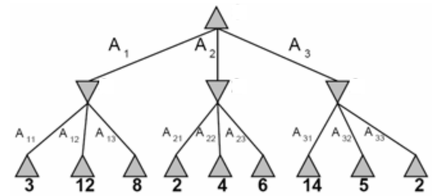
CS 188: Artificial Intelligence
Spring 2006

Lecture 25: Games II
4/20/2006

Dan Klein – UC Berkeley

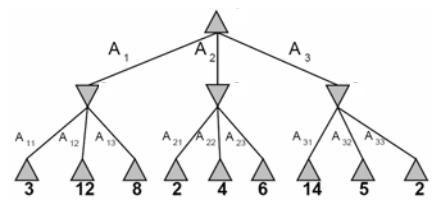
Recap: Minimax Trees



Minimax Search

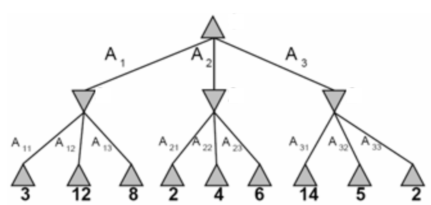
```
function MAX-VALUE(state) returns a utility value
   if TERMINAL-TEST(state) then return UTILITY(state)
   v ← −∞
   for a, s in SUCCESSORS(state) do v ← MAX(v, MIN-VALUE(s))
   return v

function MIN-VALUE(state) returns a utility value
   if TERMINAL-TEST(state) then return UTILITY(state)
   v ← ∞
   for a, s in SUCCESSORS(state) do v ← MIN(v, MAX-VALUE(s))
   return v
```

DFS Minimax



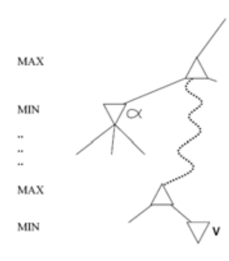
α-β Pruning Example

- [Code in book]



α-β Pruning

- General configuration
  - α is the best value (to MAX) found so far off the current path
  - If V is worse than α, MAX will avoid it, so prune V's branch
  - Define β similarly for MIN

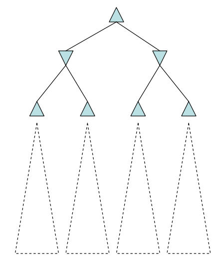


α-β Pruning Properties

- Pruning has no effect on final result
- Good move ordering improves effectiveness of pruning
- With "perfect ordering":
  - Time complexity drops to $O(b^{m/2})$
  - Doubles solvable depth
  - Full search of, e.g. chess, is still hopeless!
- A simple example of metareasoning, here reasoning about which computations are relevant

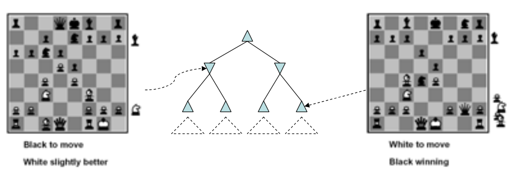
Resource Limits

- Cannot search to leaves
- Limited search
  - Instead, search a limited portion of the tree
  - Replace terminal utilities with an eval function for non-terminal positions
- Guarantee of optimal play is gone
- Example:
  - Suppose we have 100 seconds, can explore 10K nodes / sec
  - So can check 1M nodes per move
  - α-β reaches about depth 8 – decent chess program



Evaluation Functions

- Function which scores non-terminals



- Ideal function: returns the utility of the position
- In practice: typically weighted linear sum of features:

$$Eval(s) = w_1 f_1(s) + w_2 f_2(s) + \ldots + w_n f_n(s)$$

- e.g. $f_1(s)$ = (num white queens – num black queens), etc.

Function Approximation

- Problem: inefficient to learn each state's utility (or eval function) one by one
- Solution: what we learn about one state (or position) should generalize to similar states
  - Very much like supervised learning
  - If states are treated entirely independently, we can only learn on very small state spaces

Linear Value Functions

- Another option: values are linear functions of features of states (or action-state pairs)

$$\hat{U}_\theta(s) = \sum_k \theta_k f_k(s)$$

  - Good if you can describe states well using a few features (e.g. for game playing board evaluations)
- Now we only have to learn a few weights rather than a value for each state

3	0.812	0.868	0.912	+1
2	0.762		0.660	−1
1	0.705	0.655	0.611	0.388
	1	2	3	4

3	0.80	0.85	0.90	0.95
2	0.70		0.80	0.85
1	0.60	0.65	0.70	0.75
	1	2	3	4

$$\hat{U}_\theta(s) = 0.3 + 0.05x + 0.1y$$

Recap: Model-Free Learning

- Recall MDP value updates for a given estimate of U
  - If you know the model T, use Bellman update

$$U(s) \leftarrow R(s) + \gamma \max_a \sum_{s'} U(s')T(s,a,s')$$

- Temporal difference learning (TD)
  - Make (epsilon greedy) action choice (or follow a provided policy)

$$\pi(s) = \arg\max_a \sum_{s'} U(s')T(s,a,s')$$

  - Update using results of the action

$$U(s) \leftarrow (1-\alpha)U(s) + \alpha\left(R(s) + \gamma U(s')\right)$$

Example: Tabular Value Updates

- Example: Blackjack
  - +1 for win, -1 for loss or bust, 0 for tie
  - Our hand shows 14, current policy says "hit"
  - Current U(s) is 0.5
  - We hit, get an 8, bust (end up in s' = "lose")
- Update
  - Old U(s) = 0.5
  - Observed R(s) = 0
  - Old U(s') = -1
  - New U(s) = U(s) + α [γ (R(s) + U(s') – U(s)]
  - If α = 0.1, γ = 1.0
  - New U(s) = 0.5 + 0.1 [0 + -1 – 0.5]
    = 0.5 + 0.1 [-1.5] = 0.35

TD Updates: Linear Values

- Assume a linear value function:

$$\hat{U}_\theta(s) = \sum_k \theta_k f_k(s)$$

- Can almost do a TD update:

$$U(s) \leftarrow U(s) + \alpha\big([R(s) + \gamma U(s')] - U(s)\big)$$

  - Problem: we can't "increment" U(s) explicitly
  - Solution: update the weights of the features at that state

$$\theta_k \leftarrow \theta_k + \alpha\big([R(s) + \gamma U(s')] - U(s)\big) f_k(s)$$

Learning Eval Parameters with TD

- Ideally, want eval(s) to be the utility of s
- Idea: use techniques from reinforcement learning
  - Samuel's 1959 checkers system
  - Tesauro's 1992 backgammon system (TD-Gammon)
- Basic approach: temporal difference updates
  - Begin in state s
  - Choose action using limited minimax search
  - See what opponent does
  - End up in state s'
  - Do a value update of U(s) using U(s')
  - Not guaranteed to converge against an adversary, but can work in practice

Q-Learning

- With TD updates on values
  - You don't need the model to update the utility estimates
  - You still do need it to figure out what action to take!
- Q-Learning with TD updates
  - No model needed to learn or to choose actions

$$Q_{i+1}(a,s) \leftarrow (1-\alpha)Q_i(a,s) + \alpha\left(R(s) + \gamma \max_{a'} Q_i(a',s')\right)$$

$$\pi(s) = \arg\max_a Q(a,s)$$

TD Updates for Linear Qs

- Can use TD learning with linear Qs
  - (Actually it's just like the perceptron!)
  - Old Q-learning update:

$$Q(a,s) \leftarrow Q(a,s) + \alpha\left(R(s) + \gamma \max_{a'} Q(a',s') - Q(a,s)\right)$$

  - Simply update weights of features in Q_θ(a,s)

$$\theta_k \leftarrow \theta_k + \alpha\left(R(s) + \gamma \max_{a'} Q_\theta(a',s') - Q_\theta(a,s)\right) f_k(a,s)$$

Coming Up

- Real-world applications
  - Large scale machine / reinforcement learning
  - NLP: language understanding and translation
  - Vision: object and face recognition