13. *-Minimax

Introduction

- Perfect information deterministic games?
  - Lots of success…
- Imperfect information games?
  - Hard because of missing information
  - Subject of active research
  - Bridge, poker
- Non-deterministic/stochastic games?
  - Successes, but using methods unique to each application domain
  - Backgammon, Scrabble

Stochastic Games

- Non-determinism
  - Roll of the dice or deal of cards
- Minimax search trees but with the added complication of *chance* nodes
  - Search-based approaches must take into account all possibilities at a chance node
  - Increases the branching factor making deep search unlikely
- Hence, many game-playing program rely less on search and more on knowledge

Deep Search!?

- Deep “brute-force” search has been effective in deterministic, perfect-information domains.
- Deep search has also been useful in some imperfect information domains and some stochastic games (e.g., sampling, rollouts).
- Can deep full-width search be effective in stochastic domains?

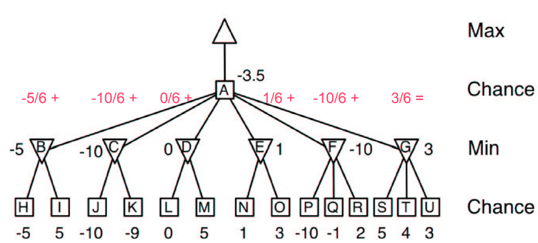
Deep Search!?

- Two-player
  deterministic
  perfect information search
  has minimax as a starting point and…
- Two-player
  stochastic
  perfect information search
  has expectimax as a starting point.

Expectimax

```
float Expectimax(Board board, int depth, int is_max_node) {
   if(terminal(board) || depth == 0) return (evaluate(board));

   N = numChanceEvents(board);
   sum = 0;
   for(i = 1; i <= N; i++) {
      succ = applyChanceEvent(board,i);
      sum += eventProb(board,i) *
             search(succ, depth-1, is_max_node);
   }

   return (sum);
}
```

Expectimax



*-MiniMax

- Need to add Alpha-beta-like cutoffs to an Expectimax search
- Idea proposed by Bruce Ballard (1983)
  - Family of *-Minimax algorithms
- The idea seems to have been forgotten…
  - No implementations in the literature
  - No follow-up research
  - Few references to Ballard's work, other than the occasional mention that *-Minimax exists

*-Minimax: Cutoffs

- Leave Max and Min nodes alone in an alpha-beta search framework
- Add cutoffs to Chance nodes
- Assume that all branches not searched have the worst-case result

- L = lowest value achievable (-10)
- U = highest value achievable (10)

*-Minimax: Cutoffs

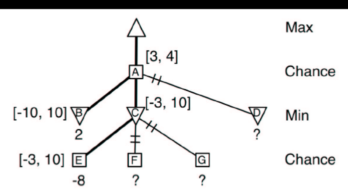
- Alpha cutoff:

$$\frac{1}{N}(\underbrace{(V_1 + \ldots + V_{i-1})}_{\text{Values seen}} + \underbrace{V_i}_{\text{Current value}} + \underbrace{U \times (N-i)}_{\text{Values to come}}) \leq alpha$$

- Beta cutoff:

$$\frac{1}{N}(\underbrace{(V_1 + \ldots + V_{i-1})}_{\text{Values seen}} + \underbrace{V_i}_{\text{Current value}} + \underbrace{L \times (N-i)}_{\text{Values to come}}) \geq beta$$

*-Minimax: Search Windows



- Alpha-beta bounds passed to C:

$$\frac{1}{3}(2 + V_i + (1) \times L) \geq beta \Rightarrow V_i \geq 20$$

$$\frac{1}{3}(2 + V_i + (1) \times U) \leq alpha \Rightarrow V_i \geq -3$$

*-Minimax: Incremental Updates

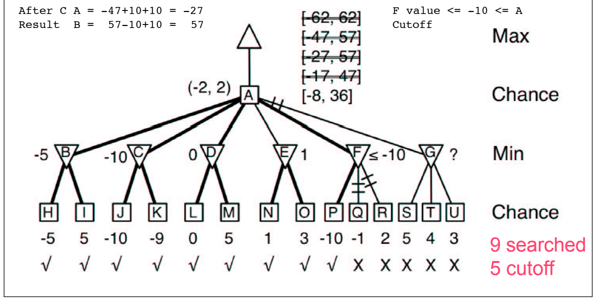
- Observation:
  - –Alpha bound check starts with the highest possible value (all V_i are unknown and thus equal to U).
  - –As each V_i becomes available, the best the player can do is improved.
  - –When the best possible score is proven not to be able to exceed alpha, cutoff.
  - –Similar for beta cutoffs
- Incrementally update alpha and beta tests

*-MiniMax: Star1

```
float Star1(Board board, float alpha, float beta, int depth) {
   if(terminal(board) || depth == 0) return (evaluate(board));
   N = numSuccessors(board);
   A = N*(alpha-U) + U;
   B = N*(beta-L) + L;
   vsum = 0;
   for(i = 1; i <= N; i++) {
      AX = max(A, L);
      BX = min(B, U);
      v = search(successor(board,i), AX, BX, depth-1);
      if(v <= A) return (alpha);
      if(v >= B) return (beta);
      vsum += v;
      A += U - v;
      B += L - v;
   }
   return (vsum/N);
}
```

Star1 Example

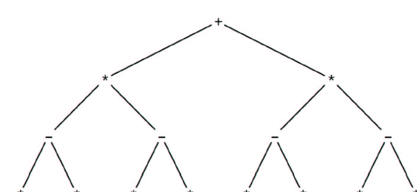
```
Initial A = 6(-2-10)+10 = -62
Window  B = 6( 2+10)-10 =  62

After B A = -62+10+ 5 = -47
Result  B =  62-10+ 5 =  57

After C A = -47+10+10 = -27
Result  B =  57-10+10 =  57

After D A = -27+10-0 = -17
Result  B =  57-10+0 =  47

After E A = -17+10-1 =  -8
Result  B =  47-10-1 =  36

F value <= -10 <= A
Cutoff
```



Star1 Observations

- Star1 is pessimistic
  - Always assumes the worst case.
- Star1 is agnostic
  - Does not know what type of node will follow the current node.
  - Even if it did, it cannot take advantage of it.
- For most games, the search tree is a regular structure.
- Can we exploit this?

Regular *-Minimax Tree



Backgammon has a regular tree structure: Max node (+), Chance node (*), Min node (-), Chance node (*), and repeat.

*-Minimax: Star2

- Star1 searches each successor completely before moving to the next one.
- A successor could be very good or very bad, and this information might be easy to obtain.
  - If we had this information, the search bounds could be tightened more quickly.
- Star2 introduces probing: do a quick look at all successors to bound their score

Star2 (part1)

```
float Star2_Min(Board board, float alpha, float beta, int depth) {
   if(terminal(board) || depth == 0) return (evaluate(board));
   N = numSuccessors(board);
   /* Initialization */
   A = N*(alpha-U);
   B = N*(beta-L);
   BX = min(B, U);
   /* Probing phase */
   for(i = 1; i <= N; i++) {
      A += U;
      AX = max(A, L);
      w[i] = Probe_Min(successor(board,i), AX, BX, depth-1);
      if(w[i] <= A) return (alpha);
      A -= w[i];
   }
```

Do a quick look at all children to get a bound on their score. Save the results in w[] so that they do not have to be repeated.

Star2 (part 2)

```
   /* Search phase */
   vsum = 0;
   for(i = 1; i <= N; i++) {
      A += w[i];
      B += L;
      AX = max(A, L);
      BX = min(B, U);
      v = search(successor(board,i), AX, BX, depth-1);
      if(v <= A) return (alpha);
      if(v >= B) return (beta);
      vsum += v;
      A -= v;
      B -= v;
   }
   return (vsum/N);
}
```

If no cutoff has occurred, then search as in Star1 (but making use of the probe results).

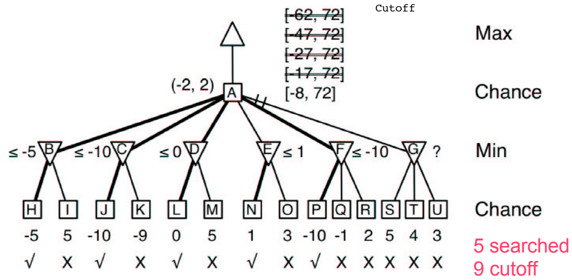
Probing

```
float Probe_Min(Board board, float alpha, float beta, int depth) {
   if(terminal(board) || depth == 0) return (evaluate(board));
   choice = PickSuccessor(board);
   return (Star2_Max(successor(board,choice), alpha, beta, depth-1));
}
```

The simplest probing function is to search one child of each successor.
Need heuristic knowledge to choose the "best" candidate to expand.

Star2 Example (1)

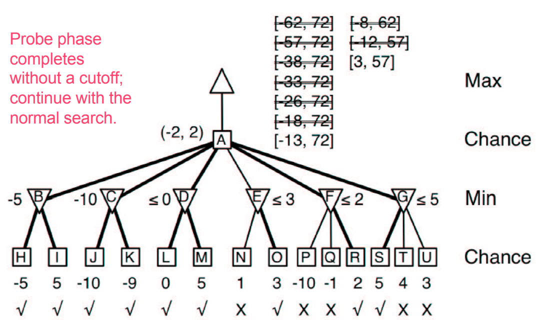
Initial A = 6(-2-10)+10 = -62
After B A = -62+10+ 5 = -47
After C A = -47+10+10 = -27
After D A = -27+10+ 0 = -17
After E A = -17+10- 1 = -8
F value <= -10 <= A
Cutoff



Star2 Comments

- With increased branching factor, Star2 becomes more effective, but it can do well with small branching factors.
- PickSuccessor function needs to be fast and effective.
- Star2 is not guaranteed to work better than Star1; it depends on the quality of the probing.

Star2 Example (2)



Comments

- Transposition table can be a big win (eliminating repeating the probe searches).
- Iterative deepening then becomes practical.
- Can use the equivalent of a fail-soft enhancement to get slightly better results.
- Star2.5: use a more sophisticated probing scheme.

*-Minimax Performance Results?

- *-Minimax has been known for over 20 years but…
- Other than Ballard's original experiments, there are no published performance numbers on the algorithm
- Ballard's results used shallow search depths and no search enhancements
- How would *-Minimax perform in a real game-playing program?

Game of Dice

- Toy domain used to better understand *-Minimax performance
- Rules:
  - –NxN board
  - –Win by forming an M-in-a-row line (H,V,D)
  - –Roll of an N-sided die tells you the column (1st player) or row (2nd player) to play in
  - –Player chooses the move to maximize their chances of winning

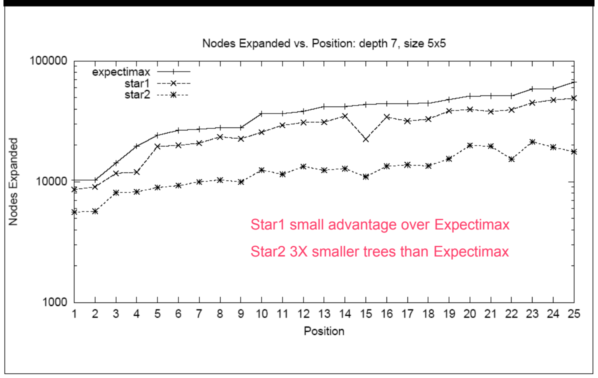
Game of Dice

- Game tree is a regular *-Minimax tree
- Chance nodes have an equal probability of taking on each of N values
- Variable branching factor (0 to N)
- Simple evaluation function based on the number and size of partial lines on board
- Deeper search should be correlated with stronger play

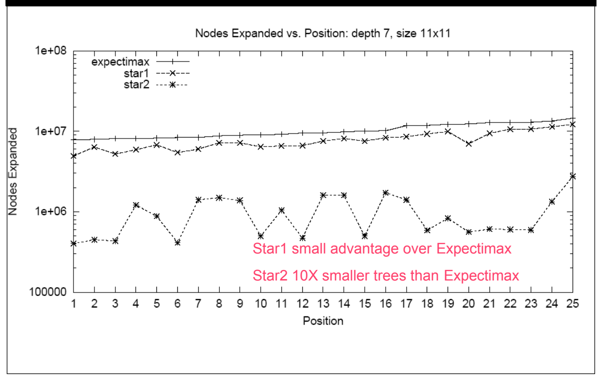
Search Depth

- Game tree starts off with a max node
- Count each Max, Min, and Chance node as a ply
- Thus, a depth 3 tree is a Max, Chance, Min node
- A depth 7 tree has two moves by each player

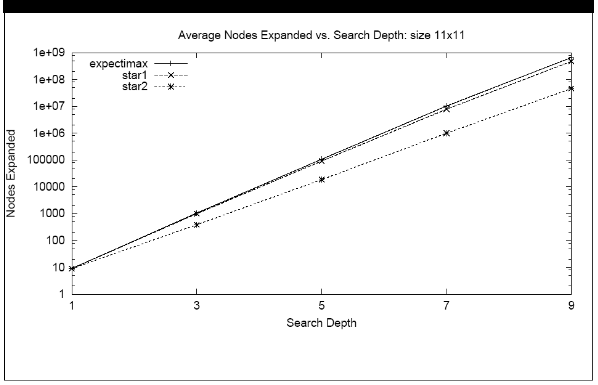
Dice: 5x5 Board (Depth 7)



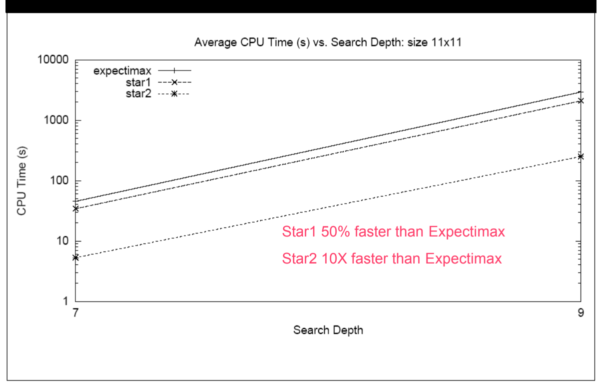
Dice: 11x11 Board (Depth 7)



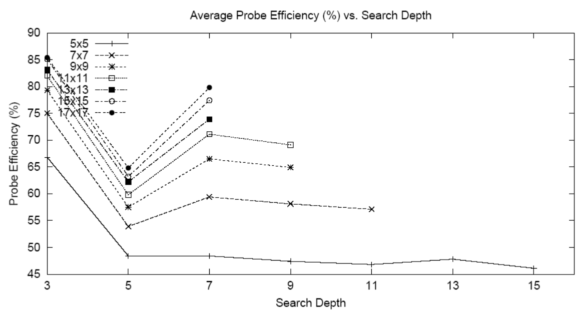
Dice: Search Depth (11x11)



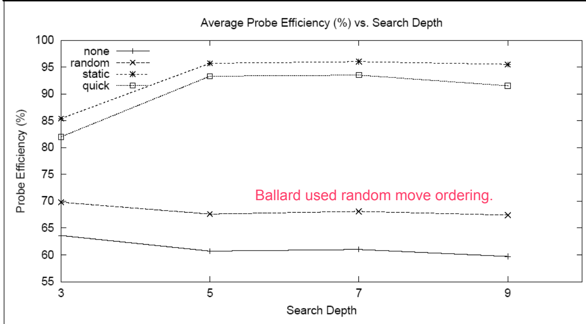
Dice: Time (11x11)



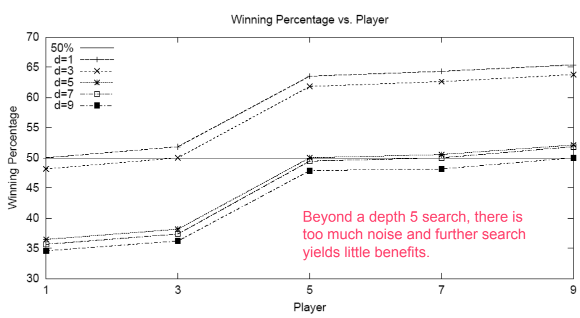
Dice: Probe Efficiency



Dice: Move ordering

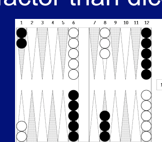


Dice: Tournament



Backgammon

- Backgammon was the original motivation for this work.
- Can deep search improve the performance of backgammon programs?
- Two die (non-uniform probabilities)
- Larger branching factor than dice
- 2^{20} search space



Backgammon Programs

- Hans Berliner's BKG 9.8
- Gerry Tesauro's Neurogammon and TD-Gammon
- Tesauro clones: Jellyfish, Snowie, GNU backgammon
- The top backgammon programs are likely stronger than the human world champion

Winning Recipe

- Modern programs uses a neural-net-based evaluation function tuned using temporal-difference learning
- Little search
  - Cost of an evaluation is very high
  - Usually only 1-ply search
  - GUNbg has a tournament mode that does a selective 5-ply search

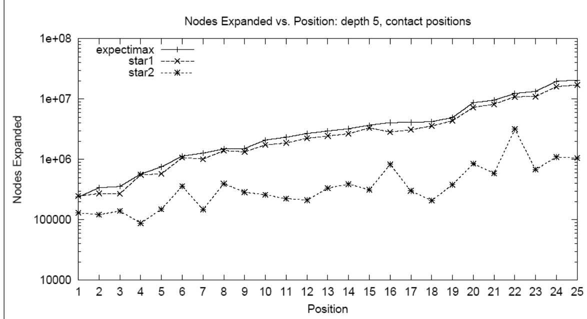
Non-uniform Chance Nodes

$$\frac{(V_1 + \ldots + V_{i-1}) + V_i + U \times (N - i)}{N} \leq alpha$$

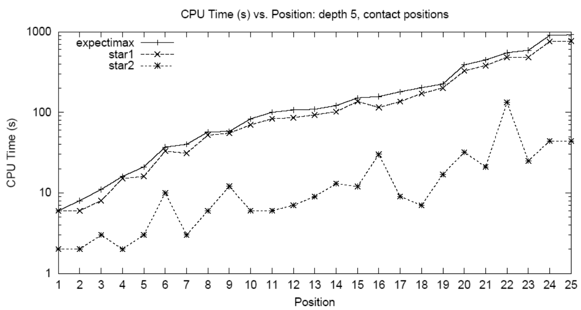
$$(P_1 \times V_1 + \ldots + P_{i-1} \times V_{i-1}) + P_i \times V_i + U \times (1 - P_1 - \ldots - P_i) \leq alpha$$

$$A_i = \frac{alpha - U \times (1 - P_1 - \ldots - P_i) - (P_1 \times V_1 + \ldots + P_{i-1} \times V_{i-1})}{P_i}$$

Backgammon: Nodes (Depth 5)



Backgammon: Time (Depth 5)



Backgammon: Time

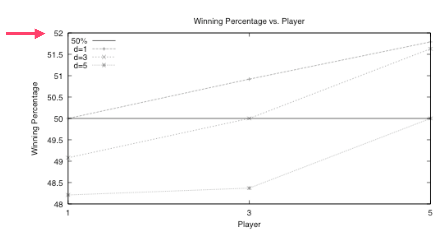
- Average Time over 500 positions

	Expectimax		Star1			Star2		
	μ	σ	μ	σ	%	μ	σ	%
d=3	1.1	0.7	1.1	0.6	100	1.0	0.1	91
d=5	315.0	566.8	258.6	472.6	82	21.0	36.9	7

- Probe Efficiency

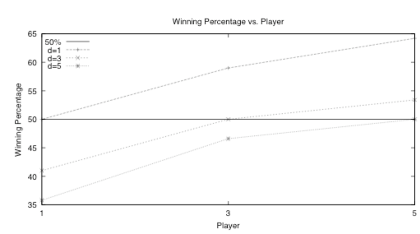
	d=3		d=5	
	μ	σ	μ	σ
contact	68.9%	29.5%	64.2%	22.6%

Backgammon: Tournament



Deeper search yields little performance benefits!?

Backgammon: Adding Noise



With a small amount of noise added to the evaluation function, then deeper search yields significant differences.

Conclusion: GNUbg's evaluation function is excellent!

Conclusions

- Expectimax < Star1 << Star2
- In some stochastic domains, search can only take you so far (depth 5)
- Full-width depth=5 search is possible in backgammon in real-time
- Backgammon programs have near-oracle evaluation functions!
- Other games: Carcassonne, Paris-Paris

Conclusion

Ballard’s work deserves to be better known!