CS61B Lecture #35

[The Lecture #32 notes covered lectures #33 and #34.]

Today: Enumerated types, backtracking searches, game trees.

Coming Up: Graph Structures: DSIJ, Chapter 12

Enum Types in Java

- New version of Java allows syntax like that of C or C++, but with more guarantees:
  
  public enum Piece {
    BLACK_PIECE, BLACK_KING, WHITE_PIECE, WHITE_KING, EMPTY
  }

- Defines Piece as a new reference type, a special kind of class type.

- The names BLACK_PIECE, etc., are static, final enumeration constants (or enumerals) of type PIECE.

- They are automatically initialized, and are the only values of the enumeration type that exist (illegal to use new to create an enum value.)

- Can safely use ==, and also switch statements:

  boolean isKing (Piece p) {
    switch (p) {
      case BLACK_KING: case WHITE_KING: return true;
      default: return false;
    }
  }

Operations on Enum Types

- Order of declaration of enumeration constants significant: .ordinal() gives the position (numbering from 0) of an enumeration value. Thus, Piece.BLACK_KING.ordinal() is 1.

- The array Piece.values() gives all the possible values of the type. Thus, you can write:

  for (Piece p : Piece.values())
    System.out.printf("Piece value #%d is %s\n", p.ordinal(), p);

- The static function Piece.valueOf converts a String into a value of type Piece. So Piece.valueOf("EMPTY") == EMPTY.

Enumeration Types

- Problem: Need a type to represent something that has a few, named, discrete values.

- In the purest form, the only necessary operations are == and !=; the only property of a value of the type is that it differs from all others.

- In older versions of Java, used named integer constants:

  interface Pieces {
    int BLACK_PIECE = 0, // Fields in interfaces are static final.
    BLACK_KING = 1,
    WHITE_PIECE = 2,
    WHITE_KING = 3,
    EMPTY = 4;
  }

- C and C++ provide enumeration types as a shorthand, with syntax like this:

  enum Piece { BLACK_PIECE, BLACK_KING, WHITE_PIECE, WHITE_KING, EMPTY };

- But since all these values are basically ints, accidents can happen.

Last modified: Mon Nov 22 11:47:25 2004
**Fancy Enum Types**

- Enums are classes. You can define all the extra fields, methods, and constructors you want.
- Constructors are used only in creating enumeration constants. The constructor arguments follow the constant name:

```java
enum Piece {
    BLACK_PIECE (BLACK, false, "b"), BLACK_KING (BLACK, true, "B"),
    WHITE_PIECE (WHITE, false, "w"), WHITE_KING (WHITE, true, "W"),
    EMPTY (null, false, " ");
}
```

- `Piece` fields:
  - `color` (final `Side`)
  - `isKing` (final `boolean`)
  - `textName` (final `String`)

```java
Piece (Side color, boolean isKing, String textName) {
    this.color = color; this.isKing = isKing; this.textName = textName;
}
```

- `Piece` methods:
  - `color () { return color; }
  - `isKing () { return isKing; }
  - `textName () { return textName; }

**New Topic: 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?

```
<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
</tr>
<tr>
<td>4</td>
<td>7</td>
<td>10</td>
<td>2</td>
<td>5</td>
<td>6</td>
<td>9</td>
<td>1</td>
</tr>
</tbody>
</table>
```

**General Recursive Algorithm**

```java
/** 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 L).
 * 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 (L.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 CANT_MOVE;
    else if (start is a lost position for who) return CANT_MOVE;
    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;
}