Searching Game Trees

1 Game Playing and AI

- Well-defined problems requiring intelligence
- Difficult problems:
  - High degree of uncertainty
  - Huge search space (chess: branching factor 35, depth 50, 35^{100} states)
- Humans play games easily
- Suitable area for studying search methods

2 Formal setting

- Special case of state space search
- Terminal nodes
- Utility function assigns values to the terminal nodes (win=1, loss=-1, draw=0)
- Game tree: MAX nodes (maximizing the utility function) MIN nodes (minimizing the utility function)

3 Minimax Algorithm

- Generate the complete game tree (Figure 1).
- Compute the utility at each node starting from the terminals
- For MAX nodes the utility is the maximum of the successor node utilities
- For MIN nodes the utility is the minimum of the successor node utilities
- Example: Figure 2

4 Improvements of Minimax

- Minimax works only for simple (small) game trees (e.g. tic-tac-toe)
- Cut the tree at some depth and use an heuristic function for computing utility
- The ideal heuristic should compute the probability that MAX wins (MIN loses) given a game position
- Chess: weighted sum of the pieces (deficiency: does not take into account the position)
- Learning heuristic functions (feature representation)
- Depth bound search (the heuristic function at deeper levels is easier to compute and more accurate)
  - Specifying a depth
  - Using iterative deepening given a specified time limit
  - Applying the heuristic function only for "non-critical" positions (where the function does not change too much after the next move)
  - Horizon effect (e.g. queening move in chess)

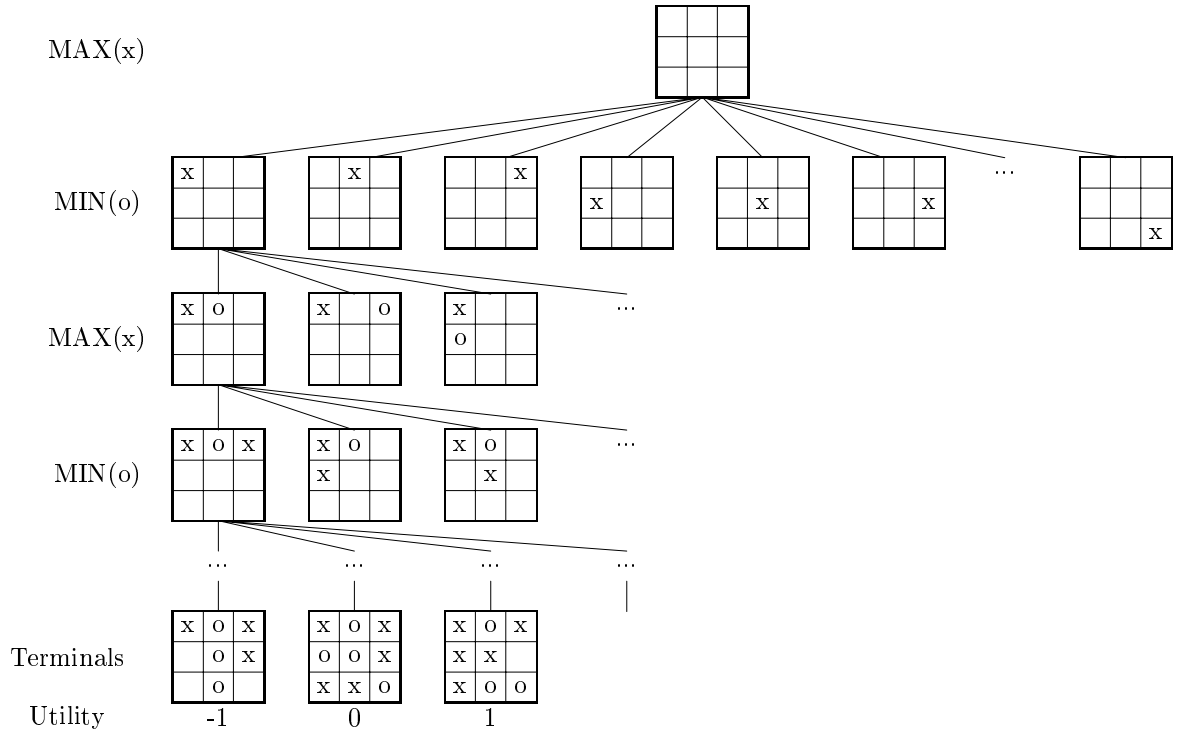

Figure 1: The game tree for tic-tac-toe

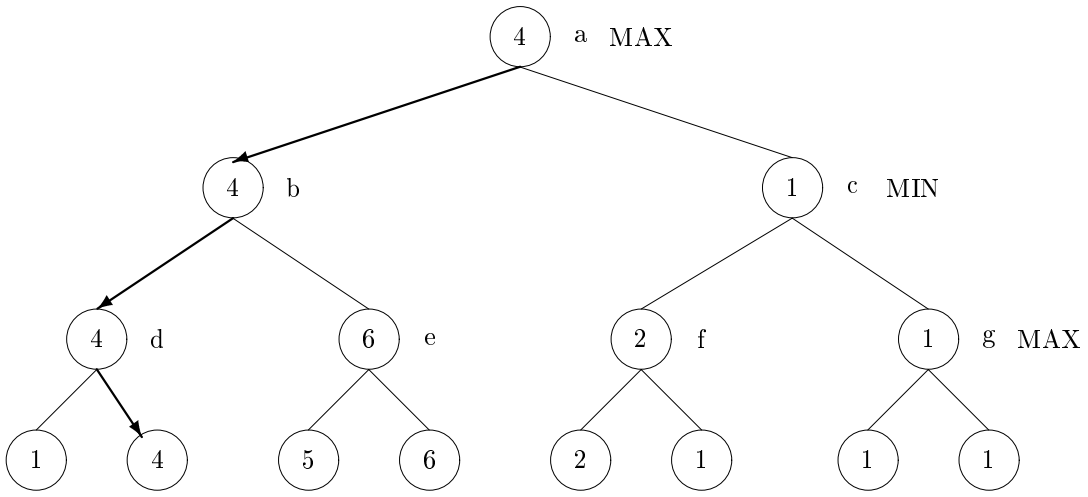

Figure 2: A simple game tree (the winning path for MAX is shown in bold)

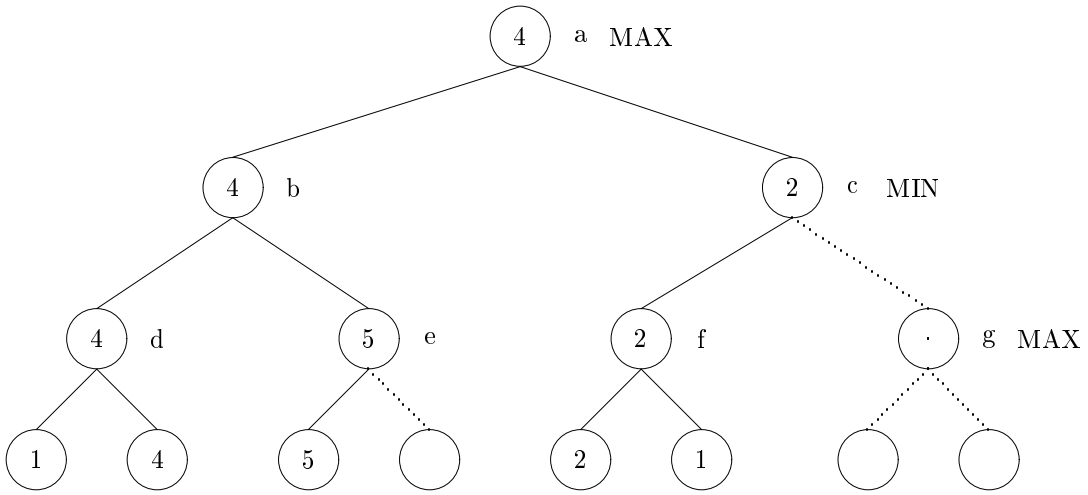

Figure 3: Alpha-Beta Pruning

5 Efficient implementation of Minimax: Alpha-Beta Pruning

- Approximate computation of utility (note the differences between Figure 2 and Figure 3)
- Returns the same move as minimax
- Depth-first search
- Two parameters:
  - α = maximal value of MAX achieved so far
  - β = minimal value of MIN achieved so far
- If MIN gets a value greater than β, then its siblings are skipped (the left successor of e gets 5, which is greater than $\beta = 4$ and the right one is skipped)
- If MAX gets a value less than α, then its siblings are skipped (f gets 2, which is less than $\alpha = 4$ and g is skipped)
- Drawback: dependence on the search order
- If the best moves are evaluated first time complexity decreases from $O(b^d)$ to $O(b^{\frac{d}{2}})$.

6 Games with an element of chance

- Example: backgammon
- Adding one more level in the game tree - chance nodes (MAX-CHANCE-MIN-CHANCE-MAX-...).
- Chance nodes have as many successors as outcomes of the random element (e.g. 21 in backgammon).
- Minimax with element of chance
  - d_i $(i = 1, ..., n)$ – outcomes from the chance nodes
  - $P(d_i)$ – probability of d_i;
  - $S(N, d_i)$ – moves from position N for outcome d_i;
  - If N is MAX: $utility(N) = \sum_{i=1}^{n} p(d_i) \max_{s \in S(N,d_i)} utility(s)$
  - If N is MIN: $utility(N) = \sum_{i=1}^{n} p(d_i) \min_{s \in S(N,d_i)} utility(s)$
- The utility is computed by using not only the terminal values. Therefore values assigned to win, loss and draw affect the choice of moves.
- Time complexity increases (n outcomes from the chance nodes) to $O(b^d n^d)$.
- Alpha-Beta pruning is more complicated.