Exhaustive Search, Backtracking, and Dynamic Programming

Miles and Mohan

August 17, 2015

Exhaustive search In the exhaustive search, we iterate through all possible solutions in the solution space. Conceptually, we are iterating through the leaves of a tree representing the solution space..

Backtracking In backtracking, we more closely model our search for a solution as navigating through the conceptual solution space tree.

However, rather than visiting every leaf (which represents a possible solution), we remove branches that corespond to a set of solutions, none of which can be the optimal solution. Sometimes, the branch removal is because the entire branch is infeasible. In others, it's because the solution is no better than a solution we've already found. By always keeping the best-solution-so-far, we can rule out branches which can't beat that current best.

Dynamic Programming In Dynamic Programming, like in divide-and-conquer, we create a recursive solution for our problem by creating multiple subproblems. However, unlike in divide-and-conquer, these subproblems may overlap.

The problem with these multiple overlapping subproblems, is that, as they generate their own subproblems, we may have duplicate subproblems being calculated. The solution, which is simple, is to save the results of solving each subproblem in a table. If that subproblem comes up again, the table is consulted, and the already-computed result is returned.

There are two ways to fill in the table:

Top-Down In this case, we augment the recursive calculation with, first, a check for whether or not the soluation has already been computed in the table. If so, that saved value is returned. If not, the solutions is computed, and then stored in the table.

Bottom-Up In this case, we fill in the table iteratively, starting at the simplest cases and working our way up.

One phrase that can be helpful is *use it or lose it*. The idea is that you consider an item, and figure out: what would happen if you chose to use this item, and what would happen if you chose not to use the item. Then, pick the better of the two. Sometimes, dynamic programming problems have more than one choice (use or not use), but the concept is still the same, consider your various choices by recursively computing the result of each choice.

1 Problems

1. Describe an algorithm for solving Sudoku mini problems. Here's an example Sudoku mini problem:

3			2		1
	5	1		6	3
	1		5	3	2
5	3	2		4	
4	2		6	1	
1		5			4

Sudoku mini rules: complete the grid so that each row, column, and 3×2 box contains every digit from 1 to 6 inclusive.

Solution

Here's an exhaustive search approach using a recursive solution:

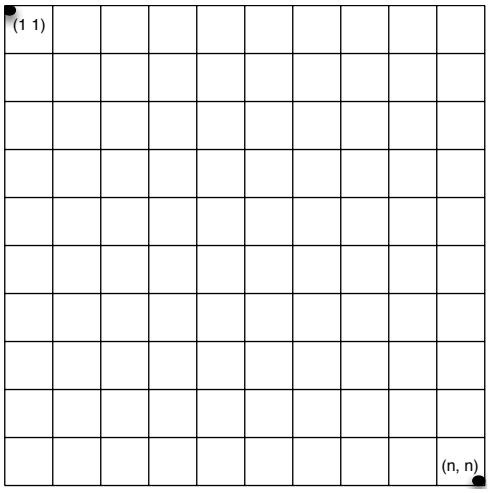
```
def solveSudoko(board);
  if board is filled in and each row, column, and 3 x 2 box
  contains every digit from 1 to 6 inclusive:
    return True
  find the row and column of the first empty square and save as r, c
```

```
  for value in range(1,7):
    board[r][c] = value
    if solveSudoku(board):
      return  True  // We solved the problem!
  return False
```

This problem works well using a backtracking approach. We'll modify the recursive solution to prune out some of the work.

```
def solveSudoko(board);
  if board is filled in and each row, column, and 3 x 2 box
  contains every digit from 1 to 6 inclusive:
    return True
  if some row, column, or 3 x 2 box has a duplicate of a digit from 1 to 6
    return False  // We're pruning the branch
  find the row and column of the first empty square and save as r, c
  for value in range(1,7):
    board[r][c] = value
    if solveSudoku(board):
      return True  // We solved the probem!
  return False
```

2. Imagine an $n \times n$ grid (in each direction, n intersections connected by $n-1$ streets). Determine how many ways there are to travel from the top-left intersection to the bottom-right intersection, assuming that travel is always down one street, or to the right one street.
Here's an example 11×11 grid.



This problem is easily solved using dynamic programming. Here's a recursive solution (that doesn't yet use dynamic programming):

```
# Determines the number of paths from vertex (toCol, toRow)
# to vertex (numCols, numRows)
def numberPaths(toCol, toRow):
  if toRow == numRows or numRows == numCols:
    return 1
  return numberPaths(fromCol, fromRow + 1)  #move down
       + numberPaths(fromRow, fromCol + 1) # move right
```

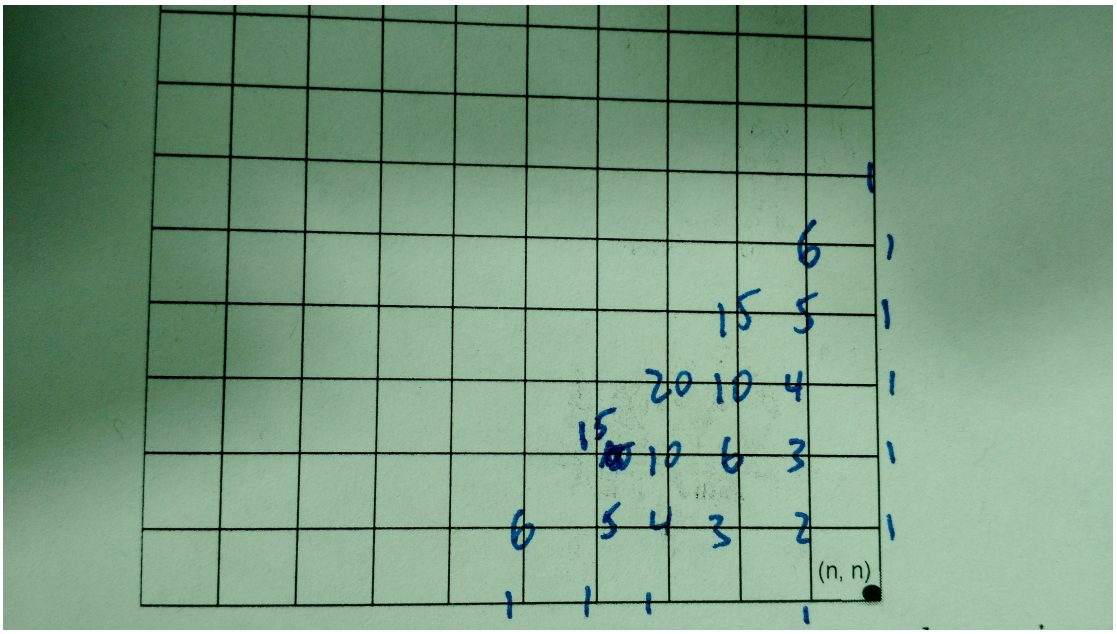
In order to turn this into a dynamic programming solution (to avoid massive recomputation), we add a two-dimensional table (initialized to -1) that is of size $n \times n$.

```
# Determines the number of paths from vertex (toCol, toRow)
# to vertex (numCols, numRows)
def numberPaths(toCol, toRow):
  if table[toCol][toRow] == -1:
    if toRow == numRows or numRows == numCols:
```

```
      table[toCol][toRow] = 1
    else:
      table[toCol][toRow] = numberPaths(fromCol, fromRow + 1)  #move down
         + numberPaths(fromRow, fromCol + 1) # move right
  return table[toCol][toRow]
```

If we wanted to fill in the table bottom-up, we'd start by filling in ones for the bottom row and right column. Then, we'd start at the bottom right and make cells be the sum of the cell below and the cell to the right.

The bottom-right of the table would look like:

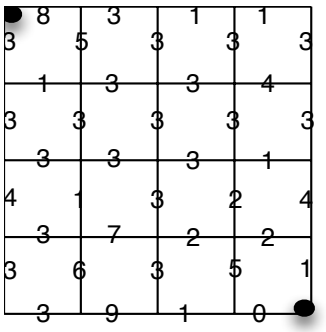


2 Homework

1. Imagine an $n \times n$ grid (in each direction, n intersections connected by $n-1$ streets), but where there's a charge to drive on each street (different streets may have different charges). Determine the minimal cost to travel from the top-left intersection to the bottom-right intersection, assuming that travel is always down one street, or to the right one street.

   (a) What is the minimal cost to travel on this grid?

   (b) How would you describe how perform an algorithm on an $n \times n$ grid with an arbitrary charge on each street?



2. Given a 4×4 chessboard, figure out how to arrange four queens such that none of them can attack any other (no two in a vertical, horizontal, or diagonal line). Describe an algorithm to solve this problem for an 8×8 chessboard. Can you actually find a solution for an 8×8 board?

3 Advanced Problems

1. Consider the problem of determining how *close* two strings are to one another. This is useful, for example, in suggesting spelling corrections. Consider that a string can be edited by:

   - adding a character,
   - removing a character, or
   - changing one character to another.

   For example, you can change "some" to "sammy" by changing the 'o' to an 'a', by adding a second 'm' after the first one, and then by changing the 'e' to a 'y'. Thus, the minimal *edit distance* is 3. (Longer edit distances also exist: you could delete the 'o' and the'e', and then add 'a', 'm', and 'y' for a total edit distance of 5.)

   Create an algorithm to determine the minimal edit distance between any two strings.

2. Imagine an $n \times n$ grid (in each direction, n intersections connected by $n-1$ streets), but where there's a square of $\frac{n}{3} \times \frac{n}{3}$ vertices in the middle that are inaccessible. Determine the number of paths from the top-left intersection to the bottom-right intersection, assuming that travel is always down one street, or to the right one street.