CSC384: Introduction to Artificial Intelligence

Game Tree Search

- *Chapter 5.1, 5.2, 5.3, 5.6 cover some of the material we cover here. Section 5.6 has an interesting overview of State-of-the-Art game playing programs.*

- *Section 5.5 extends the ideas to games with uncertainty (We won't cover that material but it makes for interesting reading).*

Generalizing Search Problem

- *So far: our search problems have assumed agent has complete control of environment*
  - *State does not change unless the agent (robot) changes it.*
    - *All we need to compute is a single path to a goal state.*

- *Assumption not always reasonable*
  - *Stochastic environment (e.g., the weather, traffic accidents).*
  - *Other agents whose interests conflict with yours*
    - *Search can find a path to a goal state, but the actions might not lead you to the goal as the state can be changed by other agents (nature or other intelligent agents)*

Generalizing Search Problem

- *We need to generalize our view of search to handle state changes that are not in the control of the agent.*

- *One generalization yields game tree search*
  - *Agent and some other agents.*
  - *The other agents are acting to maximize their profits*
    - *this might not have a positive effect on your profits.*

General Games

- *What makes something a game?*
  - *There are two (or more) agents making changes to the world (the state)*
  - *Each agent has their own interests*
    - *e.g., each agent has a different goal; or assigns different costs to different paths/states*
  - *Each agent tries to alter the world so as to best benefit itself.*

General Games

- *What makes games hard?*
  - *How you should play depends on how you think the other person will play; but how they play depends on how they think you will play; so how you should play depends on how you think they think you will play; but how they play should depend on how they think you think they think you will play; ...*

Properties of Games considered here

- *Zero-sum games: Fully competitive*
  - *Competitive: if one play wins, the others lose; e.g. Poker – you win what the other player lose*
  - *Games can also be cooperative: some outcomes are preferred by both of us, or at least our values aren't diametrically opposed*
- *Deterministic: no chance involved*
  - *(no dice, or random deals of cards, or coin flips, etc.*
- *Perfect information (all aspects of the state are fully observable, e.g., no hidden cards)*

Our Focus: Two-Player Zero-Sum Games

- *Fully competitive two player games*
  - *If you win, the other player (opponent) loses*
  - *Zero-sum means the sum of your and your opponent's payoff is zero---any thing you gain come at your opponent's cost (and vice-versa).*
    - *Key insight: How you act depends on how the other agent acts (or how you think they will act)*
    - *and vice versa (if your opponent acts rational)*
- *Examples of two-person zero-sum games:*
  - *Chess, checkers, tic-tac-toe, backgammon, go, Doom, "find the last parking space"*
- *Most of the ideas extend to multiplayer zero-sum games (cf. Chapter 5.2.2)*

Game 1: Rock, Paper, Scissors

- *Scissors cut paper, paper covers rock, rock smashes scissors*
- *Represented as a matrix: Player I chooses a row, Player II chooses a column*
- *Payoff to each player in each cell   (Pl.I / Pl.II)*
- *1: win, 0: tie, -1: loss*
  - *so it's zero-sum*

	Player II		
	R	P	S
R	0/0	-1/1	1/-1
P	1/-1	0/0	-1/1
S	-1/1	1/-1	0/0

(Player I labels the rows)

Game 2: Prisoner's Dilemma

- *Two prisoner's in separate cells, sheriff doesn't have enough evidence to convict them. They agree ahead of time to both deny the crime (they will **cooperate**).*

- *If one defects (i.e., confesses) and the other doesn't*
  - *confessor goes free*
  - *other sentenced to 4 years*

	Coop	Def
Coop	3/3	0/4
Def	4/0	1/1

- *If both defect (confess)*
  - *both sentenced to 3 years*
- *If both cooperate (neither confesses)*
  - *both sentenced to 1 year on minor charge*

- *Payoff: 4 minus sentence*

Extensive Form Two-Player Zero-Sum Games

- *Key point of previous games: what you should do depends on what other guy does*

- *But previous games are simple "one shot" games*
  - *single move each*
  - *in game theory: strategic or normal form games*

- *Many games extend over multiple moves*
  - *turn-taking: players act alternatively*
  - *e.g., chess, checkers, etc.*
  - *in game theory: extensive form games*

- *We'll focus on the extensive form*
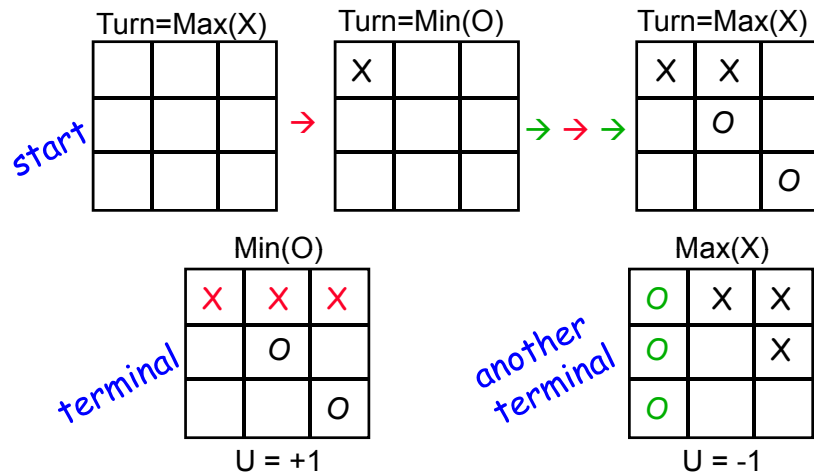 - *that's where the computational questions emerge*

Two-Player Zero-Sum Game – Definition

- *Two players A (Max) and B (Min)*
- *Set of positions P (states of the game)*
- *A starting position $s \in P$ (where game begins)*
- *Terminal positions $T \subseteq P$ (where game can end)*
- *Set of directed edges E_A between states (A's moves)*
- *set of directed edges E_B between states (B's moves)*
- *Utility or payoff function $U : T \rightarrow \mathbb{R}$ (how good is each terminal state for player A)*

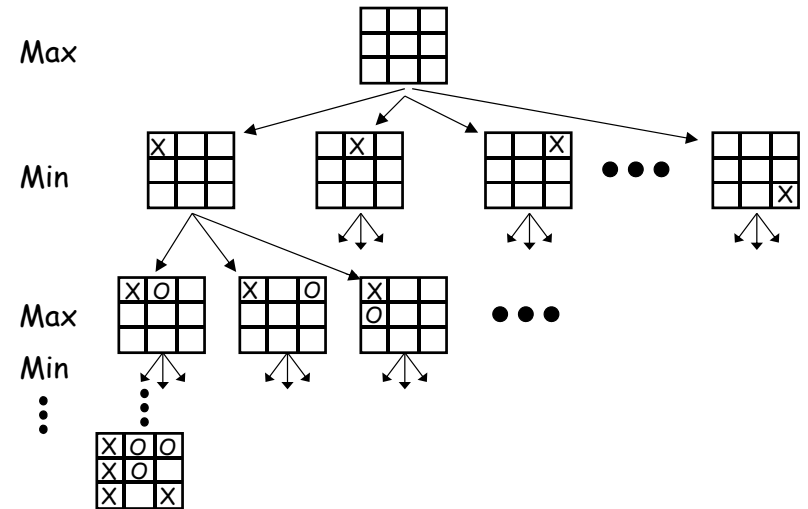 - *Why don't we need a utility function for B?*

Two-Player Zero-Sum Game – Intuition

- *Players alternate moves (starting with Max)*
  - *Game ends when some terminal $p \in T$ is reached*

- *A game **state**: a state-player pair*
  - *Tells us what state we're in and whose move it is*

- *Utility function and terminals replace goals*
  - *Max wants to maximize the terminal payoff*
  - *Min wants to minimize the terminal payoff*

- *Think of it as:*
  - *Max gets U(t), Min gets –U(t) for terminal node t*
  - *This is why it's called zero (or constant) sum*

Tic Tac Toe States

Turn=Max(X)   Turn=Min(O)   Turn=Max(X)

start →

Min(O)

X X X
  O
    O

U = +1

terminal

another terminal

Max(X)

O X X
O   X
O

U = -1

Tic Tac Toe Game Tree

Max

Min

Max

Min

Game Tree

- *Game tree looks like a search tree*
  - *Layers reflect alternating moves between A and B*
  - *The search tree in game playing is a subtree of the game tree*
- *Player A doesn't decide where to go alone*
  - *After A moves to a state, B decides which of the states children to move to*
- *Thus A must have a strategy*
  - *Must know what to do for each possible move of B*
  - *One sequence of moves will not suffice: "What to do" will depend on how B will play*

- *What is a reasonable strategy?*

Minimax Strategy

- *Assume that the other player will always play their best move,*
  - *you always play a move that will minimize the payoff that could be gained by the other player.*
  - *My minimizing the other player's payoff you maximize yours.*
- *If however you know that Min will play poorly in some circumstances, there might be a better strategy than MiniMax (i.e., a strategy that gives you a better payoff).*

- *But in the absence of that knowledge minimax "plays it safe"*

Minimax Strategy payoffs
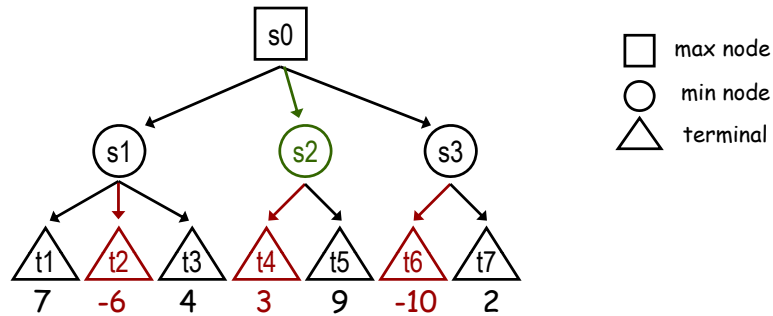
The terminal nodes have utilities.
But we can compute a "utility" for the non-terminal states, by assuming both players always play their best move.

Minimax Strategy – Intuitions
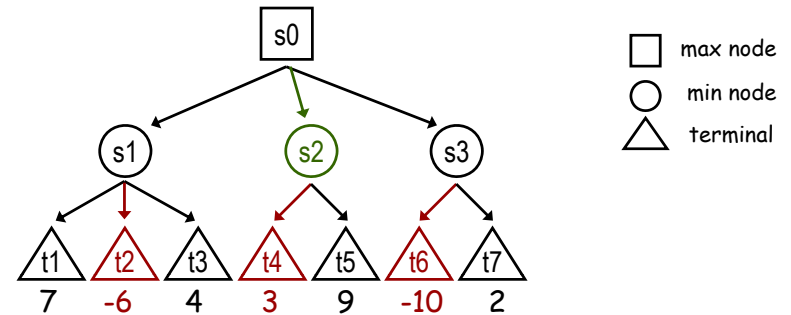
If Max goes to s1, Min goes to t2, U(s1) = min{U(t1), U(t2), U(t3)} = -6
If Max goes to s2, Min goes to t4, U(s2) = min{U(t4), U(t5)} = 3
If Max goes to s3, Min goes to t6, U(s3) = min{U(t6), U(t7)} = -10

So Max goes to s2: so U(s0) = max{U(s1), U(s2), U(s3)} = 3

Minimax Strategy

- *Build full game tree (all leaves are terminals)*
  - *Root is start state, edges are possible moves, etc.*
  - *Label terminal nodes with utilities*

- *Back values up the tree*
  - *U(t) is defined for all terminals (part of input)*
  - *U(n) = min {U(c) : c is a child of n} if n is a Min node*
  - *U(n) = max {U(c) : c is a child of n} if n is a Max node*

Minimax Strategy

- *The values labeling each state are the values that Max will achieve in that state if both Max and Min play their best moves.*
  - *Max plays a move to change the state to the highest valued min child.*
  - *Min plays a move to change the state to the lowest valued max child.*
- *If Min plays poorly, Max could do better, but never worse.*
  - *If Max, however knows that Min will play poorly, there might be a better strategy of play for Max than Minimax.*

Depth-First Implementation of Minimax

- *Building the entire game tree and backing up values gives each player their strategy.*
- *However, the game tree is exponential in size.*
- *Furthermore, as we will see later it is not necessary to know all of the tree.*

- *To solve these problems we find a **depth-first** implementation of minimax.*

  *We run the depth-first search after each move to compute what is the next move for the **MAX** player. (We could do the same for the MIN player).*
- *This avoids explicitly representing the exponentially sized game tree: we just compute each move as it is needed.*

Depth-First Implementation of Minimax
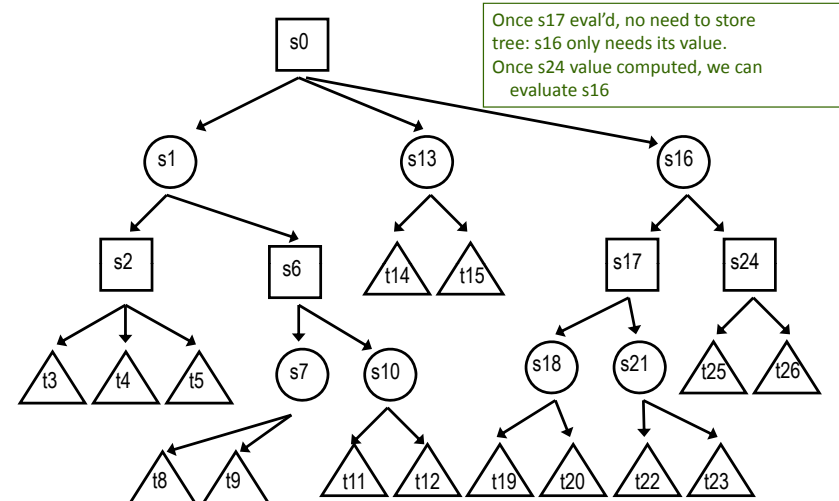
```
DFMiniMax(n, Player) //return Utility of state n given that
                     //Player is MIN or MAX
  If n is TERMINAL
     Return U(n) //Return terminal states utility
                 //(U is specified as part of game)

                 //Apply Player's moves to get
                 //successor states.
  ChildList = n.Successors(Player)
  If Player == MIN
     return minimum of DFMiniMax(c, MAX) over c ∈ ChildList
  Else //Player is MAX
     return maximum of DFMiniMax(c, MIN) over c ∈ ChildList
```
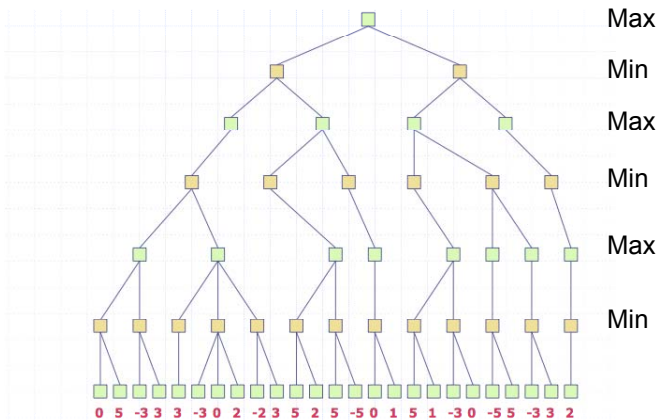
Depth-First Implementation of Minimax

- *Notice that the game tree has to have finite depth for this to work*

- *Advantage of DF implementation: space efficient*

- *Minimax will expand $O(b^d)$ states, which is both a BEST and WORSE case scenario.*
  - *We must traverse the entire search tree to evaluate all options*
  - *We can't be lucky as in regular search and find a path to a goal before searching the entire tree.*

Visualization of Depth-First Minimax

Once s17 eval'd, no need to store tree: s16 only needs its value.
Once s24 value computed, we can evaluate s16

Example

Max
Min
Max
Min
Max
Min

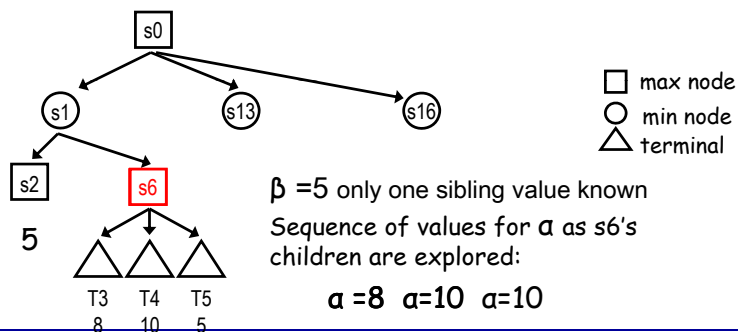0 5 -3 3 3 -3 0 2 -2 3 5 2 5 -5 0 1 5 1 -3 0 -5 5 -3 3 2

Pruning

- *It is not necessary to examine entire tree to make correct Minimax decision*
- *Assume depth-first generation of tree*
  - *After generating value for only some of n's children we can prove that we'll never reach n in a Minmax strategy.*
  - *So we needn't generate or evaluate any further children of n!*
- *Two types of pruning (cuts):*
  - *pruning of max nodes (α-cuts)*
 - *pruning of min nodes (β-cuts)*

Cutting Max Nodes (Alpha Cuts)

- *At a Max node n:*
  - *Let β be the lowest value of n's siblings examined so far (siblings to the left of n that have already been searched)*
 - *Let a be the highest value of n's children examined so far (changes as children examined)*

□ max node
○ min node
△ terminal

β =5 only one sibling value known

Sequence of values for α as s6's children are explored:

α =8   α=10   α=10

Cutting Max Nodes (Alpha Cuts)

- *If a becomes ≥ β we can stop expanding the children of n*
  - *Min will never choose to move from n's parent to n since it would choose one of n's lower valued siblings first.*


min node

P   β = 8

14   12   8   n   α = 2 / 4 / 9

2   s1   4   s2   9   s3
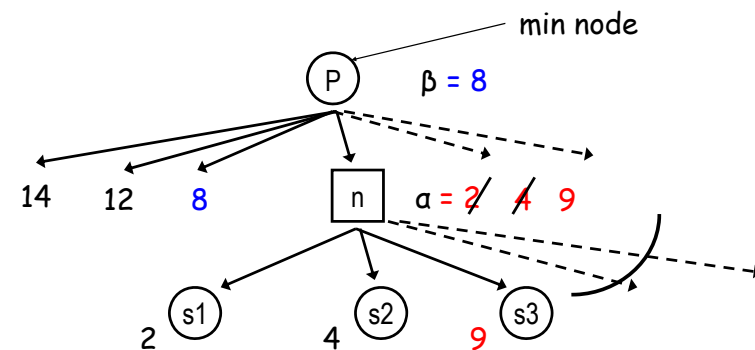
Cutting Min Nodes (Beta Cuts)

- *At a Min node n:*
  - *Let β be the lowest value of n's children examined so far (changes as children examined)*
  - *Let a be the highest value of n's sibling's examined so far (fixed when evaluating n)*

□ max node
○ min node
△ terminal

s0

s1   s13   s16

α =10

s2   s6

β =5   β =3
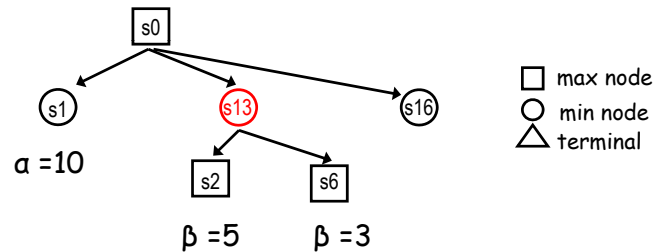
Cutting Min Nodes (Beta Cuts)

- *If β becomes ≤ a we can stop expanding the children of n.*
  - *Max will never choose to move from n's parent to n since it would choose one of n's higher value siblings first.*

P   α = 7

6   2   7   n   β = ~~9~~ ~~5~~ 3

s1   s2   s3

9   8   3
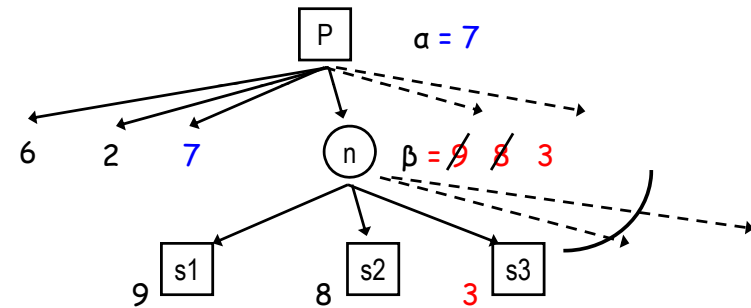
Implementing Alpha-Beta Pruning

```
AlphaBeta(n,Player,alpha,beta) //return Utility of state
  If n is TERMINAL
    return U(n) //Return terminal states utility
  ChildList = n.Successors(Player)
  If Player == MAX
    for c in ChildList
        alpha = max(alpha, AlphaBeta(c,MIN,alpha,beta))
        If beta <= alpha
            break
    return alpha
  Else //Player == MIN
    for c in ChildList
        beta = min(beta, AlphaBeta(c,MAX,alpha,beta))
        If beta <= alpha
            break
    return beta
```
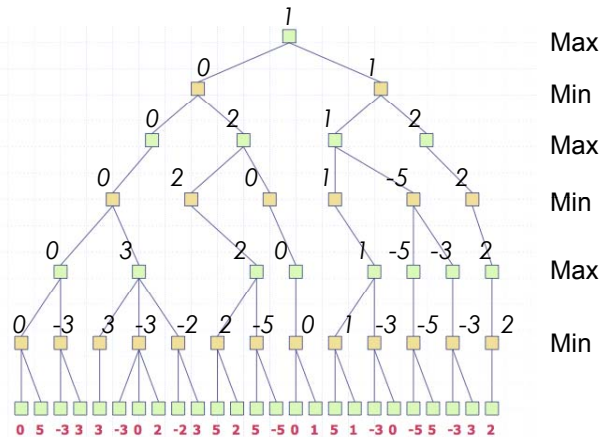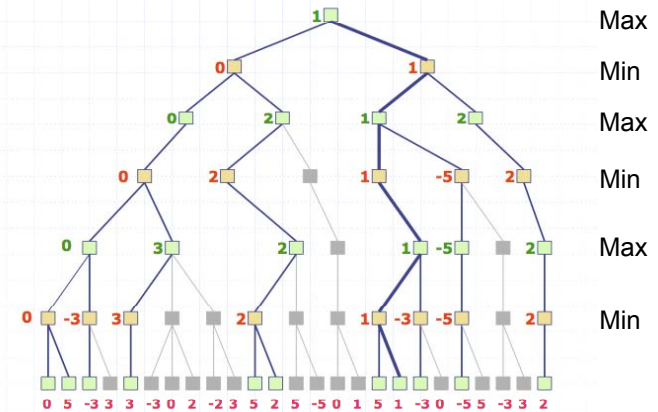
Implementing Alpha-Beta Pruning

```
Initial call

AlphaBeta(START_NODE,Player,-infinity,+infinity)
```

Example

Which computations could we have avoided here? Assuming we expand nodes left to right?

Example

Effectiveness of Alpha-Beta Pruning

- *With no pruning, you have to explore $O(b^d)$ nodes, which makes the run time of a search with pruning the same as plain Minimax.*

- *If, however, the move ordering for the search is optimal (meaning the best moves are searched first), the number of nodes we need to search using alpha beta pruning is $O(b^{d/2})$. That means you can, in theory, search twice as deep!*

- *In Deep Blue, they found that alpha beta pruning meant the average branching factor at each node was about 6 instead of 35.*

Rational Opponents

- *May want to compute your full strategy ahead of time.*
  - *you must store "decisions" for each node you can reach by playing optimally*
  - *if your opponent has unique rational choices, this is a single branch through game tree*
  - *if there are "ties", opponent could choose any one of the "tied" moves: must store strategy for each subtree*
  - *In general space is an issue.*
  - *Alternatively you compute your next move a fresh at each stage.*

Practical Matters

- *All "real" games are too large to enumerate tree*
  - *e.g., chess branching factor is roughly 35*
  - *Depth 10 tree: 2,700,000,000,000,000 nodes*
  - *Even alpha-beta pruning won't help here!*

- *We must limit depth of search tree*
  - *Can't expand all the way to terminal nodes*
  - *We must make heuristic estimates about the values of the (non-terminal) states at the leaves of the tree*
  - *These heuristics are often called evaluation function*
  - *evaluation functions are often learned*

Heuristics in Games

- *Example for tic tac toe: h(n) = [# of 3 lengths that are left open for player A] - [# of 3 lengths that are left open for player B].*

- *Alan Turing's function for chess: h(n) = A(n)/B(n) where A(n) is the sum of the point value for player A's pieces and B(n) is the sum for player B.*

- *Most evaluation functions are specified as a weighted sum of features: $h(n) = w_1*feature_1(n) + w_2*feature_2(n) + ... w_i*feature_i(n)$.*

- *Deep Blue used about 6000 features in its evaluation function.*

Heuristics in Games

- *Think of a few games and suggest some heuristics for estimating the "goodness" of a position*
  - *Chess?*
  - *Checkers?*
  - *Your favorite video game?*

An Aside on Large Search Problems

- *Issue: inability to expand tree to terminal nodes is relevant even in standard search*
  - *Often we can't expect A* to reach a goal by expanding full frontier*
  - *So we often limit our look-ahead, and make moves before we actually know the true path to the goal*
  - *Sometimes called online or realtime search*

- *In this case, we use the heuristic function not just to guide our search, but also to commit to moves we actually make*
  - *In general, guarantees of optimality are lost, but we reduce computational/memory expense dramatically*
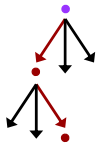
Realtime Search Graphically
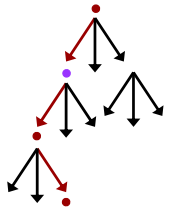
1. *We run A* (or our favorite search algorithm) until we are forced to make a move or run out of memory. Note: no leaves are goals yet.*

2. *We use evaluation function f(n) to decide which path looks best (let's say it is the red one).*

3. *We take the first step along the best path (red), by actually making that move.*

4. *We restart search at the node we reach by making that move. (We may actually cache the results of the relevant part of first search tree if it's hanging around, as it would with A*).*