# 5. Iterative Deepening and Move Ordering

Jonathan Schaeffer

jonathan@cs.ualberta.ca

www.cs.ualberta.ca/~jonathan

## Depth-first Search

- How do you know how deep to search?
- Search to terminal nodes?
  - Could be too deep!
- Search to a fixed depth?
  - Bad move ordering could make this a big search
  - What if the search depth is set too large?
  - What if the search depth is set too small?

## Iterative Deepening

- Iterate on the search depth
- Search to depth 1, then 2, then 3, until resources run out
- The advantage is that you get the deepest possible search depth given the resource constraints
  - Nice property for real-time search [1]
- The disadvantage is wasted search

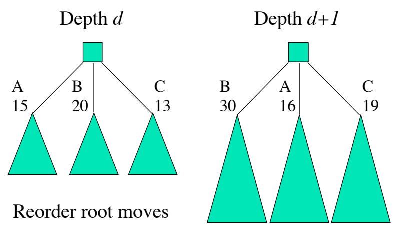
## Disadvantage: Extra Search?

- ID seems impractical because of all the repeated work:
  - Assume a growth rate of *b* and search depth *d*
  - Size = $b + b^2 + b^3 + \ldots + b^{d-1} + b^d$
  - Aren't all the early iterations wasted search?

## Advantages!

- Before searching to depth *d+1*, order the moves at the root based on the scores returned from depth *d*
- This assumes that the best move at depth *d* is a good predictor of the best move at depth *d+1*
  - For most domains, this is true
- Increases the likelihood that the best move is searched first on the last (and most expensive) iteration

## Example



## More Advantages!

- Much of the ID overhead can be eliminated by using a transposition table
- Use TT to increase likelihood that the best move is searched first at all nodes!
- Use the TT to save the best move in a position
- When the position is revisiting (e.g., on the next iteration), use the previous best move as the first move to consider
  - Likely best on the newer, deeper search

## TT Move Ordering

```
int AlphaBeta( state s, int alpha, int beta, int depth ) {
    if( terminal node || depth == 0 ) return( Evaluate( s );
    /* Look in TT before searching */
    ptr = TTLookup( s );
    if( ptr != NULL && ptr->depth >= d ) {
            …
    }
    if( ptr != NULL ) {     /* Note that ptr->depth can be < depth */
            /* move ptr->bestmove to head of successors list */
    }
    …
}
```

## Extra Search Revisited

  - Size = $b + b^{2} + b^{3} + \ldots + b^{d-1} + b^{d}$
  - Aren't all the early iterations wasted search?
- No!!
- Improved move ordering throughout the search!
- *Invest* in the early iterations to improve the last (most expensive) iteration

9/9/02 9

## Move Ordering

- Alpha-beta's success hinges on searching the "best" move first
- TT can provide a candidate best move
- What do we do if…
  - No matching TT entry?
  - TT best move does not cause a cutoff?
- How do we order the remaining moves?

9/9/02 10

## Knowledge

- Common to use application-dependent knowledge heuristics
  - Chess: consider checking and capture moves first
  - Constraint problems: consider branches that address the most tightly constrained component first
- Finding knowledge can be hard

9/9/02 11

## Discovery from the Search

- TT best move ordering was nice because it is not application dependent
  - Knowledge discovered dynamically during the search
- Is there a correlation between a property of the application and move ordering?

9/9/02 12

## History Heuristic

- In an position *p*, move *m* is best (highest score or causes a cutoff)
- Move *m* now has a *history* of being a good move
- In a new position *q*, if move *m* is legal, prefer to try moves with a history of success (albeit in a different setting)

## History Heuristic

- Maintain a table of all possible moves
- When leaving a node, update the history score of the best move
- When entering a node, sort moves based on their history heuristic score
- Score should reflect the search depth (deeper search means more meaningful result)
- HT[ m ] += ( 1 << depth ) /* $2^{depth}$ */

## History Heuristic

```
int AlphaBeta( state s, int alpha, int beta, int depth ) {
    …
    /* Move ordering -- just HH scores */
    for( child = 1; child <= NumbSuccessors( s ); child++ )
          score[ child ] = HH[ Successor( s, child ) ];
    Sort( score );
    …
    /* Search moves in order of their scores */
    …
    HH[ bestmove ] += ( 1 << depth );
    …
}
```

## History Heuristic

- Simple form of learning
- Little context -- just a move
- You can add more context
  - Better “accuracy”
  - Finer granularity
- The extreme case is adding all the context, in which case you get the entire position

## Move Ordering

- Try TT move first
- Try knowledge next (static)
- Try search-based knowledge (dynamic)
  - History heuristic
  - Countermove heuristic
  - Inertia heuristic
  - Neural move-map heuristic
  - …
- HH is simple to implement, low CPU and space overhead, and effective in many domains

## Move Ordering Effectiveness

- The ideal case is to consider only one move at a CUT node
- Extensive experiments in chess

| | |
|---|---|
| Belle (1982): | 2.2 |
| Phoenix (1985): | 1.4 |
| Hitech (1987): | 1.5 |
| Zugzwang (1993): | 1.2 |

- Other game applications report similar results

## Perspective

- Minimal tree is roughly $b^{d/2}$
- Assume that you examine an average of 2 successors at a CUT node.
- For a depth 10 tree, average search order is roughly $2^5 b^{d/2}$ ; a factor of 32 within optimal!
- Improve branching factor at a CUT node to 1.6: a factor of 10.5 within optimal!
- Small improvements a CUT nodes translate to major performance improvements.

## References

[1] D. Slate and L. Atkin. “Chess 4.5 -- The Northwestern University Chess Program”, *Chess Skill in Man and Machine*, P. Frey (ed.), Springer-Verlag, 1977.

[2] J.Schaeffer. “The History Heuristic and the Performance of Alpha-Beta Enhancements”, *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 11, no. 11, pp. 1203-1212, 1989.