Adversarial Search a.k.a. Game Search

Vincent Cicirello

Overview

- Definition of games
- Game Terminology
- Game Trees
- Game theoretic values
- Computing game theoretic values with recursive minimax
- Computing game theoretic values with dynamic programming
- Alpha-beta search
- Playing games in real-time

Two-player zero-sum discrete finite deterministic games of perfect information

- **Two player:** well, there are two players…
- **Zero Sum:** In any outcome of any game Player A's gains equals Player B's losses.
- **Discrete:** All game states and decisions are discrete values.
- **Finite:** There are only a finite number of states and decisions.
- **Deterministic:** no chance… no dice rolls… etc
- **Games:** defined shortly….
- **Perfect information:** Both players can see the state, and each decision is made sequentially.

A game defined….

- A two-player zero-sum discrete finite deterministic game of perfect information is a quintuplet, (S, I, Succs, T, V) where:
  - **S:** Finite set of states (must include sufficient information to deduce whose turn it is to move next)
  - **I:** Initial state
  - **Succs:** Function that takes a state as input and returns a set of states (legal positions after a move).
    - Must be non-empty if its argument is not a terminal state
  - **T:** The set of terminal states (i.e., states when game ends and payoff occurs)
  - **V:** Mapping from terminal states to real numbers (payoff to player A and loss to player B)

Example: Nim

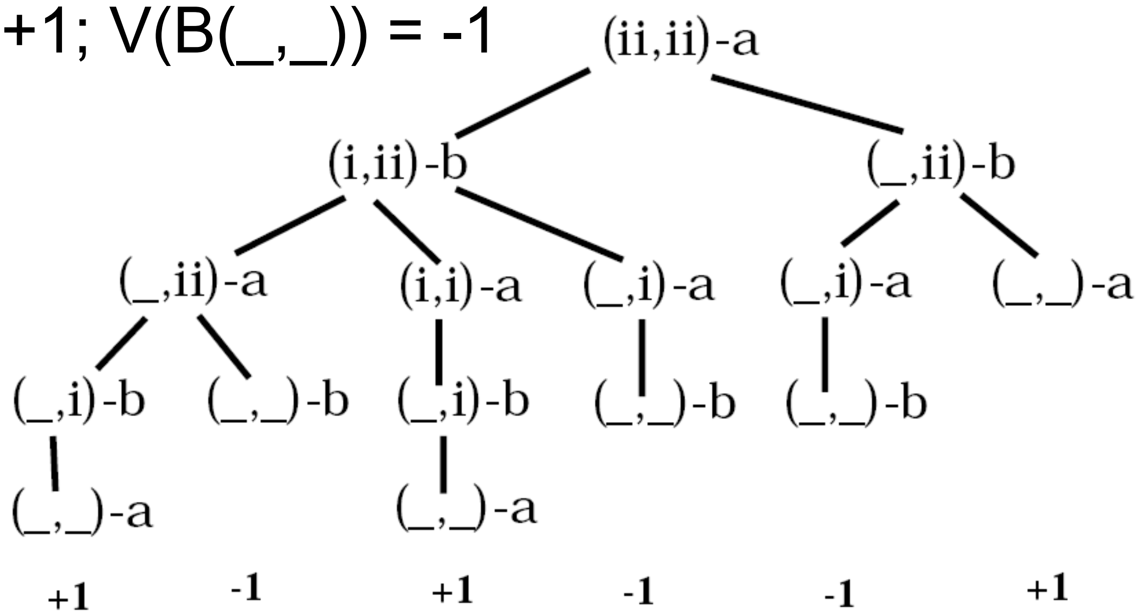
- You begin with some number of piles of matches.
- During a turn, the player may remove any number of matches from one pile
- The last person to remove a match loses
- In II-Nim, you begin with two piles each with two matches
- **States of Nim**
  - A(jj,jj); A(j,jj); A(_,jj); A(jj,j); A(jj,_); A(j,j); A(_,j); A(j,_); A(_,_)
  - B(jj,jj); B(j,jj); B(_,jj); B(jj,j); B(jj,_); B(j,j); B(_,j); B(j,_); B(_,_)

Nim (continued)

- **States of Nim**
  - A(jj,jj); A(j,jj); A(_,jj); A(jj,j); A(jj,_); A(j,j); A(_,j); A(j,_); A(_,_)
  - B(jj,jj); B(j,jj); B(_,jj); B(jj,j); B(jj,_); B(j,j); B(_,j); B(j,_); B(_,_)
- **Common Trick: Symmetry**
  - Some states are trivially equivalent (e.g., A(_,jj); A(jj,_))
  - Use some canonical description to make them one state
    - e.g., left pile always has at least as many matches as right
- **States of Nim using Symmetry**
  - A(jj,jj); A(jj,j); A(jj,_); A(j,j); A(j,_); A(_,_)
  - B(jj,jj); B(jj,j); B(jj,_); B(j,j); B(j,_); B(_,_)

Nim formalized

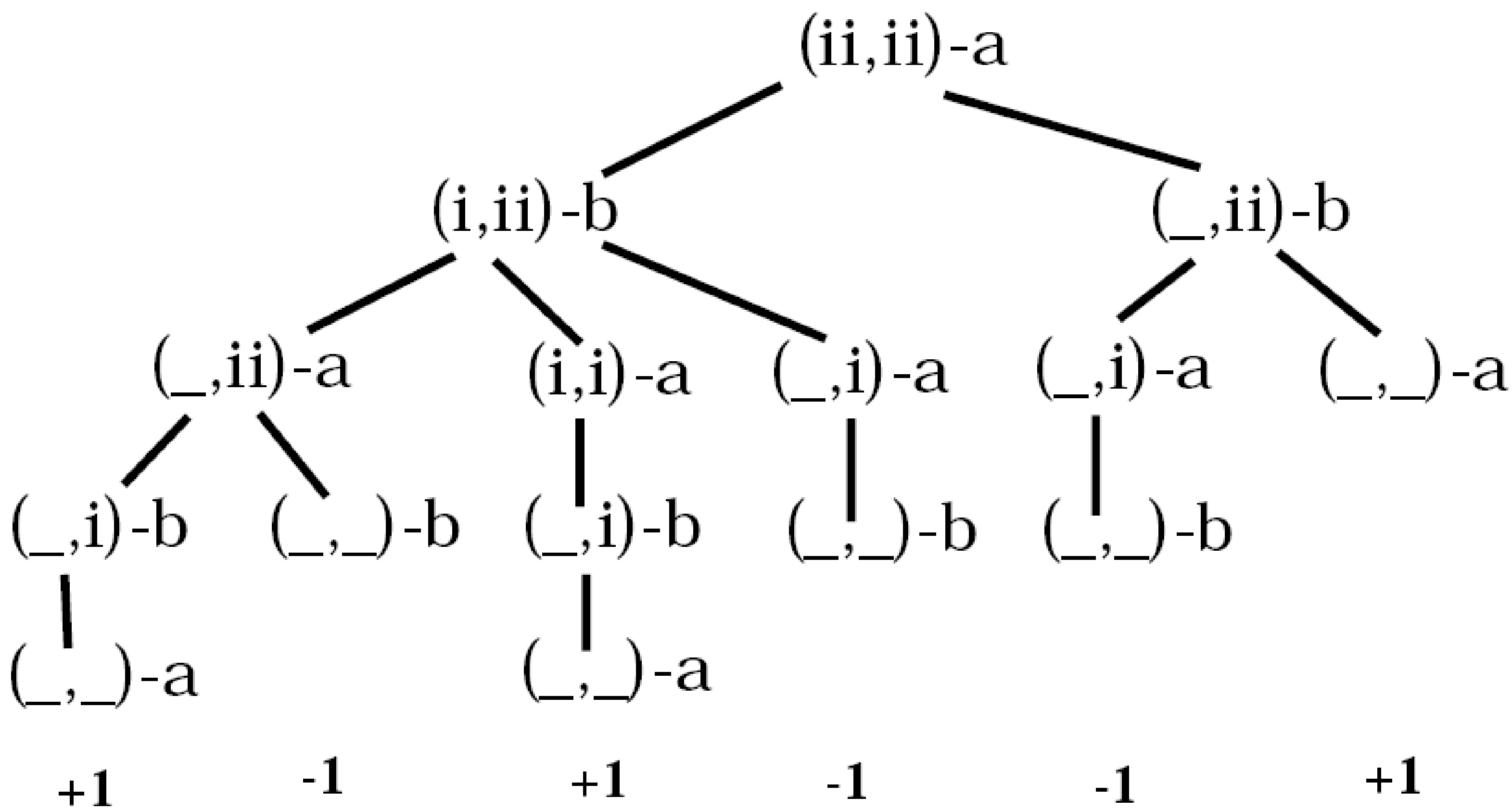
- S = {A(jj,jj); A(jj,j); A(jj,_); A(j,j); A(j,_); A(_,_); B(jj,jj); B(jj,j); B(jj,_); B(j,j); B(j,_); B(_,_)}
- I = A(jj,jj)
- Succs(A(jj,jj)) = {B(jj,j); B(jj,_)}
- Succs(B(jj,j)) = {A(jj,_); A(j,j); A(j,_)}
- Succs(B(jj,_)) = {A(j,_); A(_,_)}
- etc …
- T = {A(_,_), B(_,_)}
- V(A(_,_)) = +1; V(B(_,_)) = -1



Game Theoretic Value

- **Definition:** The **game theoretic value** (a.k.a. the **minimax value**) of a state is the value of the terminal state that will be reached if both players play optimally.
- How can we find the minimax values for non-terminal states?
- *Idea: Fill in the tree bottom-up.*

Game Theoretic Value: Nim



The Minimax Algorithm

- Generate the full Game tree, storing it in memory
- Run through all of the terminal states assigning them values.
- Run through all predecessors assigning them values, etc, etc, etc…
- Question: Do we really need to store the whole game tree in memory?
  - NO… Can do a DFS-like algorithm

- Minimax-Value(S)
  - if (S is a terminal)
    - return V(S)
  - else
    - Let S1,S2,…Sk = Succs(S)
    - Let vi = Minimax-Value (Si) for each Si
    - If PlayerToMove(S) = A
      - return Max(vi)
    - else
      - return Min(vi)

Dynamic Programming (DP)

- Dynamic Programming---Russell & Norvig’s definition:
  - “solutions to subproblems are constructed incrementally from those of smaller subproblems and are cached to avoid recomputation”
- You’ve may have encountered this in other classes (e.g., possibly if you’ve taken Data Structures, or perhaps if you’ve taken OR).

DP for Solving Games

- Consider a game with N states, where the game is usually of length l and where each state has b successors.
- Minimax requires that $O(b^l)$ states are expanded.
  - This is best case as well as worst case.
  - Whereas, DFS for simple search problems in the best case can be O(l).
- What if the number of states N is smaller than b^l?
  - E.g., for chess, N=10^40, while b^l=10^120
- In such cases, DP is a better method, assuming you can afford the memory.
  - Cost of DP: O(N l)

Example: DP for Chess Endgames

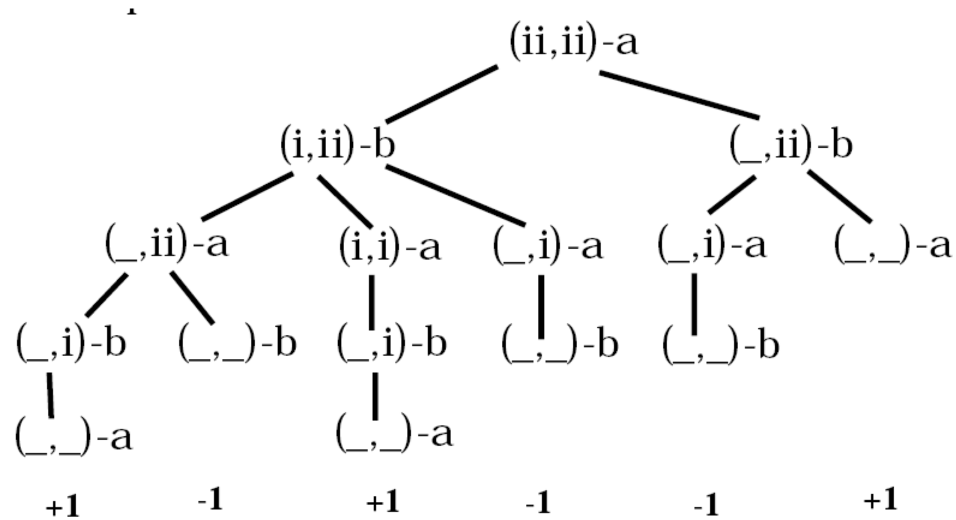
- Consider that there are only 4 chess pieces left on the board.
- With sufficient computational resources, you can compute, for all possible positions, whether it is a win for black, white, or a draw.
- Details next slide….

DP for Chess Endgames

- Assume there are N positions with no more than 4 pieces left:
  1. Define a 1-to-1 mapping from the N board positions to the integers 0..N-1
  2. Create a large array of length N (with 2 bits per entry). Each element in the array can take on one of three values:
     - W: White will eventually win.
     - B: Black will eventually win
     - ?: We don't know who wins from this state
  3. Mark all terminal states with their values, W or B.
  4. Look through all states still marked by "?"
     - If W is about to move, then
       - if all successors are marked with B, mark the state B
       - if any successor state is marked W, then mark the state W
       - else leave the state unchanged (marked "?")
     - if B is about to move, then
       - if all successors are marked with W, mark the state W
       - if any successor state is marked B, then mark the state B
       - else leave the state unchanged (marked "?")
  5. If 4 changed the label of at least one state, then repeat 4.
  6. Any state still marked with "?" is a state from which no one can force a win---thus a draw

Cutting off unneeded search states

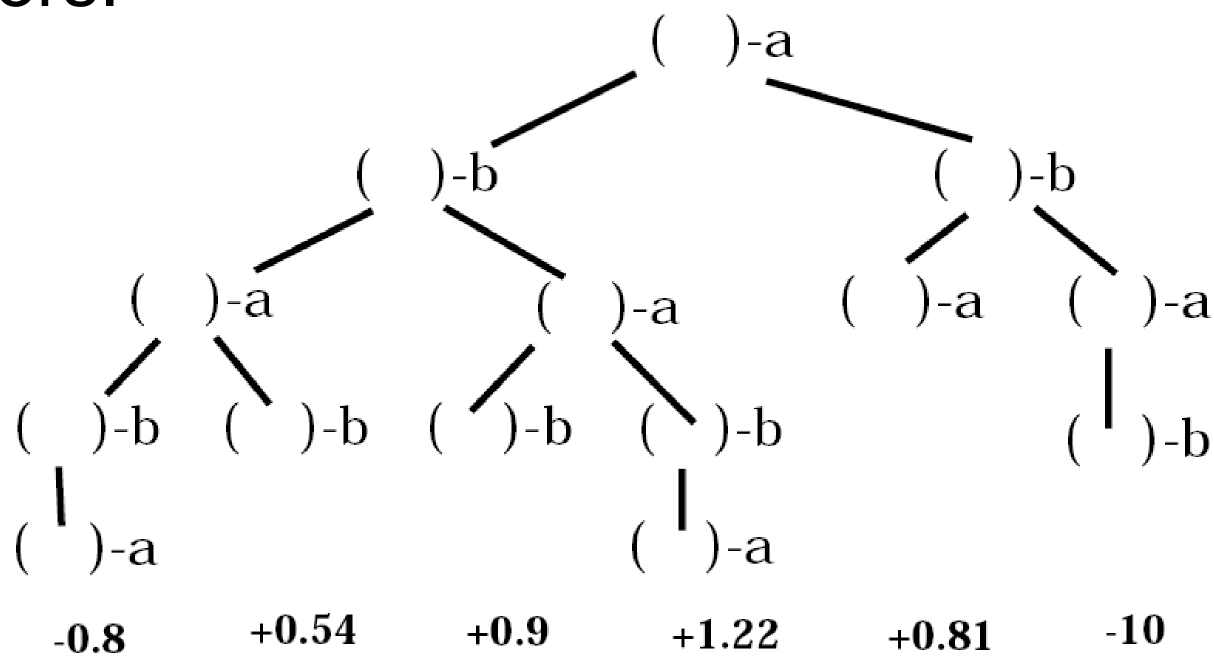
- If we knew the only possible outcomes were +1 and -1, can we save computation?



- Yes… a lot actually
  - though not much in this example
  - if any successor is a forced win for the current player, don't bother expanding further successors

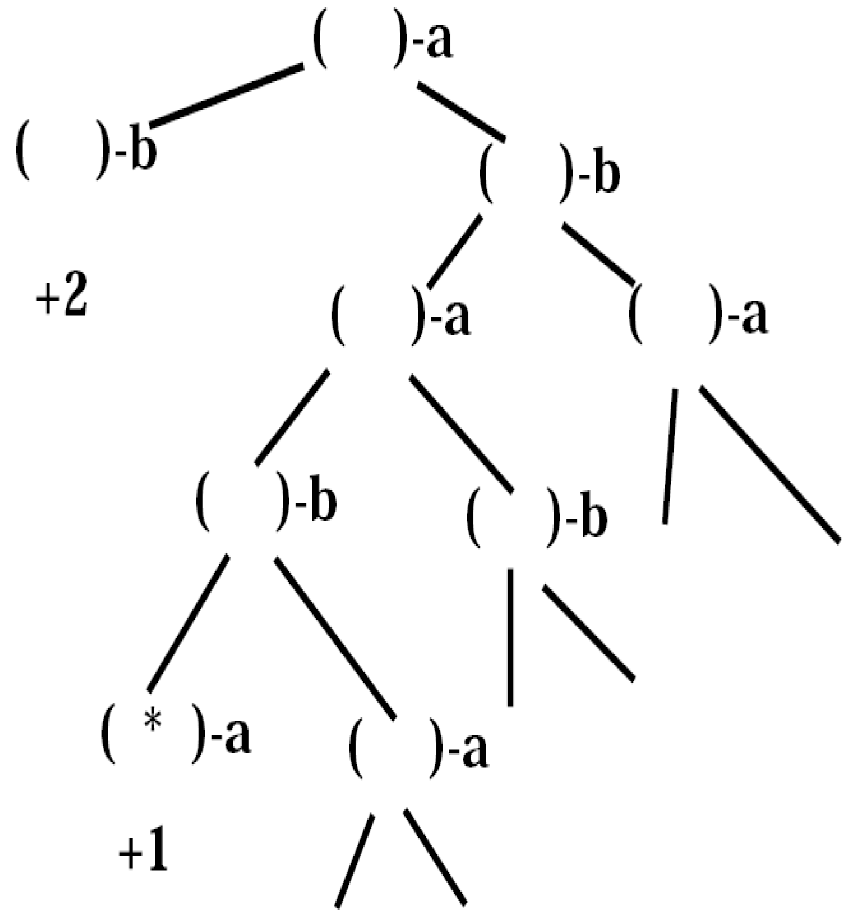
What if possible terminal values are unknown?

- Do DFS, but if something is discovered that implies your parent would not choose you, then don't bother expanding further successors.
- More generally, not just your parent, but any ancestors.

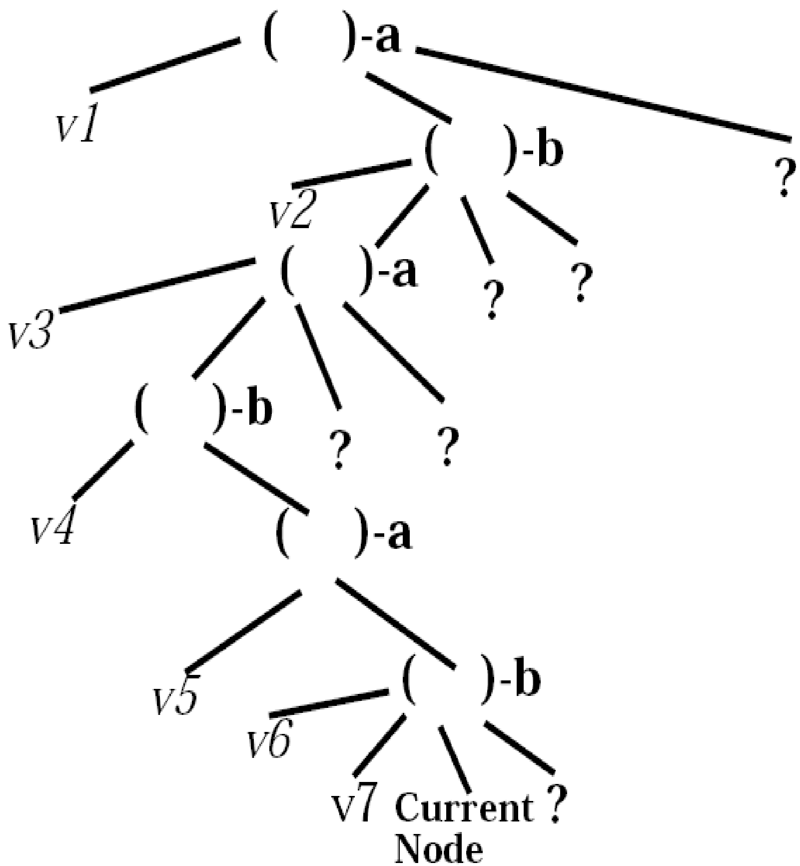


An ancester causing cut-off

- Suppose we've done a full DFS, expanding left-most successors first and that we are currently at the search state marked by the *
- What can we cut off in the rest of the search?
- If either player has a better alternative at an ancestor of a given search node, then it will not be visited.



A general cut-off rule



- In this example:
  - let α=max(v1,v3,v5)
  - let β=min(v6,v7)
  - if β <= α, then we can be certain that it is a waste of time searching the "current node" or its sibling to the right
- In general:
  - if at a B-move node,
    - let α =max of all A's choices on current path, and
    - let β=min of all B's choices including those at current node
    - Cut-off if β<= α
  - Converse rule if at an A-move node

Alpha-Beta Pruning

- Alpha-Beta Pruning from Russell & Norvig
- Assumes players alternate moves

What's the top-level call look like?

Max-Value(S,-∞ ,+ ∞)

function **Max-Value**(s,α,β)
inputs:
  s: current state in game, A about to play
  α: best score (highest) for A along path to s
  β: best score (lowest) for B along path to S
output: *min(β , best-score (for A) available from s)*

```
if ( s is a terminal state )
then return ( terminal value of s )
else for each s' in Succs(s)
        α := max( α , Min-value(s',α,β) )
        if ( α >= β ) then return β
  return α
```

function **Min-Value**(s,α,β)
output: *max(α , best-score (for B) available from s)*

```
if ( s is a terminal state )
then return(terminal value of S)
else for each s' in Succs(s)
        β:= min( β, Max-value(s',α,β) )
        if ( β <= α) then return α
  return β
```

How useful is alpha-beta pruning?

- What is the best case performance of alpha-beta?
- How much of the tree would you examine if you were very lucky in the order you tried successors?
- Best case:
  - The number of nodes you need to search in the tree is $O(b^{d/2})$.
  - The square root of the recursive minimax cost.
  - Large real-sized games with a huge number of states are still problematic (e.g., chess)

Solving Games

- **Solving a game** means proving the game-theoretic value of the start state
- Some games have been solved
  - by brute-force DP
    - Four-in-a-row
    - Some chess endgames, e.g.:
      - rook and king against king (from most starting positions): win
      - two bishops and king against king (from most starting positions): win
      - bishop, knight, and king against king (from most starting positions): win
      - two knights and king against king (from most starting positions): draw… a few rare exceptions: win
  - brute-force DP from end to create an end-game DB plus alpha-beta search from start
    - nine men's morris
  - Mostly brute-force with some game specific analysis
    - Connect-Four
  - Checkers has been solved (draw)
    - Chinook solved Checkers during a period spanning 1989-2007
    - Mostly brute-force DP (via dozens of computers)
    - In 1996, Chinook became first computer program to win a human world championship

Game Playing vs Game Solving

- Two very different activities
- **Game Solving:** finding the true game-theoretic value of a state.
- What about **game playing**?
- Game solving often very different from playing a game well.
- Example, what do real chess playing programs do?
- Some features that the search algorithms covered so far in this course don't have:
  - Cannot possibly find a guaranteed solution.
  - Must make decisions quickly in real-time.
  - It is not possible to pre-compute a solution.

Heuristic Evaluation Functions

- Popular solution: use heuristic evaluation functions
- An evaluation function maps a state to a real value.
  - The larger the evaluation, the larger the true game-theoretic position is estimated to be.
- Note: this is not the same as the heuristic in A*…
  - no notion of admissibility
  - not an estimate of path cost to reach a goal
- Search the game tree as deeply as time allows
- Leaves of tree you search are not leaves of game tree, but are intermediate nodes
- The values assigned to leaves are from the heuristic evaluation function

Heuristic Evaluation Intuition

- **Visibility:**
  - Evaluation function will be more accurate nearer the end of the game.
  - So worth using heuristic estimates from there.
- **Filtering:**
  - If we used the evaluation function without searching, we'd be using a handful of inaccurate estimates (near the root).
  - By searching, we're combining thousands of these estimates, hopefully eliminating noise.
- Is this "intuition" dubious?
  - Yes. Can give counter-examples…
  - But often works very well in practice for real games…

Heuristic Evaluation Example

- A simple heuristic for chess:
  - The typical introductory chess book will label:
    - a bishop or knight worth the value of 3 pawns; a rook worth 5; a queen worth 9
  - This leads to a simple **weighted linear evaluation** function
- More sophisticated chess heuristics consider other state features:
  - good pawn structure might be worth value of a pawn
  - “king safety” might be worth a pawn
- Or **nonlinear evaluation functions** are possible:
  - two bishops might be worth slightly more than twice the value of a single bishop
  - a bishop near the end of the game may be worth more than earlier in the game (e.g., more powerful in open space)
- Machine learning also applicable here

Some Other Issues for Real Game Playing Programs

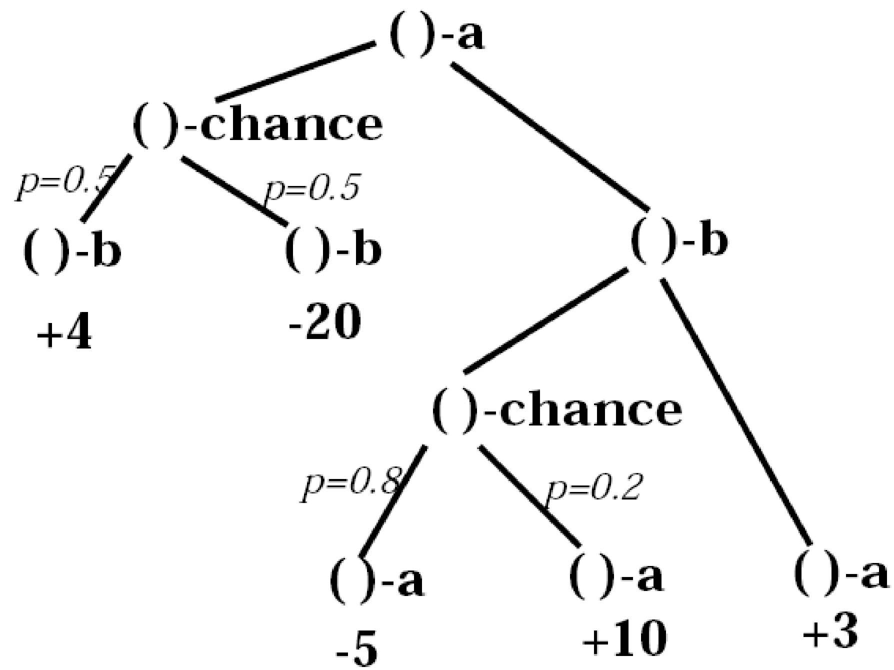
- How to determine how far to search if you only have a fixed time to make a decision.
- **Quiescence:** What if you stop the search at a state where subsequent moves drastically change the evaluation?
  - e.g., you search to depth d in chess, but at depth d+1, a queen is taken…
- **Quiescence search:** an extra bit of search to attempt to reach a quiescent state
  - e.g., in chess, continue search only considering “capture” moves to resolve any uncertainties in position

More issues for real game playing

- **The horizon problem:**
  - Consider a state in which it is inevitable that your opponent will be able to do something bad to you.
    - e.g., an inevitable queening of a pawn
  - Now consider that you have some delaying tactics.
  - The search algorithm won't recognize the inevitable if the number of delaying steps exceeds the search depth limit…
  - Thus not recognizing the badness of the search state.
- **Endgames:** Are easy to play well. How?
  - An end game database
    - essentially a lookup table (e.g., generated by DP)
- **Openings:** Are easy to play well. How?
  - An opening book
  - e.g., for chess, based on hundreds of years of human chess playing knowledge

2-player zero-sum finite NONdeterministic games of perfect information

- The search tree now includes states in which neither player makes a choice.
- Instead, a random decision is made according to a known set of outcome probabilities.
- Game-theoretic value if the **expected** final outcome if both players are optimal.



Expectiminimax

- Obvious generalization of minimax:
  - Expectiminimax(n) =
    - Value(n) if n is a terminal state
    - max{s in successors(n)} Expectiminimax(s) if n is a Max node
    - min{s in successors(n)} Expectiminimax(s) if n is a Min node
    - Sum{s in successors(n)} P(s) Expectiminimax(s) if s is a chance node
- Can we use alpha-beta pruning?
  - Yes…
  - for Min and Max nodes it works unchanged
  - for chance nodes, if we have a bound on terminal values
    - then we can place an upper bound on the value of a chance node without looking at all of its children

Bad News for Expectiminimax

- Assume a game with dice rolls.
- Expectiminimax considers all possible dice roll sequences, then it is:
  - $O(b^m n^m)$ where n is the number of distinct dice rolls
- Example: Backgammon
  - n=21
  - b usually 20, but as high as 4000 for dice rolls that are doubles
  - can probably only manage about m=6 (3 moves each player)
- The equivalent of Alpha-Beta pruning helps the situation a bit but not much
- State-of-the-art Backgammon programs rely heavily on sophisticated evaluation heuristics utilizing machine learning techniques