Recursion and DFS

Kevin Geng

21 October 2016

1 Exploring search spaces

The basic idea behind recursion is that a function can call itself. This has a lot of applications, but in particular, we're interested in using recursion to explore states. If from any given state in a problem, we can reach other states, then we can simply recur to explore all possible states. This is known as a depth-first search. (Why?)

This idea can be represented by a search tree. The root node of the tree is the state from which we start. Then, its children are the other states that can be generated from it, and so on. We'll go over a few examples to demonstrate how we can use this idea to solve problems.

1.1 Permutations

Problem: Given an integer n, print out all possible permutations of the integers $1..n$.[1] For example, if $n = 3$, the permutations would be (1, 2, 3), (1, 3, 2), (2, 1, 3), (2, 3, 1), (3, 1, 2), (3, 2, 1).

This is easy to solve for a fixed value of n, such as $n = 3$, just by writing n for-loops. But we can't exactly output n for-loops. Instead, we can build each permutation of length 1. We can then recur on those results to build permutations of length 2, and so on.

Algorithm 1 Generating permutations

```
used ← {false, false, ..., false}
function GeneratePermutations(depth, prefix)
    if depth = n then
        print(prefix)
        return
    for i = 1..n do
        if not used[i] then
            used[i] ← true
            GeneratePermutations(depth + 1, prefix ⊕ i)
            used[i] ← false
```

1.2 n-Queens

Problem: Place n queens on an $n \times n$ chessboard so that no two are attacking each other. In other words, no two queens may be in the same row, column, or diagonal.

We can approach this problem in a similar fashion. First, we observe that we need to make is that every column must contain a queen. Then we can start by trying to place a queen in the first column. For each of those, we can try to place a queen in the second column, eliminating states that result in two queens attacking each other. We continue until we've filled the board.

[1] In Python, you can use `itertools.product(range(n))`!

1.3 More problems

1. Farmer John has received an order for exactly M units of milk ($1 \leq M \leq 200$) that he needs to fill, but all he has are two milk pails of integer sizes X and Y ($1 \leq X, Y \leq 100$) with which he can measure milk. Using these two pails, he can perform up to K of the following types of operations ($1 \leq K \leq 100$):

   - fill either pail completely to the top
   - empty either pail
   - pour the contents of one pail into the other, until the former is empty or the latter is full

   FJ may not be able to end up with exactly M total units of milk in the two pails, so compute the minimum value of $|M - M'|$ such that FJ can construct M' units of milk collectively between the two pails. (USACO February 2016 Silver: Milk Pails)

2. Two teams are competing in a game of basketball: the Exonians and the Smurfs. There are n players on the Exonian team and m players on the Smurf team, with $n + m \leq 17$. Each player has an integer skill level s between 1 and 108.

   Define the strength of a set of players as the sum of their individual skill levels. In order to ensure a fair game, the Exonians and Smurfs plan on choosing two equally strong starting lineups. In how many ways can the two teams choose their lineups? (Two lineups are considered different if there exists a player who starts in one game, but not in the other.) (Crash Course Coding Companion §2.2.2)

2 Graph traversal

If you're familiar with graph theory, you may recognize that we can use recursive DFS to traverse all nodes in a graph. We can treat each node as a possible state, and recur on each of its neighbors. One change that we must make, however, is to check whether we've already visited a node before recurring on it. If we don't do this and the graph contains a cycle, then we will recurse indefinitely.

We can apply this to a graph represented by an adjacency list, which stores the neighbors of each node. We can also apply this to a grid, in which the neighbors of a cell are the four adjacent cells.

Interestingly, the edges that were traversed in the graph correspond to the search tree of the recursion. Assuming the graph is connected, the search tree is a *spanning tree* of the graph, which means that it includes all of the vertices in the graph.

2.1 Problems

1. Bessie and her sister Elsie want to travel from the barn to their favorite field. The farm is a collection of N fields ($1 \leq N \leq 16$) numbered $1..N$, where field 1 contains the barn and field N is the favorite field. An assortment of M paths connect pairs of fields. However, each path can only be followed in the direction $X \to Y$ if $X < Y$. Each pair of fields is connected by at most one path, so $M \leq N(N-1)/2$.

   Bessie and Elsie leave at exactly the same time from the barn, and also arrive at exactly the same time at their favorite field. However, it might take Bessie and Elsie different amounts of time to follow a path. Please help determine the shortest amount of time Bessie and Elsie must take in order to reach their favorite field at exactly the same moment. (January 2015 Bronze: Meeting Time)

2. December 2015, Silver: Switching on the Lights

3. January 2016, Silver: Build Gates

3 Recursion stack

When we perform recursion, we are implicitly using a stack: the call stack. This stores the arguments to functions and their local variables. We can transform any function that uses recursion to a function that does not, simply by keeping track of the relevant state ourselves rather than letting the programming language do it for us. One reason we might want to do this is that programming languages limit the maximum recursion depth. So one advantage of explicitly using a stack is that you can "recurse" more deeply. However, expressing programs in this way may be less clear than plain recursion.

When performing graph traversal, we can use an explicit stack to keep track of nodes to visit rather than using recursion. One of the downsides of this, however, is that performing an inorder or postorder traversal of a graph is not as simple. This is something we will need in our algorithm for topological sort below.

4 Topological sort

4.1 Introduction

A formal definition of a topological sort is an ordering of the vertices of a directed graph such that for every edge $u \to v$, u comes before v in the ordering. The canonical example is for scheduling courses that have prerequisites, which are represented by edges. In this case, the topological sort represents a valid order in which to take courses.

4.2 Tarjan's algorithm

With Tarjan's algorithm[2], we can easily solve this problem in $O(n)$ with just a few lines of code. Though it's not immediately obvious why, the reverse post-order of a DFS traversal is a topological ordering.

Algorithm 2 Tarjan's algorithm for topological sort

```
function DFS(v)
    add v to visited
    for w in v.neighbors do
        if w not in visited then
            DFS(w)
    push v onto postorder
function TopologicalSort(graph)
    initialize empty stack postorder
    initialize empty set visited
    for v in graph do
        if v not in visited then
            DFS(v)
    return postorder
```

Tarjan's algorithm, however, can't easily be changed to use a stack or queue instead of recursion. This is because it requires a post-order traversal, which is only easily accomplished through recursion. Kahn's algorithm solves the problem iteratively using either a stack or a queue, but it is more complicated.

[2]Not to be confused with the algorithm for strongly connected components, which *Tarjan's algorithm* usually refers to.