CSCE625: Introduction to Artificial Intelligence

Programming Assignment 2:

Search for Games

Wei Qiao 522006310

In this programming report, we consider a simple game played between two people. A pile of n matches is placed between the two players. Each player takes turns removing some matches from the pile, in an alternating fashion. Players are permitted remove one, two, or three matches in each turn. The player who is forced to take the last match loses.

The work I do for this programming assignment is depicted one by one as follows:

1. Write a program that generates the game tree for this game for each of $n \in \{7,15,21\}$ via the minimax algorithm.

The minimax algorithm computes the minimax decision from the current state. It uses a simple recursive computation of the minimax values of each successor state, directly implementing the defining equations. The recursion proceeds all the way down to the leaves of the tree, and then the minimax values are backed up through the tree as the recursion unwinds. At each

The algorithm is depicted as:

```
Algorithm:  Minimax
Function: minimax(node, depth, maximizingPlayer)
      if depth = 0 or node is a terminal node
              return the heuristic value of node
          if maximizingPlayer
              bestValue := -∞
              for each child of node
                  val := minimax(child, depth - 1, FALSE)
                  bestValue := max(bestValue, val)
              return bestValue
          else
              bestValue := +∞
              for each child of node
                  val := minimax(child, depth - 1, TRUE)
                  bestValue := min(bestValue, val)
              return bestValue
```

In the above algorithm, we choose 1 or 0 as the heuristic value of the terminate node where 1 means the max player wins the game while 0 means the opponent, i.e., the min player, wins the game. By using the above algorithm, we could finally obtain the value of each node in the game tree. The generalized game tree with utility is shown in Fig.1.. Here, we just present the game tree under n=7 since when n is 15 or 21, the game tree is too large to be presented. But as we know, the game tree of n=15 or n=21 is similar to the one of n=7. Hence, it's enough to just present the game tree of n=7.

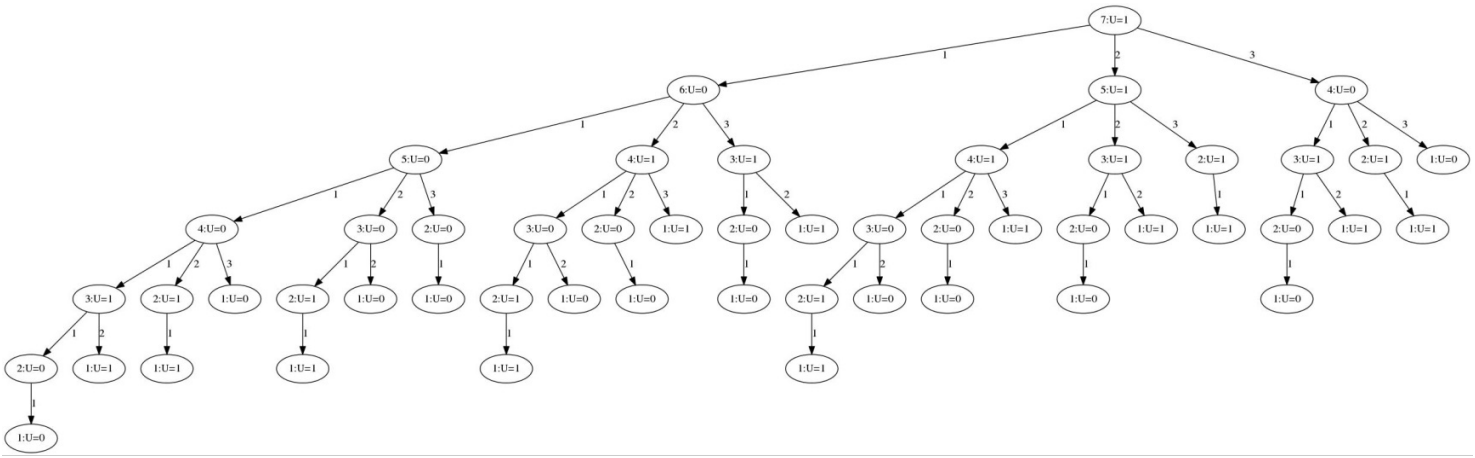


Fig.1. The game tree generated using the minimax algorithm for n = 7

In this part, minimax algorithm is realized for generating the game tree which is shown in Fig.1. for n = 7. Each number before "U" in the ellipse means the current number of matches remaining. The number after "U=" means the corresponding utility under the current state. The number on the edge means the number of matches picked for playing the game. What we need to notice here is that the utility we use is 1 or 0 where 1 means the player will win the game while one means the opponent will win the game. The tree is generated by choosing the maximum value when

2. Extend your code to use alpha-beta pruning. Build those trees and analyze the games:

In the 1st question, we could know that as the depth goes large, the number of nodes in the game tree which is generated by minmax algorithm increases exponentially. That's a disaster of computing. But in fact, some of the computing cost could be eliminated without affecting the result and the decision of the game. The strategy we could use is alpha-beta pruning. Unfortunately, we can't eliminate the exponent, but it turns out we can effectively cut it in half. The trick is that it is possible to compute the correct minimax decision without looking at every node in the game tree. That is, we can use the idea of pruning to eliminate large parts of the tree from consideration. When applied to a standard ALPHA–BETA PRUNING minimax tree, it returns the same move as minimax would, but prunes away branches that cannot possibly

influence the final decision. In our case, since the heuristic value of the leaf node is only 1 or 0, which means α and β could only be 1 or 0, we could further simplify the algorithm.

The algorithm is shown as follow:

```
Algorithm:  alpha-beta pruning
Function: alphabeta(node, depth, maximizingPlayer)
      if depth = 0 or node is a terminal node
                return the heuristic value of node
            if maximizingPlayer
                for each child of node
                    α := alphabeta(child, depth - 1, FALSE)
                    if α==1  return α
                return 0
            else
                for each child of node
                    β := alphabeta(child, depth - 1, TRUE)
                    if β==0  return β
                return 1
```

After exploiting this algorithm, we could finally obtain the pruned tree we want as shown in Fig.2.. As the same reason, we present the pruned game tree of n = 7 here.

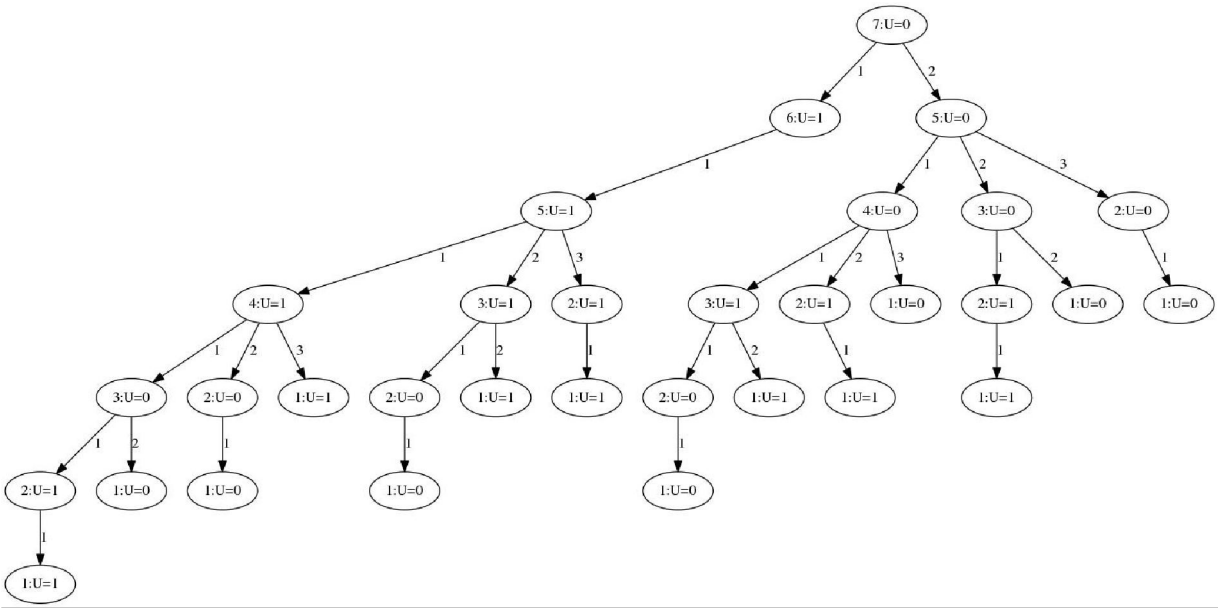


Fig.2. The game tree pruned using the alpha-beta pruning algorithm for n = 7

Now, we are going to answer the following several questions:

(a) What are the relative sizes of the trees?

Alpha-beta pruning can highly reduce the number of nodes in a game tree. We now give some data to depict this fact.

Table 1: the numbers of nodes in the game trees generated using different algorithm for different N

N	Minmax algorithm (the number of nodes)	Alpha-beta pruning (the number of nodes)
21	266079	30215
15	6872	1600
7	52	32

From the above table, we could see that alpha-beta pruning could highly reduce the number of nodes corresponding to minmax algorithm. Especially when N is large, the relative size of the game tree generated by alpha-beta pruning is very small compared to the game tree generated by minmax algorithm. That means alpha-beta pruning could save us a lot of computing cost.

(b) Does opting to go first or second play a role in the outcome of the game? What is the value of the game?

The answer of this question is "Yes". Opting to go first or second play an important role in the outcome of the game. The values of the game corresponding to different N are given as follow:

N=21 The value is 0
N=15 The value is 1
N=7 The value is 0

On the contrary, if the opponent goes first, the result will be the opposite since the decision strategy is totally the same no matter what the player is. Hence, under a situation that the player who goes first wins the game, if he goes the second, he will lose the game.

(c) How well will the optimal player fair in a play-off against a random player (i.e., one who chooses from amongst the valid actions with uniform probability)?

In this question, we could say that in the play-off where an optimal player VS a random play, the optimal nearly always win the game.

When N is large (>5 is enough), the optimal strategy we could use is that keep the number of matches is 4n+1,where n =0,1,2…, for the random opponent. Using this strategy we could nearly always win the game. Extremely, when we meet 4n+1 matches, since the opponent is a random player who chooses the valid actions with uniform probability, we could know that the opponent can not always keep 4n+1 for us.

When N (the initial number of the matches) is small, we can analyze the probability to win the game.

N=5, the random player picks first: the probability that the optimal player wins is 1 since the optimal player could always keep 1 match for the random player.

N=5, the optimal player picks first: the optimal player will pick one match and the probability to win is 2/3 which is the same with the probability that when N=5, the random player picks first.

N=4, the random player picks first: the probability that the optimal player wins is 2/3.

N=4, the optimal player picks first: the probability that the optimal player wins is 1.

N=3, the random player picks first: the probability that the optimal player wins is 2/3.

N=3, the optimal player picks first: the probability that the optimal player wins is 1.

N=2, the random player picks first: the probability that the optimal player wins is 1/2.

N=2, the optimal player picks first: the probability that the optimal player wins is 1.

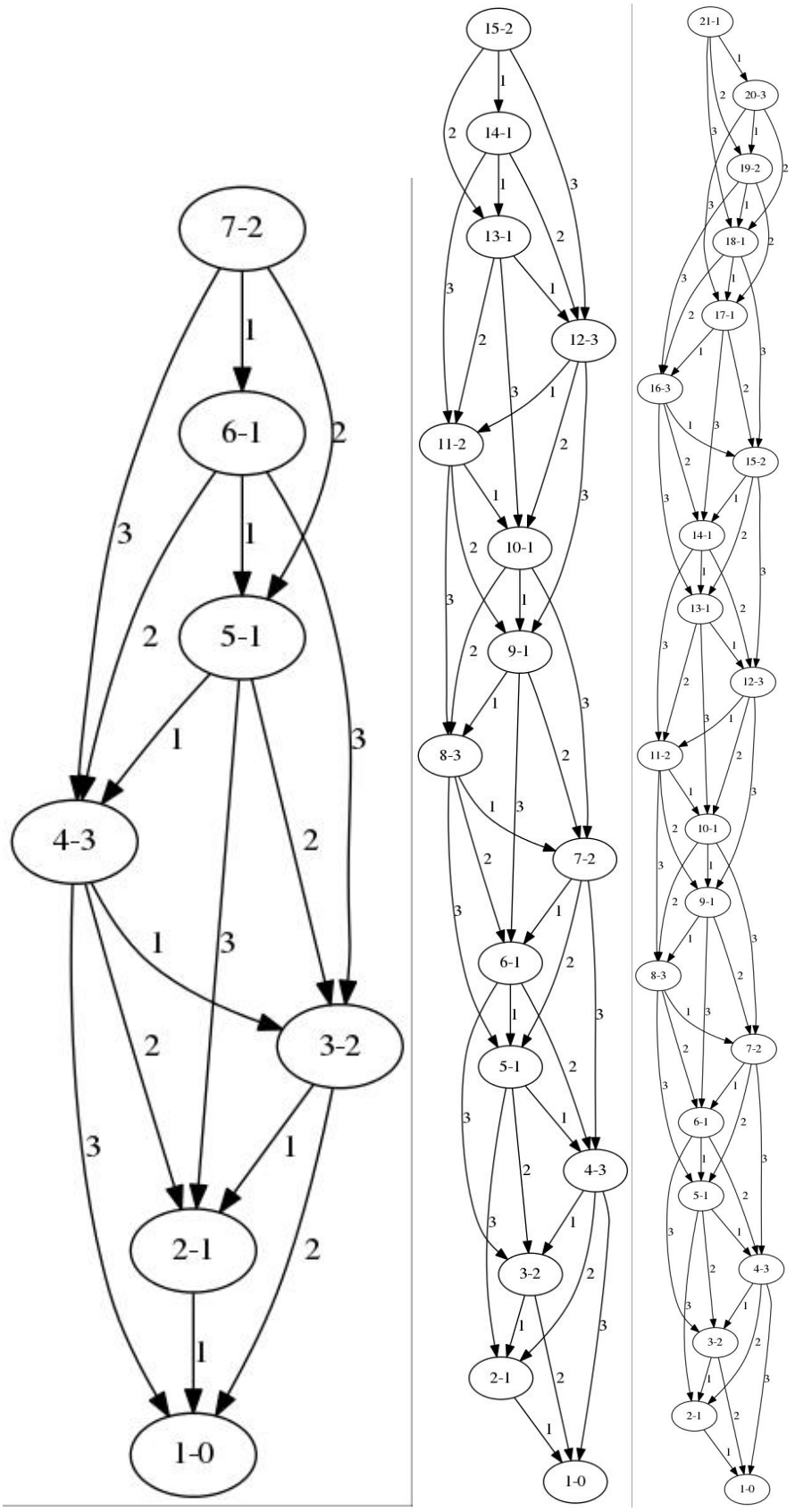
When N is larger than 5, we could know that if the random player wants to win, he should always keep 4n+1 for the optimal player. It's a very small probability (which can be ignored). Hence, we could know that in the play-off where an optimal player VS a random play, the optimal nearly always win the game.

3. Using the results from above synthesize a Moore machine for each of the games (i.e., for the three values of n and whether you go first or second). A natural way to do this is to think about an input alphabet $\sum = \{1,2,3\}$ representing the number of matches taken by the opponent and an output alphabet $\Lambda = \{1,2,3\}$ representing the number of matches you choose to take. Compress the information in the game by generating as small a Moore machine as you can.

For each game tree depicted above, we could generate a Moore machine which includes the change of state and the output corresponding to the given state.

The Moore machines corresponding to N=7, 15, 21 are shown in the next page. The left Moore machines is of N=7; the middle one is of N=15; the right one is of N=21. The number before "-" in the ellipse means the state which is also the number of matches. The number after "-" is the corresponding output given the current state. The number on the edge is the input which brings us the change of the state.

Compared to the original game tree, we could find that the Moore machine is much easier to present the game decision strategy. For an initial number N of matches, there are only N states in the corresponding Moore machine. But the number of the nodes in the game tree is very high.


7-2
6-1
5-1
4-3
3-2
2-1
1-0
15-2
14-1
13-1
12-3
11-2
10-1
9-1
8-3
7-2
6-1
5-1
4-3
3-2
2-1
1-0
21-1
20-3
19-2
18-1
17-1
16-3
15-2
14-1
13-1
12-3
11-2
10-1
9-1
8-3
7-2
6-1
5-1
4-3
3-2
2-1
1-0

References:

[1] Artificial Intelligence: A Modern Approach 3rd Edition by Stuart Russell and Peter Norvig, 2009.

[2] http://en.wikipedia.org/wiki/Alpha%E2%80%93beta_pruning

[3] http://en.wikipedia.org/wiki/Minimax

An Aggie does not lie, cheat or steal or tolerate those who do.