CS61B Lecture #32

Today: Backtracking searches, game trees (*DSIJ*, Section 6.5)

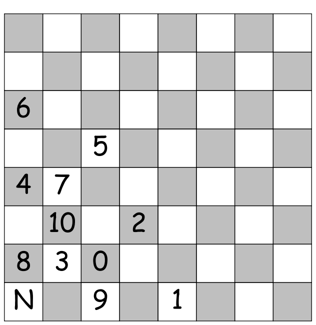
Coming Up: Concurrency and synchronization (*Data Structures*, Chapter 10, and *Assorted Materials On Java*, Chapter 6; Graph Structures: *DSIJ*, Chapter 12.

Searching by "Generate and Test"

- We've been considering the problem of searching a set of data stored in some kind of data structure: "Is $x \in S$?"
- But suppose we *don't* have a set S, but know how to recognize what we're after if we find it: "Is there an x such that $P(x)$?"
- If we know how to enumerate all possible candidates, can use approach of *Generate and Test:* test all possibilities in turn.
- Can sometimes be more clever: avoid trying things that won't work, for example.
- What happens if the set of possible candidates is infinite?

Backtracking Search

- Backtracking search is one way to enumerate all possibilities.
- Example: *Knight's Tour.* Find all paths a knight can travel on a chessboard such that it touches every square exactly once and ends up one knight move from where it started.
- In the example below, the numbers indicate position numbers (knight starts at 0).
- Here, knight (N) is stuck; how to handle this?



General Recursive Algorithm

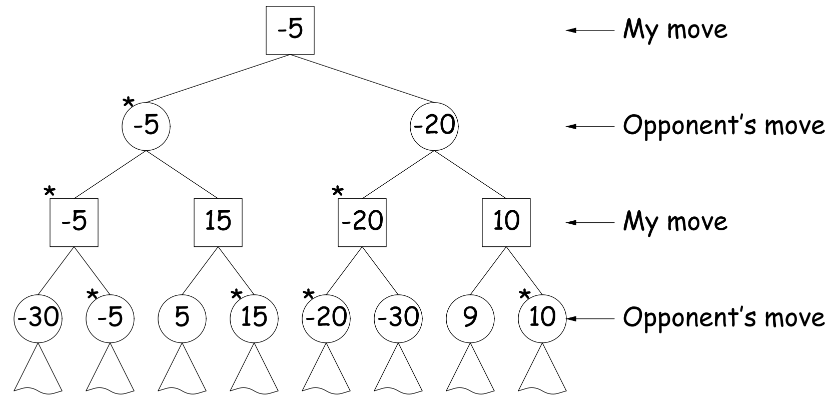
```
/** Append to PATH a sequence of knight moves starting at ROW, COL
 *  that avoids all squares that have been hit already and
 *  that ends up one square away from ENDROW, ENDCOL. B[i][j] is
 *  true iff row i and column j have been hit on PATH so far.
 *  Returns true if it succeeds, else false (with no change to PATH).
 *  Call initially with PATH containing the starting square, and
 *  the starting square (only) marked in B. */

boolean findPath (boolean[][] b, int row, int col,
                  int endRow, int endCol, List path) {
  if (path.size () == 64)   return isKnightMove (row, col, endRow, endCol);
  for (r, c = all possible moves from (row, col)) {
    if (! b[r][c]) {
      b[r][c] = true; // Mark the square
      path.add (new Move (r, c));
      if (findPath (b, r, c, endRow, endCol, path))  return true;
      b[r][c] = false; // Backtrack out of the move.
      path.remove (path.size ()-1);
    }
  }
  return false;
}
```

Another Kind of Search: Best Move

- Consider the problem of finding the *best* move in a two-person game.
- One way: assign a value to each possible move and pick highest.
  - Example: number of our pieces - number of opponent's pieces.
- But this is misleading. A move might give us more pieces, but set up a devastating response from the opponent.
- So, for each move, look at *opponent's* possible moves, assume he picks the best one for him, and use that as the value.
- But what if *you* have a great response to his response?
- How do we organize this sensibly?

Game Trees, Minimax

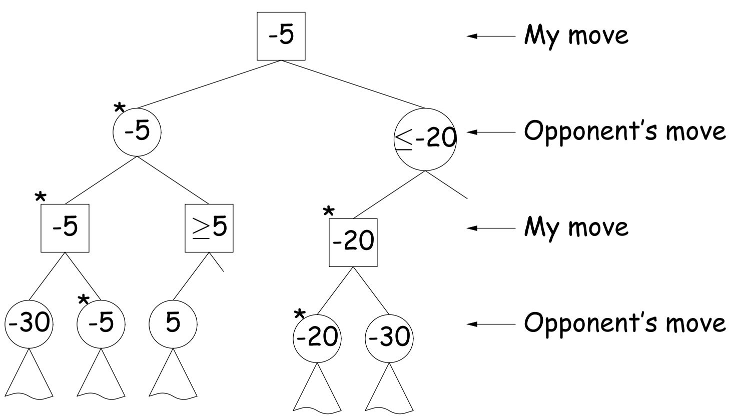
- Think of the space of possible continuations of the game as a tree.
- Each node is a position, each edge a move.



- Numbers are the values we guess for the positions (larger means better for me). Starred nodes would be chosen.
- I always choose child (next position) with maximum value; opponent chooses minimum value ("Minimax algorithm")

Alpha-Beta Pruning

- We can *prune* this tree as we search it.



- At the '≥ 5' position, I know that the opponent will not choose to move here (since he already has a -5 move).
- At the '≤ -20' position, my opponent knows that I will never choose to move here (since I already have a -5 move).

Cutting off the Search

- If you could traverse game tree to the bottom, you'd be able to force a win (if it's possible).
- Sometimes possible near the end of a game.
- Unfortunately, game trees tend to be either infinite or impossibly large.
- So, we choose a maximum *depth*, and use a heuristic value computed on the position alone (called a *static valuation*) as the value at that depth.
- Or we might use *iterative deepening* (kind of breadth-first search), and repeat the search at increasing depths until time is up.
- Much more sophisticated searches are possible, however (take CS188).

Some Pseudocode for Searching

```
/** A legal move for WHO that either has an estimated value >= CUTOFF
 *  or that has the best estimated value for player WHO, starting from
 *  position START, and looking up to DEPTH moves ahead.   */
Move findBestMove (Player who, Position start, int depth, double cutoff)
{
  if (start is a won position for who) return WON_GAME; /* Value ∞ */
  else if (start is a lost position for who) return LOST_GAME; /* Value −∞ */
  else if (depth == 0) return guessBestMove (who, start, cutoff);

  Move bestSoFar = REALLY_BAD_MOVE;
  for (each legal move, M, for who from position start) {
    Position next = start.makeMove (M);
    Move response =  findBestMove (who.opponent (), next,
                                   depth-1, -bestSoFar.value ());
    if (-response.value () > bestSoFar.value ()) {
      Set M's value to -response.value (); // Value for who = - Value for opponent
      bestSoFar = M;
      if (M.value () >= cutoff) break;
    }
  }
  return bestSoFar;
}
```


Static Evaluation

- This leaves static evaluation, which looks just at the next possible move:

```
Move guessBestMove (Player who, Position start, double cutoff)
{
  Move bestSoFar;
  bestSoFar = Move.REALLY_BAD_MOVE;
  for (each legal move, M, for who from position start) {
    Position next = start.makeMove (M);
    Set M's value to heuristic guess of value to who of next;
    if (M.value () > bestSoFar.value ()) {
      bestSoFar = M;
      if (M.value () >= cutoff)
        break;
    }
  }
  return bestSoFar;
}
```