Backtracking by image search

Search, Backtracking, Heuristics

- How do you find a needle in a haystack?
  - How does a computer play chess?
  - Why would you write that program?
- How does Bing/Googlemap find routes from one place to another?
  - Shortest path algorithms
  - Longest path algorithms
- Optimal algorithms and heuristic algorithms
  - When is close good enough? How do measure “closeness”?
  - When is optimality important, how much does it cost?

Searching with no guarantees

- Search for best move in automated game play
  - Can we explore every move?
  - Are there candidate moves ranked by “goodness”? Can we explore entire tree of possible moves?
- Search with partial information
  - Predictive texting with T9 or iTap or ...
  - Finding words on a Boggle board
  - What numbers fit in Sudoku square
- Try something, if at first you don’t succeed ...

Exhaustive Search/Heuristics

- We use binary search trees to organize data, in searching we don’t need to examine all the data to find what we’re looking for
  - Where is the smallest item in a search tree? Largest?
  - How are trees balanced?
- What do we do when the search space is huge?
  - How many chess boards are there?
  - Count routes between my house and yours?
- Exhaustive search: look at everything!
Classic problem: N queens

- Can queens be placed on a chess board so that no queens attack each other?
  - Easily place two queens
  - What about 8 queens?
- Make the board NxN, this is the N queens problem
  - Place one queen/column
  - Horiz/Vert/Diag attacks
- Backtracking
  - Tentative placement
  - Recurse, if ok done!
  - If fail, undo tentative, retry

Backtracking idea with N queens

- For each column C, tentatively place a queen
  - Try first row in column C, if ok, move onto next column
    - Typically “move on” is recursive
  - If solved, done, otherwise try next row in column C
    - Must unplace queen when failing/unwind recursion
- Each column C “knows” what row R it’s on
  - If first time, that’s row zero, but might be an attack
  - Unwind recursion/backtrack, try “next” location
- Backtracking: record an attempt go forward
  - Move must be “undoable” on backtracking/unwinding

N queens backtracking: Queens.java

```java
public boolean solve(int col){
    if (col == mySize) return true;
    // try each row until all are tried
    for(int r=0; r < mySize; r++){
        if (myBoard.safeToPlace(r,col)){
            myBoard.setQueen(r,col,
                      // try each column until all are tried
            if (solve(col+1)){
                return true;
            }
            myBoard.setQueen(r,col,
        }
    return false;
}
```

Basic ideas in backtracking search

- Enumerate all possible choices/moves
  - We try these choices in order, committing to a choice
  - If the choice doesn’t pan out we must undo the choice
    - Backtracking step, choices must be undoable
- Inherently recursive, when to stop searching?
  - When all columns tried in N queens
  - When we have found the exit in a maze
  - When every possible moved tried in Tic-tac-toe or chess?
    - Is there a difference between these games?
- Summary: enumerate choices, try a choice, undo a choice, this is brute force search: try everything
Pruning vs. Exhaustive Search

- If we consider every possible placement of 4 queens on a 4x4 board, how many are there? (N queens)
  - \(4 \times 4 \times 4 \times 4\) if we don’t pay attention to any attacks
  - \(4 \times 3 \times 2 \times 1\) if we avoid attacks in same row

- What about if we avoid diagonal attacks?
  - Pruning search space makes more search possible, still could be lots of searching to do!

- Estimate how long to calculate # solutions to the N-queens problem with our Java code....

Queens Details

- How do we know when it’s safe to place a queen?
  - No queen in same row, or diagonal
  - For each column, store the row that a queen is in
  - See QBoard.java for details

- For GUI version, we use a decorator
  - The QBBoardGUI is an IQueenState class and it has an IQueenState object in it
  - Appears as an IQueenState to client, but uses an existing one to help do its work
  - One of many object oriented design patterns, seen in Huff in the BitInputStream class

Daphne Koller

- 2004, Macarthur
- 2008, first ACM/Infosys

“The world is noisy and messy …You need to deal with the noise and uncertainty.”

“...I find it distressing that the view of the field is that you sit in your office by yourself surrounded by old pizza boxes and cans of Coke, hacking away at the bowels of the Windows operating system,” she said. “I spend most of my time thinking about things like how does a cell work or how do we understand images in the world around us?”

Computer v. Human in Games

- Computers can explore a large search space of moves quickly
  - How many moves possible in chess, for example?

- Computers cannot explore every move (why) so must use heuristics
  - Rules of thumb about position, strategy, board evaluation
  - Try a move, undo it and try another, track the best move

- What do humans do well in these games? What about computers?
  - What about at Duke?
Games at Duke

- Alan Biermann
  - Natural language processing
  - CompSci 1: Great Ideas
  - Duchess, checkers, chess

- Tom Truscott
  - Duke undergraduate working with/for Biermann
  - Usenet: online community
  - Second EFF Pioneer Award (with Vint Cerf!)

Heuristics

- A heuristic is a rule of thumb, doesn’t always work, isn’t guaranteed to work, but useful in many/most cases
  - Search problems that are “big” often can be approximated or solved with the right heuristics

- What heuristic is good for Sudoku?
  - Is there always a no-reasoning move, e.g., 5 goes here?
  - What about “if I put a 5 here, then…”

- What other optimizations/improvements can we make?
  - For chess, checkers: good heuristics, good data structures

Boggle Program

Boggle Search for Word

- Starting at board location (row,col): find a string S
  - We want to keep track of where we are in the string
  - Also track what board locations used for S search

- How do we know when we’re done?
  - Base case of recursive, backtracking call
  - Where we are in the string?

- How do we keep track of used locations?
  - Store in array list: tentatively use current one, recurse
  - If we don’t succeed, take off the last one stored!