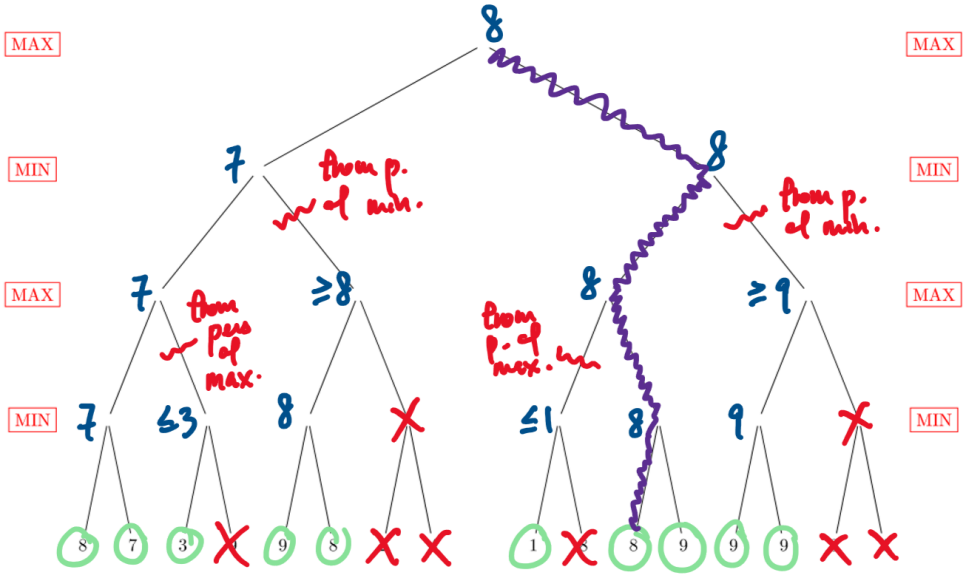

Figure 5.2: A game tree with 16 terminal nodes

Similar to the game tree for bags and balls, we will be evaluating the nodes of this tree from left to right, in an orderly fashion.

1. When the first leaf node (with *Utility* $= 8$) is evaluated, *MIN* knows that it is able to get a value of ≤ 8. After evaluating the second leaf node (with *Utility* $= 7$), *MIN* will select 7 instead.
2. When the third leaf node (with *Utility* $= 3$) is evaluated, *MIN* is able to get a value of ≤ 3 from this branch, and *MAX* knows that there is no point picking from this branch. Hence, we do not need to check the value of the fourth leaf node, and we will treat it as if this branch never existed.
3. Moving up the tree, *MIN* on the layer 2 knows that it is able to secure a value of ≤ 7. Following the reasoning in (2), we will be able to prune nodes 7 and 8 from our tree as well.
4. After repeated exploring and pruning, we will be able to end up with the game tree as shown above. The path marked in purple indicates our **strategy**, that is, it specifies *MAX*'s move in the initial state, followed by *MIN*'s move in the following state, followed by *MAX*'s subsequent move, etc.
    - More generally, a strategy is a function that maps from a set of states to a set of actions, i.e. $f:$ states $\rightarrow$ actions.

5.2.2 Minimax algorithm with alpha-beta pruning

Recall our minimax algorithm prior to introducing alpha-beta pruning:

Algorithm 1 Minimax

```
1: procedure MaxVal(s)
2:     if Terminal(s) == True then
3:         return Utility(s)
4:     end if
5:     v ← −∞
6:     for each a in Actions(s) do
7:         result ← Result(s, a)
8:         v ← max(v, MinVal(result))
9:     end for
10:    return v
11: end procedure
```

So, what is the meaning of α and β in the context of alpha-beta pruning?

- α: best value from the perspective of *MAX*.
- β: best value from the perspective of *MIN*.

By incorporating α and β into our minimax algorithm, we will be able to update our minimax algorithm (as shown on the next page).

We can then call the algorithm on our game tree as shown in Figure 5.2 with $MAXVAL(\text{root}, -\infty, \infty)$.

- The best value for *MAX* is initialized to $-\infty$.
- The best value for *MIN* is initialized to ∞.

Algorithm 2 Minimax with $\alpha - \beta$ pruning

```
1: procedure MaxVal(s, α, β)
2:     if Terminal(s) == True then
3:         return Utility(s)
4:     end if
5:     v ← −∞
6:     for each a in Actions(s) do
7:         result ← Result(s, a)
8:         v ← max(v, MinVal(result, α, β))
9:         if v ≥ β then                          ▷ condition for pruning
10:            return v
11:        end if
12:        α ← max(α, v)
13:    end for
14:    return v
15: end procedure
16:
17: procedure MinVal(s, α, β)
18:    if Terminal(s) == True then
19:        return Utility(s)
20:    end if
21:    v ← +∞
22:    for each a in Actions(s) do
23:        result ← Result(s, a)
24:        v ← min(v, MaxVal(result, α, β))
25:        if v ≤ α then                          ▷ condition for pruning
26:            return v
27:        end if
28:        β ← min(β, v)
29:    end for
30:    return v
31: end procedure
```

5.2.3 Space complexity of minimax

Considering a game tree with depth d,

- The original minimax algorithm would require a space of $O(b^d)$.
- Minimax with alpha-beta pruning would require a space of $O(b^{d/2})$.
  - This space consumption depends heavily on the order in which the nodes are visited.
  - However, under most circumstances, we would be able to get a space comsumption of just $O(b^{d/2})$.

Even though we have managed to reduce the branching factor of our tree from b to $\sqrt{b}$, we would still need to go through $35^{100/2} = 35^{50}$ distinct states for a simple game of chess. Is there any way for us to do better than this?

5.2.4 Improvements on minimax

Idea: At every iteration of our algorithm, we only explore nodes up to a depth l instead of exploring the entire tree (which has depth d). We will also need to address the following issues:

1. How do we pick l?
   - We start with $l = 1$, then $l = 2$, and so on.
   - At every depth, we will keep track of the best value known as far.
     - If we have more time to perform additional computations, we can then evaluate deeper levels.
   - This is also known as **iterative deepening**.
2. How do we know the cost of going down this path?
   - Only terminal nodes have a utility associated with them; the utility of intermediate nodes was previously derived from the utility of their child nodes (e.g. refer to Figure 5.1 and 5.2).
   - Hence, we use an **evaluation function** as a heuristic for guiding our branch selection.
     - An evaluation function is defined as: $f_{\text{eval}}(s) = w_1 f_1(s) + w_2 f_2(s) + \ldots + w_n f_n(s) = \sum_{i=1}^{n} w_i f_i(s)$, where:
       - f_i is a feature of the position, and
       - w_i is a weight associated with f_i.
         - w_i can be calculated through learning, or defined by humans.
       - For example, in chess, we *may* define $f_1(s)$ to be the number of bishops present, $f_2(s)$ to be the number of knights present, etc.
     - In our minimax algorithm, we will substitute $\mathit{Utility}(s)$ with $f_{\text{eval}}(s)$.

5.3 Local Search

Let us now head back to problems involving single agents. Particularly, we will be examining the n-queens puzzle.

5.3.1 n-queens puzzle

The objective of this problem is to arrange n queens on an n-by-n chessboard, such that the n queens won't attack each other. We can model our problem as a local search problem with:

1. State s.
2. $N(s)$: the neighbours of state s.
3. $Val(s)$: the value associated with state s.
   - As far as possible, $Val(s)$ should reflect the notion of "quality".
   - If s_g is a goal state, then $Val(s_g) = 0$.

For the n-queens puzzle, we can define $Val(s)$ to be the number of pairs of queens that are attacking each other in state s.

5.3.2 Hill climbing algorithm

After modelling the puzzle as a local search problem, how can we solve the problem? One way would be to use the hill climbing algorithm:

Algorithm 3 Single iteration of hill climbing

```
1:  procedure HillClimbing(s)
2:      minVal ← Val(s)
3:      minState ← ∅
4:      for u ∈ N(s) do
5:          if Val(u) < minVal then
6:              minState = u
7:              minVal = Val(u)
8:          end if
9:      end for
10:     return minState
11: end procedure
```

To get from the goal state from the initial state, we can recursively call $HILLCLIMBING(s)$ on *minState*, and terminate when *minState* is the goal state.

However, the hill climbing algorithm may not always lead to the goal state (it may get stuck in some local optima or a plateau, where neighbouring states have equally or less optimal $Val(s)$ than the current state). One example of such as state in shown below:

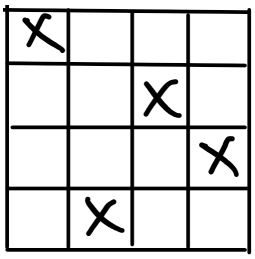

Figure 5.3: A state in the 4-queens puzzle, where "×" denotes the locations of the queens

In the state above, it can be observed that none of its neighbouring states would give a $Val(s)$ which is better than the current $Val(s)$, and so we are stuck in a plateau. How do we deal with cases such as this?

- Idea: Our algorithm should be able to permit minor mistakes once in a while. Then, we might be able to get to the goal state (i.e. global optima).
  - For instance, in the figure above, even though moving the queen in the upper-right corner down by 1 step would result in $Val(s)$ remaining as 1, it would actually lead us closer to the goal state.

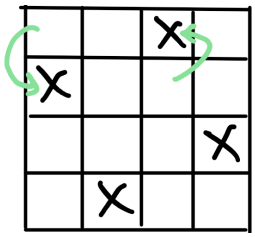

Figure 5.4: A goal state for the 4-queens puzzle

However, our algorithm shouldn't tolerate mistakes that are too bad (e.g. a step that results in $Val(s)$ increasing by more than 2). How could we design an algorithm which accepts some mistakes which are not too bad, while rejecting others?

- Suggestion: After the for loop in Algorithm 3, we can loop through all the neighbours of s for a second time, and return all the states with $Val(minState) + \alpha$ instead of just $minState$, where α is a constant that represents the amount of increase in $Val(s)$ that is still acceptable. This would result in a longer runtime of our algorithm, but prevent our algorithm from getting stuck.