Backtracking by image search

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
  - Autocorrect for SMS, mobile typing, …
  - Finding words on a Boggle board
  - What numbers fit in Sudoku square

- Try something, if at first you don’t succeed …
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?

Exhaustive Search/Heuristics

• We can probably explore entire game tree for tic-tac-toe, but not for chess
  • How many tic-tac-toe boards are there?
  • How many chess boards are there?

• What do we do when the search space is huge?
  • Brute-force/exhaustive won't work, need heuristics?
  • What about google-maps/Garmin finding routes?

• Backtracking can use both concepts
  • Game tree pruning a good idea most of the time
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
- wikipedia-n-queens

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,true);
            if (solve(col+1)){
                return true;
            }
            myBoard.setQueen(r,col,false);
        }
    }
    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 move 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
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

Recursive backtracking

- void solve(ProblemClass instance){
  - base cases
  - save current state
  - recursion
  - if current state DIDN’T work
    - remove current state
  - reassemble problem
  - }

Recursive backtacking

- Write `getPathSum`
  - Takes int as `target` sum
  - Returns Stack of Intgers, the path to target sum
    - `getPathSum(19)` returns 7, 12
    - `getPathSum(8)` return an empty stack

```
Recursive backtacking

- Write `getPathSum`
  - Takes int as `target` sum
  - Returns Stack of Intgers, the path to target sum
    - `getPathSum(19)` returns 7, 12
    - `getPathSum(8)` return an empty stack
```

![Recursive backtacking diagram](image)

```
Recursive backtacking

- Write `getPathSum`
  - Takes int as `target` sum
  - Returns Stack of Intgers, the path to target sum
    - `getPathSum(19)` returns 7, 12
    - `getPathSum(8)` return an empty stack
```

![Recursive backtacking diagram](image)