CS472 Module 8 Part A - Backtracking

Athens State University

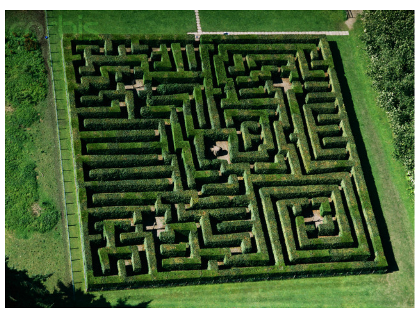
March 13, 2016

Outline

Contents

1 Introduction

Get me out of here!



How do you (efficiently) find a way out of a garden maze?

Backtracking

- Let's start with a brute force approach
- But stop and place a marker when we find that we heading towards a dead end
- In general, we a problem we can express the set of solutions found via brute force in terms of a state space
  - And our attempts to solve the problem as being a depth-first traversal of that state space
    - And we prune back (mark) the paths that don't lead us to a solution

The General Method

- The backtracking algorithm enumerates a set of *partial candidates* that could be *completed* in different ways to give all the possible solutions to a problem
- This is done incrementally by a sequence of *candidate extension* steps
- We represent partial candidates as a state space tree that we will call the *potential search tree*
  - This tree is ordered so that the each partial candidate is the parent of candidates that differ from it by a single extension step
- The algorithm will do a depth-first traversal of the search tree that checks at each node if we can reach a solution from that node
- If not, then it "prunes" the tree by skipping the whole sub-tree rooted at that node

The General Method

- We must provide a data set P and six helper functions:
  1. `root(P)`: return the partial candidate at the root of the search tree
  2. `reject(P,c)`: a Boolean function that returns `true` if the partial candidate `c` is not worth completing
  3. `accept(P,c)`: a Boolean function that returns `true` if `c` is a solution of P and `false` otherwise
  4. `first(P,c)`: generate the first extension of candidate `c`
  5. `next(P,s)`: generate the next alternative extension of a candidate, after the extension `s`
  6. `output(P,c)`: use the solution `c` of `P`, as appropriate to the problem

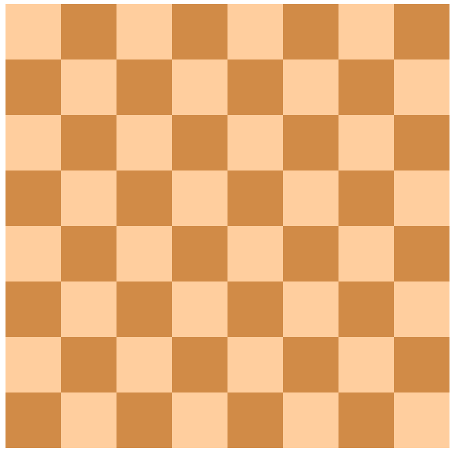
Algorithm 1: backtrack(): A general backtracking algorithm

Input: A set P of data for an instance of a problem
Output: A solution c for the problem instance P

```
if reject(P,c) then
    return
if accept(P,c) then
    output(P,c);
s ← first(P,c);
while s is not empty do
    backtrack(s);
    s ← next(P,s);
```

2 Example: The n-Queens Problem

Example: The n-Queens Problem



- We ask the question "How can one place n queen pieces on a n-by-n chessboard so that no two queens threaten each other for any natural number n?"
  - The 8-by-8 queens problem fits the standard chess board

Consider the 8-queens problem

- There are ${}_{64}C_8 = 4,426,165,368$ possible arrangements of eight queens on a standard board but only 92 distinct solutions
- We can narrow the problem by constraining each queen to a single column or row so that we only have $8^8 = 16,777,216$ possible combinations
- By generating permutations, we can narrow the state space down to $8! = 40,320$ possibilities that can be checked for diagonal attacks

A backtracking solution

Algorithm 2: solveNQueens: Backtracking algorithm for the n-queens problem

```
column ← 1;
if all queens are placed then
    return true;
for row ∈ rows in the current column do
    if can place queen in this row then
        Place queen at board[row][column];
        if solveNQueens(column + 1) returns true then
            return true;
        Backtrack by removing queen from board[row][column];
Indicate no queen can fit in this column by returning false;
```

3 Example: Sudoku puzzles

Example: Sudoku puzzles

- Sudoku puzzles are a logic-based combinatorial number-placement puzzle

- The objective is fill a 9x9 grid with digits so that each column, each row, and each of the nine 3x3 sub-grids that compose the puzzle contains all of the digits from 1 to 9
- Completed puzzles must meet additional constraints on contents of individual regions
  - For example, the same integer cannot appear twice in the same row, column, or in any of nin3 3x3 squares in the grid
- The author of the puzzle provides a partially completed grid, which assuming a well-posed puzzle, has a unique solution

Example: Sudoku puzzles

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

5	3	4	6	7	8	9	1	2
6	7	2	1	9	5	3	4	8
1	9	8	3	4	2	5	6	7
8	5	9	7	6	1	4	2	3
4	2	6	8	5	3	7	9	1
7	1	3	9	2	4	8	5	6
9	6	1	5	3	7	2	8	4
2	8	7	4	1	9	6	3	5
3	4	5	2	8	6	1	7	9

A few additional facts about the math behind Sudoku:

1. A completed Sudoku grid is a special type of what in combinatorics is known as a *Latin square*: a n by n array filled with n different symbols occurring exactly once in each row and column.
2. The difference between a Sudoku grid and a Latin square is that one must also maintain the invariant of no repeated values in any of the 9 blocks of contiguous 3 by 3 squares.
3. It is has been proved that the number of classic Sudoku grids is approximately $6.67 * 10^{21}$. This is number is reduced to approximately $5.47 * 10^9$ if you consider only those solutions that are essentially different solutions
4. The number of minimal classic Sudoku puzzles is not precisely known. A minimal puzzle is one in which no clue can be deleted without losing uniqueness of the solution.
5. The maximum number of givens provided while still not rendering a unique solution is four short of a full grid. The inverse problem of the fewest number of givens that render a unique solution was only proved in 2012 as being 17 givens. Over 49,000 examples of such puzzles have been identified.
6. The general problem of solving Sudoku puzzles on n^2 by n^2 boards of n by n blocks is known to be NP-complete.

A Backtracking Algorithm for 9x9 Sudoku puzzles

Algorithm 3: Sudoku: A backtracking algorithm for 9x9 puzzles

```
Input: A 9x9 array grid with known values and a position
Output: The array grid with solution
if endOfGrid?() then
    return true;
for x ∈ [1..9] do
    grid[position] ← x;
    if gridisValid?() then
        if Sodoku(nextPosition()) then
            return (true);
gridPosition[position] = NULL;
return false;
```

A Theoretical Aside

- Sudoku is an example of a mathematical problem known as the *exact cover* problem
  - Given a collection S of subsets of a set X, an **exact cover** is a subcollection S^* of S s.t. each element of X is contained in *exactly one* subset in S^*
- In computer science, the **exact cover problem** is a decision problem to determine if an exact cover exists.
  - This problem is one of the classic *NP-complete* problems
  - Other examples of the exact cover problem include the tromino problem and (in a slightly more generalized form) the N-queens problem
  - There are more advanced algorithms (Knuth's DLX algorithm, for example) that can be used to get even better performance for these problems

4 Key Points

Key Points

- What is backtracking?
- What is the n-queens problem? How do we use backtracking to solve the problem?
- Use of backtracking for solving Suduku puzzles