Today: Backtracking searches, game trees (DSIJ, Section 6.5)
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 chess-board 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?

![Chessboard with numbers indicating positions](image-url)
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 **heuristic value** to each possible move and pick highest (aka **static evaluation**). Examples:
  - number of black pieces – number of white pieces in checkers.
  - weighted sum of white piece values – weighted sum of black pieces in chess (Queen=9, Rook=5, etc.)
  - Nearness of pieces to strategic areas (center of board).
- 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

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

-30
-5
5
15
-20
-30
9
10

My move
(maximizing)

Opponent’s move
(minimizing)

My move

Opponent’s move

• Suppose numbers at the bottom are the values of those final positions to me. Smaller numbers are of more value to my opponent.

• What should I move? What value can I get if my opponent plays as well as possible?
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 \textit{static valuation}) as the value at that depth.
- Or we might use \textit{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).
Overall Search Algorithm

- Depending on whose move it is (maximizing player or minimizing player), we’ll search for a move estimated to be optimal in one direction or the other.

- Search will be exhaustive down to a particular depth in the game tree; below that, we guess values.

- Also pass $\alpha$ and $\beta$ limits:
  - High player does not care about exploring a position further once he knows its value is larger than what the minimizing player knows he can get ($\beta$), because the minimizing player will never allow that position to come about.
  - Likewise, minimizing player won’t explore a position whose value is less than what the maximizing player knows he can get ($\alpha$).

- To start, a maximizing player will find a move with
  $$\text{findMax}(\text{current position}, \text{search depth } -\infty, +\infty)$$

- minimizing player:
  $$\text{findMin}(\text{current position}, \text{search depth } -\infty, +\infty)$$
The most basic kind of game-tree search is to assign some heuristic value to any given position, looking at just the next possible move:

```plaintext
Move simpleFindMax(Position posn, double alpha, double beta) {
    if (posn.maxPlayerWon())
        return artificial "Move" with value +∞;
    else if (posn.minPlayerWon())
        return artificial "Move" with value −∞;
    Move bestSoFar = artificial "Move" with value −∞;
    for (each M = a legal move for maximizing player from posn) {
        Position next = posn.makeMove(M);
        next.setValue(heuristicEstimate(next));
        if (next.value() >= bestSoFar.value()) {
            bestSoFar = next;
            alpha = max(alpha, next.value());
            if (beta <= alpha) break;
        }
    }
    return bestSoFar;
}
```
One-Level Search for Minimizing Player

```java
Move simpleFindMin(Position posn, double alpha, double beta) {
    if (posn.maxPlayerWon())
        return artificial "Move" with value +\infty; 
    else if (posn.minPlayerWon())
        return artificial "Move" with value -\infty;
    Move bestSoFar = artificial "Move" with value +\infty;
    for (each M = a legal move for minimizing player from posn) {
        Position next = posn.makeMove(M);
        next.setValue(heuristicEstimate(next));
        if (next.value() <= bestSoFar.value()) {
            bestSoFar = next;
            beta = min(beta, next.value());
            if (beta <= alpha) break;
        }
    }
    return bestSoFar;
}
```
Some Pseudocode for Searching (Maximizing Player)

/** Return a best move for maximizing player from POSN, searching
 * to depth DEPTH. Any move with value >= BETA is also
 * "good enough". */
Move findMax(Position posn, int depth, double alpha, double beta) {
    if (depth == 0 || gameOver(posn))
        return simpleFindMax(posn, alpha, beta);
    Move bestSoFar = artificial "Move" with value −∞;
    for (each M = a legal move for maximizing player from posn) {
        Position next = posn.makeMove(M);
        Move response = findMin(next, depth-1, alpha, beta);
        if (response.value() >= bestSoFar.value()) {
            bestSoFar = next;
            next.setValue(response.value());
            alpha = max(alpha, response.value());
            if (beta <= alpha) break;
        }
    }
    return bestSoFar;
}
Some Pseudocode for Searching (Minimizing Player)

/** Return a best move for minimizing player from POSN, searching
 * to depth DEPTH. Any move with value <= ALPHA is also
 * "good enough". */
Move findMin(Position posn, int depth, double alpha, double beta) {
    if (depth == 0 || gameOver(posn))
        return simpleFindMin(posn, alpha, beta);
    Move bestSoFar = artificial "Move" with value +\infty;
    for (each M = a legal move for minimizing player from posn) {
        Position next = posn.makeMove(M);
        Move response = findMax(next, depth-1, alpha, beta);
        if (response.value() <= bestSoFar.value()) {
            bestSoFar = next;
            next.setValue(response.value());
            beta = min(beta, response.value());
            if (beta <= alpha) break;
        }
    }
    return bestSoFar;
}