CISS450 Lecture 5: Adversarial search

Yihsiang Liow

October 17, 2022

Table of contents I

Agenda

- MAX-MIN games
- Minimax algorithm
- Alpha-beta pruning
- Evaluation function
- Cut-off test

Games I

- Multiagent environments
  - Agents can be cooperative and/or competitive
- Adversarial search problems = games
  - Competitive agents
- Many applications outside computer games:
  - Bid-buy actions in stock market
  - Real-life war decisions

Games II

- Focus on two players, alternating, zero-sum, perfect information games
- Example: checkers and chess
- **Zero-sum**: assign values to outcomes in such a way that sum of players' scores is constant, (sometimes zero, not always)
  - Example: Checkers. For red, win=1, loss=-1, draw=0. For black, win=-1, loss=1, draw=0. Sum is always 0.
  - Example: Chess. For red, win=1, loss=0, draw=1/2. For black, win=0, loss=1, draw=1/2. Sum is always 1.

Games III

- Chess:
  - Extremely large state search space and huge branching factor.
  - Average branching factor = 35
  - Assume total number of moves = 50, then search tree is 35^{100} $= 10^{154}$. Number of distinct nodes in search graph is 10^{40}.

MAX-MIN I

- Game with two players: MAX and MIN
- Assume no uncertainty. Perfect information. Players take turns making moves
- State = describe state of game
  - Example: the playing board for checkers and chess
- Initial state = beginning state
  - Example: initial checker/chess board
- `player(s)` = which player is making move at this point.
- `actions(s)` = list of valid actions at this point.
- `result(s, a)`: resulting state after applying action a to state s. (Call new state a successor state.)

MAX-MIN II

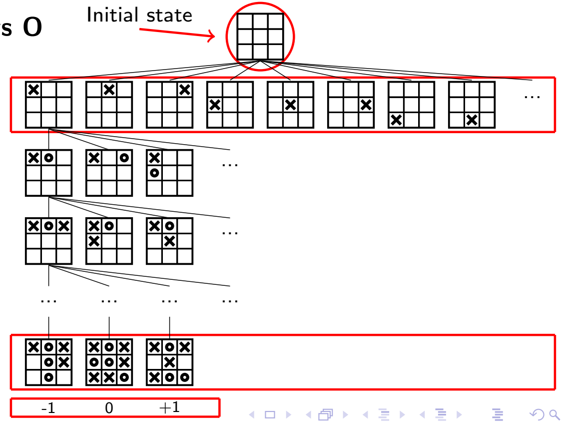
- `terminal_test`(s): tests if game has terminated. Input is game state.
- `utility`(s): numeric value that determines win, loss or draw given state s. For MAX-MIN games, MAX wants to maximize utility function and MIN wants to minimize utility function.
    - Example: Chess. For white 1=win, -1=loss, 0=draw. For black -1=win, 1=loss, 0=draw. White is MAX. Black is MIN. With the setup, the game is zero sum.

Tic-Tac-Toe

MAX plays X, MIN plays O

Initial state

States with actions for MAX
Branching factor = 9



Terminal states

Values of utility function
(for player X)

Tic-Tac-Toe

- State space not too big – 9!, about 360,000 terminal nodes.

Tic-Tac-Toe: Utility function I

- Here's an example of the a utility function (for tic-tac-toe) where player X is MAX and player O is MIN:

```
def UTILITY(ttt):
    if ttt has a row/col/diag of X:
        return 1
    elif ttt has a row/col/diag of O:
        return -1
    else:
        return 0
```

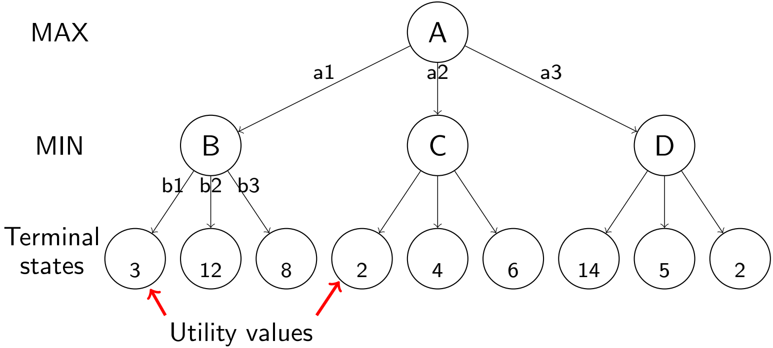
Optimal decisions in games I

- Note the difference between single agent perfect information search and adversarial search – opponent will attempt to disrupt your search toward a goal state
  - For tic-tac-toe, the goal states MAX wants to reach are those with utility value 1, MIN wants to reach goal states with utility value -1.
- Look at a simpler example ...
- Terms: Game ends after 1 move. 1 move = each player has made a move. **Ply** = 1/2 move.

Optimal decisions in games II

- The following is an example of the game tree of states that I will use for tracing. In real life, the game tree of states i much larger and there are also negative utility values.

Optimal decisions in games III



Optimal decisions in games IV

- **Minimax value** of state s:

$$\texttt{MINIMAX\_VALUE}(s, \text{player})$$

$$= \begin{cases} \texttt{UTILITY}(s) & \text{if } s \text{ is terminal} \\ \max\{\texttt{MINIMAX\_VALUE}(s', \text{MIN}) \mid s' \text{ successor of } s\} & \text{if player} = \text{MAX} \\ \min\{\texttt{MINIMAX\_VALUE}(s', \text{MAX}) \mid s \text{ successor of } s\} & \text{if player} = \text{MIN} \end{cases}$$

- (NOTE: I'm deviating from AIMA. AIMA uses node for minimax value computation instead of state and also put player in the node.)
- If both players play optimally from state s to terminal state s' then

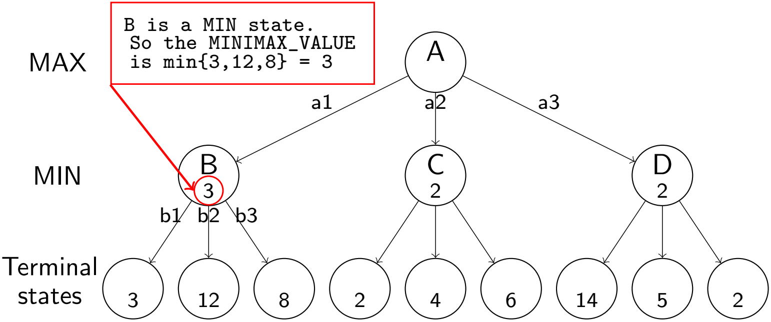
$$\texttt{MINIMAX\_VALUE}(s, \text{player}) = \texttt{UTILITY}(s')$$

Optimal decisions in games V

- Therefore MAX wants to choose action to arrive at state with maximal `MINIMAX_VALUE`.
- If MIN does not play optimally, then MAX will still optimize the utility value.

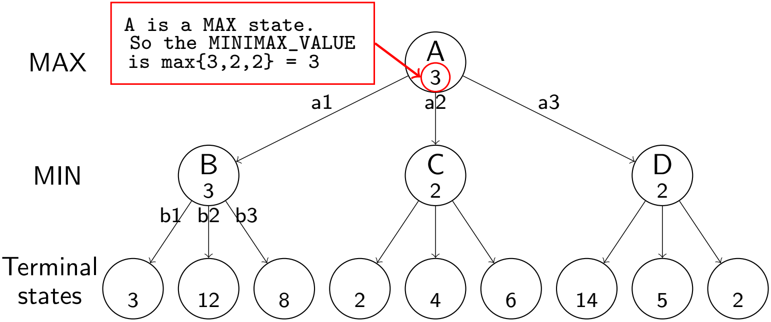
Optimal Strategies

- Compute the minimax value of B,C,D:



Optimal Strategies

- Compute the minimax value of A:



Optimal Strategies

- If MAX plays a3, what will MIN do?

Optimal Strategies

- Compare previous to A playing a1. What is best move for MIN?

MINIMAX Algorithm I

- The `MINIMAX_DECISION(state, player)` is an easy recursive function involving `MAX_VALUE` and `MIN_VALUE` will return an action for MAX, except that the corresponding action is returned (instead of the minimax value). See p.166.

```
def MINIMAX_DECISION(state, player):
    return action such that MIN_VALUE(RESULT(state,
           action) is maximum among all possible actions
def MAX_VALUE(state):
    if TERMINAL_TEST(state): return UTILITY(state)
    v = -INFINITY
    for action in ACTIONS(state):
        s = RESULT(state, action)
        v = max(v, MIN_VALUE(s))
```

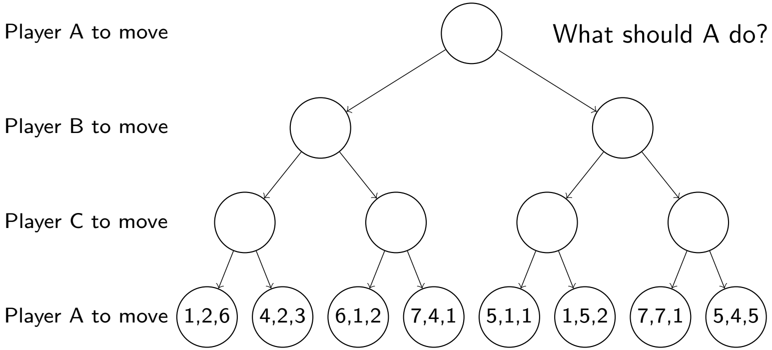
MINIMAX Algorithm II

- `MIN_VALUE` is similar
- Minimax algorithm:
  - Complete **depth-first** traversal of game tree
  - Time $= O(b^m)$, m=max depth, b=branching factor
  - Space $= O(bm)$, if all sucessors generated
  - Space $= O(m)$, is one successor generate at one time
- Problem:
  - Game tree is large
  - Game tree of checkers about 10^{40} nodes
  - Game tree of chess about 10^{120} nodes

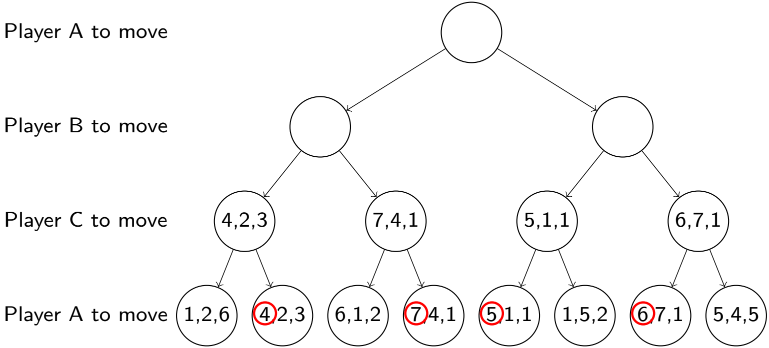
Multiplayer Games I

- If there are n players, then utility function gives a vector of n values
- Each player maximize his value within the vector

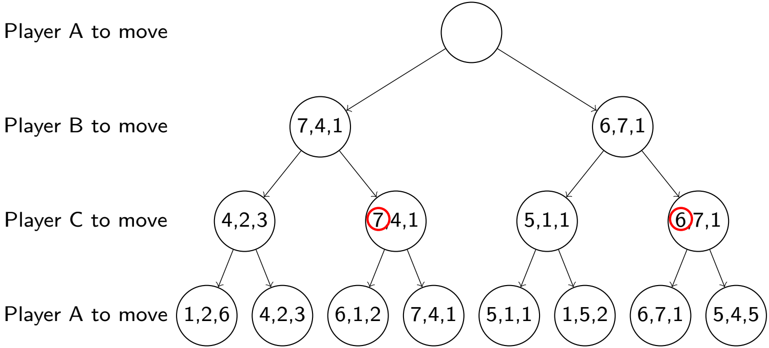
Multiplayer Games II



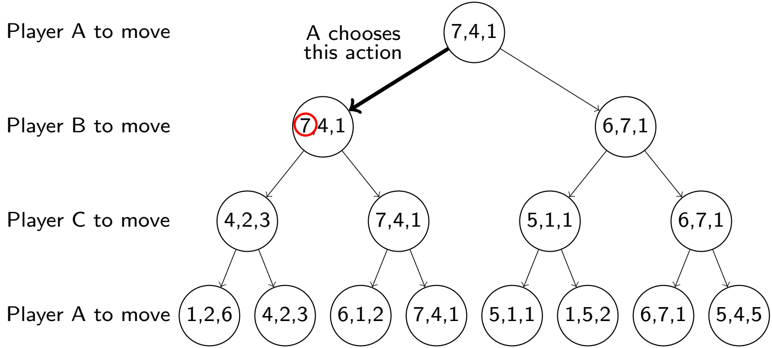
Multiplayer Games



Multiplayer Games



Multiplayer Games



Multiplayer Games I

- (Note: The utility values are filled in breadth first manner. I'm just doing it by hand. Follow the minimax algorithm, the utility values should be filled in DF manner).
- The above game is adversarial.
- Some games allow formation of temporary alliances.

Alpha-Beta Pruning I

- **Alpha-beta pruning**: It's possible to prune away from branches
- Main idea: Look at the expression

$$\max(\min(...), \min(...), \min(...))$$

Alpha-Beta Pruning II

- Note that:

max(min(3,12,8), min(2,4,6), min(3,1,?))
= max(3, min(2, ?, ?), min(3, 1, ?))
= 3

We can ignore these values!

Alpha-Beta Pruning I

- So keep track of running max. Cut off subtree when values for min are too small.

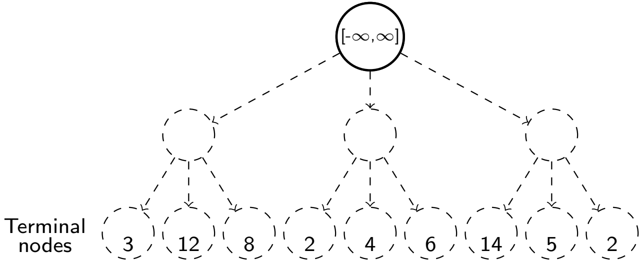
$$\max(\ \min(3,12,8),\ \min(2,4,6),\ \min(3,1,?)\ )$$
$$= \max(\ 3,\ \min(2,?,?),\ \min(3,1,?)\ )$$
$$= 3$$

Why? MAX knows that first option is already 3.
For the second option, the first value is 2. MAX knows that MIN will want to minimize. So the second option will be 2 or worse for MAX. No point continuing with option 2!!!
For the third option, the first suboption is 3. MIN might choose this option but the second suboption might be higher. Continue! ... too bad the second suboption is below 3 (the first option). No point continuing. Too bad the first suboption was a waste!

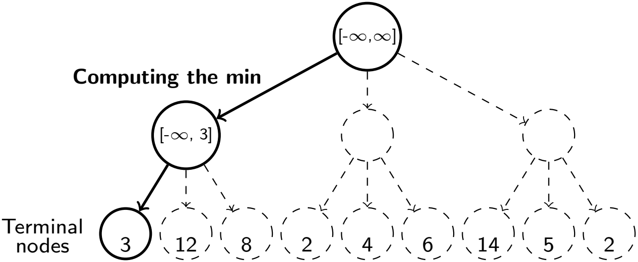
Alpha-Beta Pruning I

- For previous slide. Intuitively, keep running max. If a value computed for an option at the min level drops below the running max, ignore the rest for this option.

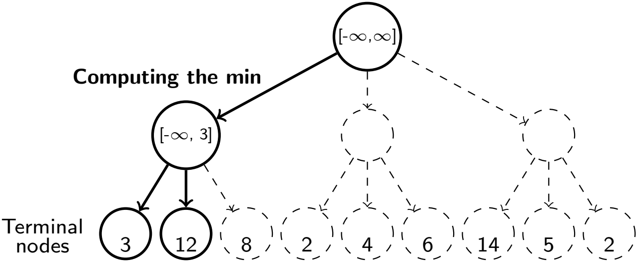
Alpha-Beta Pruning



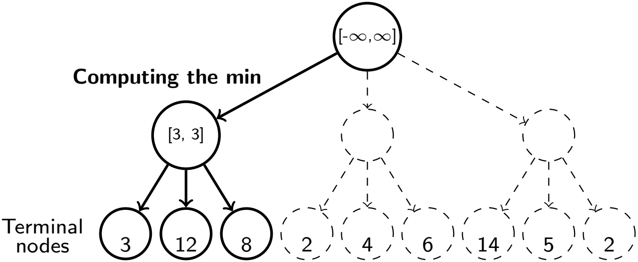
Alpha-Beta Pruning



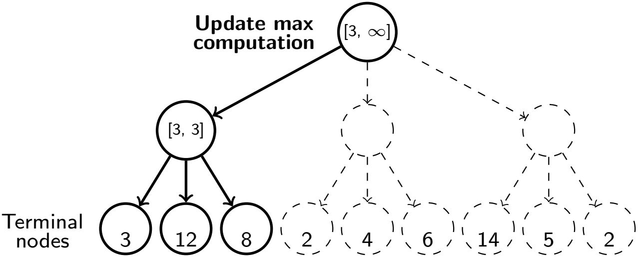
Alpha-Beta Pruning



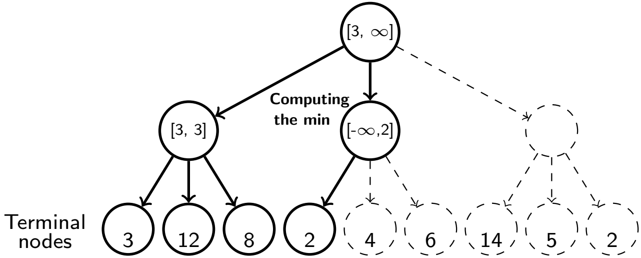
Alpha-Beta Pruning



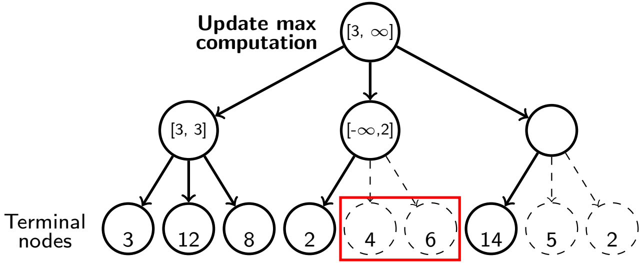
Alpha-Beta Pruning



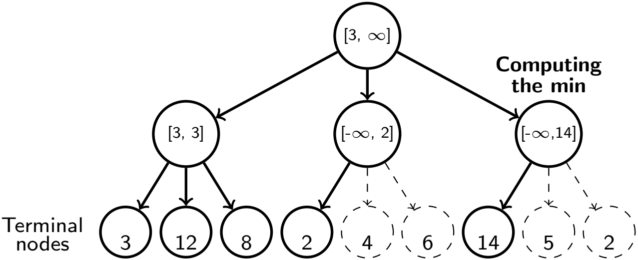
Alpha-Beta Pruning



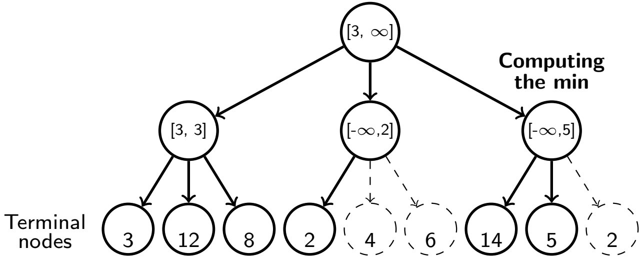
Alpha-Beta Pruning



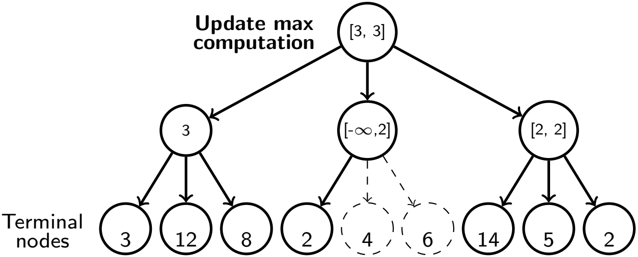
Alpha-Beta Pruning



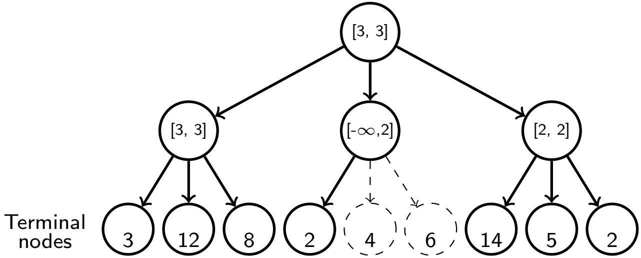
Alpha-Beta Pruning



Alpha-Beta Pruning



Alpha-Beta Pruning



Alpha-Beta Pruning I

- α = value of best alternative for MAX = the running max value computation of MAX
- β = value of best alternative for MIN = the running min value computation of MIN

Alpha-Beta Pruning II

```
def ALPHA_BETA_SEARCH(state, player, alpha, beta):
    v = MAX_VALUE(state, player, -∞, ∞)
    return action in SUCCESSORS(state) with value v
def MAX_VALUE(state, alpha, beta):
    if TERMINAL_TEST(state): return UTILITY(state)
    v = -∞
    for a,s in SUCCESSORS(state):
        v = max(v, MIN_VALUE(s, alpha, beta))
        if v >= beta: return v
        alpha = max(alpha, v)
    return v
```

- MIN_VALUE is similar

Alpha-Beta Pruning III

- I suggest not following AIMA: Combine `MAX_VALUE` function and `MIN_VALUE` function into the `ALPHA_BETA_SEARCH` function. The combined function is not that long and save on unnecessary function calls. This is AI programming and speed even at the level of saving on the constant of big-O is important.

Alpha-Beta Pruning IV

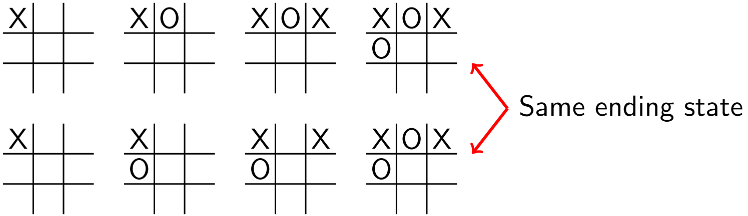
- For implementation, action is returned by `ALPHA_BETA_SEARCH`. So after testing your `ALPHA_BETA_SEARCH`, make your function return (value, action). To make the alpha beta search easy to use, make your alpha beta search `ALPHA_BETA_SEARCH_HELPER` and write another `ALPHA_BETA_SEARCH` function that receives (value, action) and only return action.

Alpha-Beta Pruning V

- Note that pruning depends on ordering of successors – order successors so as to analyze successors that are likely to be best
    - Chess: capturing action, threat action, forward move action, backward move action
- Time
    - Minimax = O(b^m)
    - Best first Alpha-beta = O($b^{m/2}$)
- Effective branching factor for alpha-beta = $b^{1/2}$

Alpha-Beta Pruning VI

- Transpositions = permutations of moves that end in same state



- Store evaluation of states
- Transposition table = store of old states (with values)
- Cannot keep all states – have to choose

Heuristics I

- Minimax, alpha-beta requires search to terminal nodes for part of game tree – too expensive because game tree is too deep
  - Need to search unpruned subtrees down to leaves
- Use heuristics to stop search earlier:
  - **Evaluation function** to estimate utility value
  - **Cutoff test** to replace terminal test function

Evaluation Functions I

- Required qualities of evaluation function EVAL:
  - For leaf nodes n, n':

$$\texttt{UTILITY}(n) \le \texttt{UTILITY}(n') \implies \texttt{EVAL}(n) \le \texttt{EVAL}(n')$$

  - `EVAL` is quickly computed
  - For nonleaf node n, `EVAL(n)` should approximate chances of winning

Evaluation Functions II

- How does `EVAL` approximate chances of winning? Theoretically, compute `EVAL` as expected value.
  - Form classes of states by feature functions
  - Let C be a class. Suppose C = W U D U L (win, draw, lose states)
  - Suppose n is in C. Then for MAX-MIN game: `EVAL(n) = 1(#W/#C) + 0(#D/#C) + (-1)(#L/#C)`
- Practically, compute some form of EVAL for each feature and then compute EVAL as weighted sum

Evaluation Functions III

- Weighted linear function for features f_1 , ..., f_n

  $\texttt{EVAL(x)} = w_1 f_1 \texttt{(x)} + \ldots + w_n f_n \texttt{(x)}$

  - w_i are the weights
  - Example:
    - f_1 = number of pawns $w_1 = 1$
    - f_2 = number of knights $w_2 = 3$
    - f_3 = number fo bishops $w_3 = 3$
    - f_4 = number of rooks $w_4 = 5$
    - f_5 = number of queens $w_5 = 9$

Evaluation Functions IV

- Advantage of simple weighted linear sum: independent of game rules
- Linear weighted sum as above does not take into account many real-life issues
  - Certain pieces work well together
    - 2 bishops work well together
  - Feature value might change with time
    - Bishop frequently more powerful during endgames
  - Feature value might change with general layout of chess pieces
    - Rook more powerful if not blocked
- There are techniques for making games learn to play better with time – Learning Theory
  - Basically use training data to improve algorithm

Cutting Off Search I

- In alpha-beta search, replace

  ```
  if TERMINAL_TEST(state): return UTILITY(state)
  ```

  by

  ```
  if CUTOFF_TEST(state,depth): return EVAL(state)
  ```

- Alpha-beta search needs to remember how deep in the game search and CUTOFF_TEST returns true when current depth is greater than a fixed depth d
  - d is chosen to maximize use of time
- Better: use iterative deepening, varying d. Return action for deepest complete search when time's up.

Cutting Off Search II

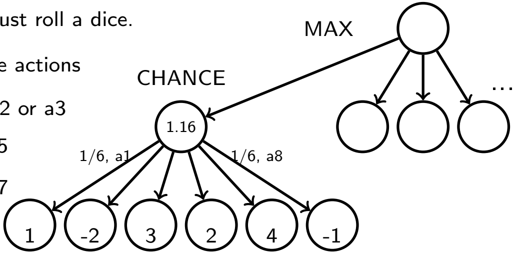
- Evaluation works when there is great variation in EVAL values – quiescent states
- After a capturing move, EVAL based on counting pieces might experience great variation
- **Quiescence search** = search until quiescent state is reached
- Modify `CUTOFF_TEST`: when state is not quiescent, keep expanding (within time).
  - Example: Analyze all possible capturing moves

Probabilistic Games I

- Probabilitic MAX-MIN games: MAX-MIN is a game where there is an element of chance. Usually the game tree is the same as a MAX-MIN game tree but actions of MAX and MIN are controlled/limited by probabilitic events

Example: MAX must roll a dice.

Value	possible actions
1	a1 or a2 or a3
2 or 3	a4 or a5
4	a6 or a7
5 or 6	a8



Probabilistic Games I

- Instead of minimax value compute expected minimax value based on probability:

```
EXPECTIMINIMAX(n)
= UTILITY(n)                                         if n leaf
= max{EXPECTIMINIMAX(s) | s successor of n}
                                                     if n MAX node
= min{EXPECTIMINIMAX(s) | s successor of n}
                                                     if n MAX node
= sum of P(s)*EXPECTIMINIMAX(s) where s sucessor of n
                                                  if n CHANGE node
```

- Recall time for deterministic case $=$ O(b^m)
- So time for probabilistic MAX-MIN $=$ O((bn)^m) $=$ O($b^m n^m$) where n $=$ number of probabilistic events at each change node

State-of-the-Art Game Programs I

- Chess
  - Deep Blue beat Kasparov in 1997
  - Speed: avg=126M nodes/s, max=330M nodes/s
  - Depth: avg=14, max=20
  - Iterative deepening alpha-beta with transposition table
  - 8000 feature functions
  - Database: 4000 openings, 700000 grandmaster games, endgames with 5 pieces or less

State-of-the-Art Game Programs II

- Checkers:
  - Arthur Samuel's checkers program best champion in 1962. 10K member, 0.000001 GHz CPU.
  - Chinook won 2^{nd} place in 1990 US Open
    - Database of endgames of 444 billion positions with 8 or fewer pieces