

## Assignment5 – Adversarial Search

### Problem 5.1 (Games for Adversarial Search)

Consider the following properties a game can have.

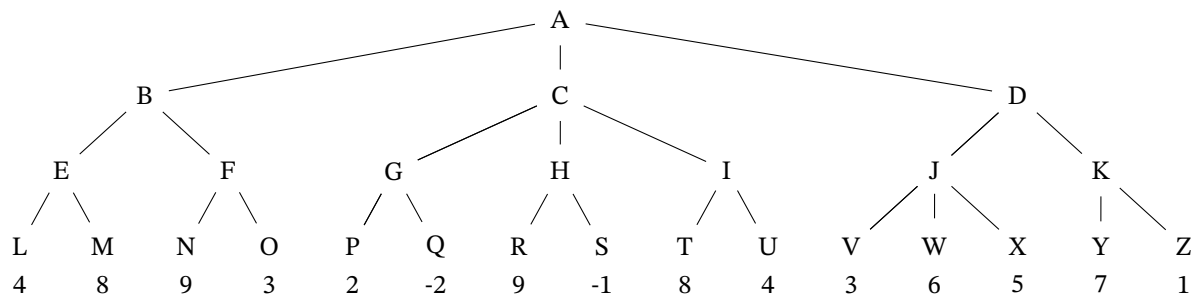
- A 2 players alternating moves
- I discrete state space
- C players have complete information about state
- F finite number of move options per state
- E deterministic successor states
- T games guaranteed to terminate
- U terminal state has zero-sum utility.

For each of the following games, state whether the game violates the property; fill in the corresponding letters (no spaces) into the box or write "none".

1. 2-player poker (until one player is bankrupted): CET
2. *Backgammon*: ET
3. Wrestling (one 5 minute round): AIF
4. Connect Four: none
5. Rock-Paper-Scissors (with a repeat to break ties): AT
6. Meta-Game (player 1's first move is to choose a game that satisfies all properties, subsequent moves play that game): F

### Problem 5.2 (Game Tree)

Consider the following game tree. Assume it is the maximizing player's turn to move. The values at the leaves are the static evaluation function values of the states at each of those nodes.



1. Compute the minimax game value of nodes A, B, C, and D.

---

*Solution:* B = 8, C = 2, D = 6, A = 8

---

2. Max would select move B
3. List the nodes that the alpha-beta algorithm would prune (i.e., not visit). Assume children of a node are visited left-to-right.

---

*Solution:* O, H (and R and S), I (and T and U), K (and Y and Z)

---

4. What reordering of the evaluation function values at the bottom would prune as many branches as possible?

---

*Solution:*

---

If this question is used as a homework, submit your solution as a text file containing the following:

1. Line 1: 4 numbers, separated by space, corresponding to the nodes in alphabetical order, e.g., "1 3 2 5" means A=1, B=3, C=2, D=5.
2. Line 2: The uppercase letter for the selected move.
3. Line 3: The uppercase letters of the pruned moves.

### **Problem 5.3 (Minimax Search in ProLog)**

Consider the following game:

1. There is a pile of  $n$  matches in the middle.
2. Two players alternate taking away 1, 2, or 3 matches.
3. The winner is whoever takes the last match.

Solve this game (for all values of  $n$ ) by *implementing* the minimax algorithm in *Prolog*. Specifically, *implement* exactly the following

- a *Prolog* predicate `value(S, P)` that holds if player  $P$  wins from initial state  $S$ ,
- where the *Prolog* constructor `state(N, P)` represents the game state with  $N$  remaining matches and player  $P$  going next,
- where we represent players  $P$  using 1 for the starting player and  $-1$  for the opponent.

---

*Note:* A partial solution will be explained in the *tutorials*, especially the use of  $\backslash +$  for negation-as-failure and  $!$  for cut.

---

*Solution:*

```
% Game state: number N of remaining matches and current player P=1 or P=-1
```

```
% possible moves in state(N,P) yielding successor state T
successor(state(N,P),T) :- N>0, N2 is N-1, P2 is -P, T=state(N2,P2).
successor(state(N,P),T) :- N>1, N2 is N-2, P2 is -P, T=state(N2,P2).
successor(state(N,P),T) :- N>2, N2 is N-3, P2 is -P, T=state(N2,P2).
```

```
% membership in a list
```

```
contains([H|T],A) :- not(H=A), contains(T,A).
contains([_|_],A).
```

```
% find list Ts of successor states of S using accumulator Acc
```

```
successors(S, Acc, Ts) :- successor(S,T), \+ contains(Acc,T), !,
                           successors(S, [T|Acc], Ts).
successors(_, Acc, Acc).
```

```
% shown until here in the tutorials
```

```
% minvalue(Ss,Sofar,V) holds if V is the minimum value of list of states Ss
% Sofar is accumulator for minimum value seen so far
```

```
% end of list — return accumulator
```

```
minvalue([],Sofar,Sofar).
```

```
% next state has smaller value, replace accumulator and continue with rest
```

```
minvalue([S|Ss],Sofar,V) :- value(S,V1), V1<Sofar, minvalue(Ss,V1,V).
```

```
% next state has non-smaller value, keep accumulator and continue with rest
```

```
minvalue([S|Ss],Sofar,V) :- value(S,V1), V1>=Sofar, minvalue(Ss,Sofar,V).
```

```
% like minvalue
```

```
maxvalue([],Sofar,Sofar).
```

```
maxvalue([S|Ss],Sofar,V) :- value(S,V1), V1>=Sofar, maxvalue(Ss,V1,V).
```

```
maxvalue([S|Ss],Sofar,V) :- value(S,V1), V1<Sofar, maxvalue(Ss,Sofar,V).
```

```
% value(S,V) holds if state S has winner V (1 or -1)
```

```
% our turn (P=1): choose successor with maximal value
```

```
% we lose if no possible move (Ts=[], accumulator initialized to -1)
```

```
value(S,V) :- state(_,P)=S, P = 1, successors(S,[],Ts), maxvalue(Ts,-1,V).
```

```
% opponent's turn (P=-1): choose successor with minimal value
```

```
% we win if no possible move (Ts=[], accumulator initialized to 1)
```

```
value(S,V) :- state(_,P)=S, P = -1, successors(S,[],Ts), minvalue(Ts,1,V).
```

---