Backtracking and Dynamic Programming

1 Notes on Backtracking

- *When does it apply:* It applies when we need to find a solution (of a specific format) to a problem, under certain constraints. Often, the time needed for an exhaustive search may be prohibitive, so we may approach the problem with a backtracking algorithm.
- *What is it:* Backtracking is an important improvement over exhaustive search (a.k.a. brute force approach). It constructs the solution one component at a time, and it evaluates the partial solution (by checking if it can be developed further without violating the problem constraints). If the partial solution can not be developed further, it does not consider and evaluate all the possible developments. Instead, it backtracks directly to the last legitimate partial solution, and it considers the next legitimate choice of the next component.
- *Why do we use it* (is it anyway better?): Backtracking is a way to evaluate in a fast way many candidate solutions, until we find a solution that satisfies all the constraints. It can solve many problems which can not be solved with exhaustive search (because of the size of the problem). In other words, for specific problems, it can save a lot of time.
- *How does it work (main idea):*
  - if we want to apply backtracking, we should be able to write the solution as a series of (successive) choices $(1, 2, \ldots, N)$;
  - each choice is done over a finite number of outcomes;
  - if a partially constructed solution at step n satisfies all the constraints, we make a new choice at step $n + 1$;
  - if a partially constructed solution at step n does not satisfy all the constraints, we consider another choice at step n; if there is no other legitimate choice at step n, we backtrack to step $n - 1$;
  - the algorithms stops when we find a solution at the last step, N, which satisfies all the constraints (or after we are able to show that a valid solution does not exist).

2 Notes on Dynamic Programming

- *When does it apply:* Dynamic programming can be applied when we are able to identify a *recursive procedure* to solve the problem. In a recursive procedure, a problem can be decomposed into a set of smaller problems of the same type. The solution is obtained by combining the solutions of all the smaller problems. In particular, it is useful when such procedure is making repeated calls to the same sub-problems.

- *What is it:* Instead of solving the same sub-problems multiple times (each time they are called), with dynamic programming we can provide some structure to the problem. Each sub-problem is solved only once, and after it has been solved once its result is recorded on a table, from which it is possible to re-use such solution every time the sub-problem is called again.
- *Why do we use it* (is it anyway better?): In problems that can be solved with a recursive procedure (as in the examples), the recursive procedure often recalls the same sub-problem several time, leading to an unnecessary exponential blow-up in the total computation time. In dynamic programming, these repetitions are identified, and the answer to each subproblem are stored and reused, thus avoiding the unnecessary repetitions. Dynamic programming is a technique to avoid re-calculating the same sub-problems, thus significantly speeding up the search for the optimal solution to the problem.
- *How does it work (main idea):*
  - we identify a recursive solution to the problem; usually, such solution is a simple backtracking algorithm; we look for the solution in a *top-down* approach:
    - we identify the sub-problems, and we find a smart way to classify and label the repeated sub-problems;
    - the way in which we label the subproblems will also suggest us the dimension of the problem, and how to store all the results;
  - once the sub-problems have been identified, labelled and classified, we proceed in a *bottom-up* approach to find the solution:
    - we start by solving the *base-cases*, i.e., the simplest sub-problems that can not be decomposed into simpler sub-problems;
    - the solutions of the base-cases provide us with the bricks to build the solution to another set of sub-problems; we calculate the solution to all the sub-problem whose solution can be calculated by combining the available solutions; usually, if the problem is well-structured, the identification of this set of sub-problems is an easy step;
    - we proceed iteratively, considering the next set of sub-problems whose solution can be obtained by composing the available solutions;
    - the algorithm terminates when we solve the initial problem.
- *Correctness:* the correctness of the dynamic programming approach derives from the correctness of the recursion, which is usually easy to prove. Indeed, we do the same operation as in the standard recursion, just in a different order and avoiding to repeat the same calculation.

3 Problem

Problem 1: N Queens

Place N queens on an N x N chessboard so that no two queens attack each other by being in the same column, row, or diagonal.

Solution: N Queens

Problem Overview

Restate the problem in a way that is meaningful to you. This will be different for everyone, but generally it may involve creating some small examples, using different words, stating it from a different perspective, coming up with some notation to make it easy to talk or write about. For the n-queen problem, we can restate it as follows. First, we notice that the problem has a trivial solution in the case in which $N = 1$, while it is easy to see that it does not have any solution for $N = 2$ and $N = 3$. Let's consider the case $N = 4$. We have 4 successive choices to make, i.e., we need to place the 4 queens in the 4x4 chessboard.

Brainstorming

- *Problem Complexity:* First of all, let's consider how many choices we have to place the queens in the 4x4 chessboard. In general, we can place the first queen in 16 different places, the second one in the 15 remaining and so on. This makes a total of 16x15x14x13 = 43680 possibilities. Indeed, we are not interested in the order of the queens placed, but only in the final placement. The total number of possible placement of the 4 queens is $16!/(4!(16-4)!) = 1820$. If we use an exhaustive search (brute force approach) we need to verify all of them.

- *Note:* In general, the number of ways to choose k different objects (the order of which is not important) from a set of n different objects, is called combinations of n objects taken k at a time. It is denoted by $C(n,k) = n!/(k!(n-k)!)$.

- *Observation:* Let's try to simplify the problem. We should position one queen per column, from 1 to 4, in order to satisfy the constraints (no more than a queen per column !). The first choice corresponds to the row in which we want to place the queen in the first column, the second choice corresponds to the row in which we place a queen in the second column, and so on. This observation simplifies significantly the problem. It reduces the number of choices to 4x4x4x4= 256, which is better, even if it is still too high.

  After this observation, with an exhaustive search (brute force approach) we need to verify all 256 possibilities. But with backtracking we can do better.

Solution

- *Case $N = 4$:* Let's start to consider the consecutive choices. We start by placing the first queen in the first available position (column=1, row=1). We go to the second choice. The second queen can not be placed in the (c=2, r=1), and neither in (c=2, r=2). The first valid position is (c=2, r=3). We go to the third choice. We discover that there is no way to place the third queen, then we backtrack. We try to place the second queen in the last available position, (c=2, r=4). The only available position to place the third queen is (c=3, r=2). But there are no valid position to place the fourth queen. We backtrack. No other valid position to place the third queen. We backtrack again. No other valid position to place the second queen. We backtrack again. We place the first queen in the next available position, (c=1, r=2). The only valid position to place the second queen is (c=2, r=4). We go to the next step. The only valid position to place the third queen is (c=3, r=1), and finally, there is a position also to place the fourth queen in the fourth column, i.e., (c=4, r=3)!!!

- *Analysis of the solution:* We have found a valid solution! (c=1, r=2), (c=2, r=4), (c=3, r=1), and (c=4, r=3). Let's analyze what we have found, and the method we have used, by answering to some questions.

1. Is the solution found with a backtracking method always valid? do we need to prove that it is a valid solution? By correctly applying backtracking, at each step we verify that the new choice satisfies all the constraints. Thus, if the final choice satisfies all the constraint, we have identified a set of choices that solve the problem. In the specific case of the n-queen problem, by placing the first queen, we are sure we satisfy the constraints (always true). By placing the second queen, we make sure that it does not violate the constraints. Thus, the solution up to that point (with two queens) does not violate the constraints. Before placing the n-th queen, we have verified that the solution up to the (n-1)th queen does not violates the constraints. It is then sufficient to verify that the n-th queen does not violates the constraints to have a solution with n queens.
2. Is the solution found unique? With backtracking, we are looking for one solution to the problem proposed. There may be many different solutions. To find all the solutions with backtracking, we need to evaluate all the choices that satisfy the constraints of the problem. Usually, one solution to the problem is enough.
3. Do we always find a solution if it exists? Is there any efficiency bound if we use backtracking? By running backtracking over all the possible choices that satisfies the constraints, we always find a solution if it exists. For some problems, this approach may be much faster thank an exhaustive search, but for some other problems, in the worst case the complexity may be the same as an exhaustive search.

- General Solution: In the general case we have $N > 3$ queens to place in a NxN chessboard. We place one queen per column. We have N choices to make, i.e., we need to choose the row in which to place each queen. We enumerate the columns from 1 to N. We start from the first column, and we place a queen in the first available position. Then we follow the general backtracking procedure:
  - we consider the next column, that we call in general n, and we place a queen in the first position that satisfies the constraints;
  - if this valid position exists, we proceed to the following column $(n + 1)$, repeating the previous point;
  - if instead we can not find any solution at step n that satisfies all the constraints, we backtrack to step $n - 1$, and we consider the next valid choice at step $n - 1$;
  - the algorithms stops when we find a solution at the last step, N, which satisfies all the constraints.

Variants and Generalizations

The number of solutions for the 4-queens problem is 2 (the second one is easy to find). We might be interested to know that the total number of different solutions to the 8-queens problem is 92, twelve of which are qualitatively distinct, with the remaining 80 obtainable from the basic twelve by rotations and reflections. As to the general N-queens problem, it has a solution for every $N > 3$, but no convenient formula for the number of solutions for an arbitrary N has been discovered. It is known that this number grows very fast with the value of N. For example, the number of solutions for $N = 10$ is 724, of which 92 are distinct, while for $N = 12$ the respective numbers are 14200 and 1787.

In general, many puzzles can be solved by backtracking, and the search for a solution is usually more efficient than with exhaustive search. Anyway, a more efficient search, with respect to backtracking, exists for many of these problems.

Problem 2: Maximum Sum Descent

Some positive integers are arranged in a triangle like the one shown in Fig. 1. Design an algorithm (more efficient than an exhaustive search, of course) to find the largest sum in a descent from its apex to the base through a sequence of adjacent numbers, one number per each level. Start by solving the problem in Fig. 1, then generalize the solution to a N levels problem.

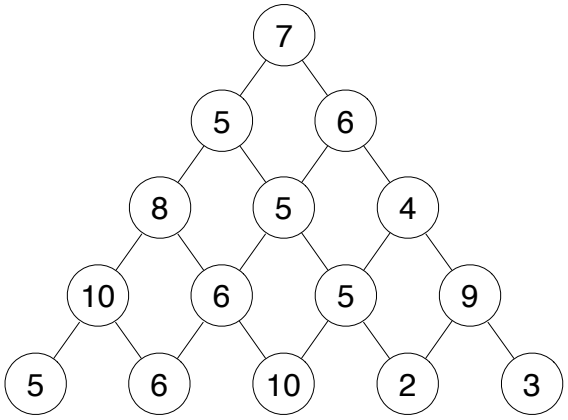

Figure 1: Example of numbers in the Maximum Sum Descent problem.

Problem 3: Maximum Sum Descent

Problem Overview

Restate the problem in a way that is meaningful to you. For the maximum sum descent problem, we can restate it as follows. We consider each number as a node, and we observe that if we have a triangle of numbers of height N, we can divide the nodes in layers and enumerate the layers from 0 (the top layer) to $N-1$ (the lowest layer).

We need to find the path which connect the top node to one of the bottom nodes such that the sum of all the numbers on that path is the maximum (considering all the possible paths). Starting from the top node, a valid path is obtained by choosing one adjacent node on the lower layer (the first on the left or the first on the right), for all the layers until we reach the lowest layer.

Brainstorming

- *Problem Complexity:* First of all, let's consider how many possible paths are available from layer 0 to layer $N-1$. If we have only one layer, we have of course only one trivial path. If we consider a triangle of height two, we have two possible path. If we consider a triangle with levels form 0 to 2, we have, for each path from level 0 to level 1, exactly two possibility to choose a node at level 2, thus the number of paths double again. In general, the number of paths doubles at each new layer we add at the bottom, since for each path at layer $n-1$, we have two possible choices of a node at layer n. Thus, if we have N layers, from 0 to $N-1$, we have a total number of path equal to 1(at level 0) x 2 (at level 1) x 2 (at level 2) x ... x 2 (at level $N-1$) $= 2^{N-1}$. For each path, we need to sum over N numbers (the length of the

path), thus we need to perform $N \cdot 2^N$ simple sums. For $N = 5$ the number of operation is 160, for $N = 10$ is 10240, for $N = 20$ is 20971520.

- *Greedy approach:* we can start from the top of the triangle, and choose the node at layer 1 with the highest number, then we can choose among the two nodes at layer 2 the one with the highest number and so on. We find a solution in linear time (N comparisons and N summations), but is this the optimal solution? We have no guarantee, and it is actually very easy to find cases in which, with this method, we fail to find the optimal solution (see e.g., the specific case in Fig. 1).

- *Observation 1:* We do not need to calculate the sums over all possible paths (brute force approach). Indeed, we can decompose the problem into sub-problems. If we have calculated the path with the highest sum from the bottom to the first node at level 1, and the highest sum from the bottom to the second node at level 1, we pick up the highest of the two, we sum the node at level 0, and we have found the solution.

- *Observation 2:* This is true in general, at each level n of the triangle of numbers. If we have already computed the path with the highest sum for all the nodes from the bottom until level $n+1$, we can calculate the path with the highest sum from the bottom to each node at layer n, by considering for each node the highest sum between the two choices of the lower layer node.

Solution

- *Case $N = 3$:* Let's consider this simple case. We are looking for a solution of the problem for a triangle of numbers of height 3. This is given by the choice (based on the highest sum) between the two triangles of height two, considering layer 1 and layer 2, and by summing the number at layer 0. With a bottom up approach, we know that the path with the highest sum from the bottom to each of the 2^2 nodes at layer 2 is just given by the number in that node (layer 2 is the bottom layer). We need to make a choice and a sum to define the path with the highest sum for each of the 2^1 nodes at layer 1, and again a choice and a sum for each of the $2^0 = 1$ nodes at layer 0, and we have defined the solution

- *Analysis of the solution:*

  1. Is the solution found with dynamic programming the optimal solution? Yes, and this can be proven by induction. Let's give a sketch of the proof. The solutions are trivial, and optimal, for each subproblem at the bottom layer, $N - 1$. The solutions for each subproblems at level $n < N - 1$, obtained with the procedure illustrated, are also optimal, given that the solutions to all the subproblems at layer $n + 1$ are optimal.
  2. Is the solution found unique? In this kind of problems, there may exist more than one optimal solution. With the dynamic programming technique discussed, we find only one of the optimal solutions. It is possible, with minor modifications to the technique (you are invited to think about them) to find all the optimal solutions that exist.
  3. Is there any efficiency bound if we use dynamic programming? This depends on the structure of the problem. For the maximum sum descent, if we have a triangle of numbers of height N, we need to compute N-1+N-2+...+1= N(N-1)/2 operations (including a comparison and a sum). Note that in the case of $N = 20$, we have seen that with a brute force approach we need 20971520 operations, while with dynamic programming we need 380 operations!

- General Solution: We just apply the same technique as in the case of $N = 3$. Let's write it down step by step.
  - we consider the optimal solutions for each node at layer $N - 1$, the bottom layer (*base-cases*);
  - for each node at layer $N - 2$, we choose the highest sum between the two possible choices at layer $N - 1$, and we sum it to the value of the node at layer $N - 2$; to keep track of the optimal path, we need to keep track also of the chosen node at layer $N - 1$;
  - we repeat the same procedure for all nodes at layer $N - 3$, then $N - 4$, up to layer 0;
  - the solution at layer 0 is the solution to the problem.

Reflections

Dynamic programming is a very general and powerful technique which can be applied to a vast range of problems, in the fields of mathematics, computer science, economics, bioinformatics, telecommunications, and so on.

4 Homework

Problem 4: Magic Square Revised

A magic square of order 3 is a 3×3 table filled with nine distinct integers from 1 to 9 so that the sum of the numbers in each row, column, and two corner-to-corner diagonals is the same. Find all the magic squares of order 3.

Problem 5: Shortest Path Counting

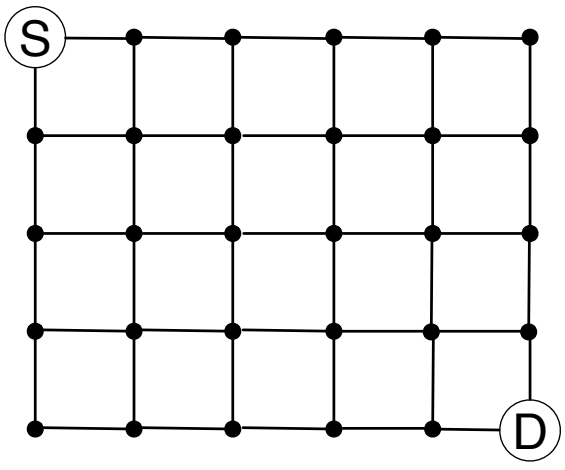

Figure 2: The city streets of the Shortest Path Counting problem.

Find the number of the shortest paths from the starting point S to the destination D in a city with perfectly horizontal streets and vertical avenues shown in Figure 2.

5 Advance Problems

Problem 6: The Icosian Game

This is a 19th-century puzzle invented by the renowned Irish mathematician Sir William Hamilton (1805–1865) and presented to the world as the "Icosian Game." The game was played on a wooden board with holes representing major world cities and grooves representing connections between them (see Fig. 3).

The object of the game is to find a circular route that would pass through all the cities exactly once before returning to the starting point. Can you find such a route?

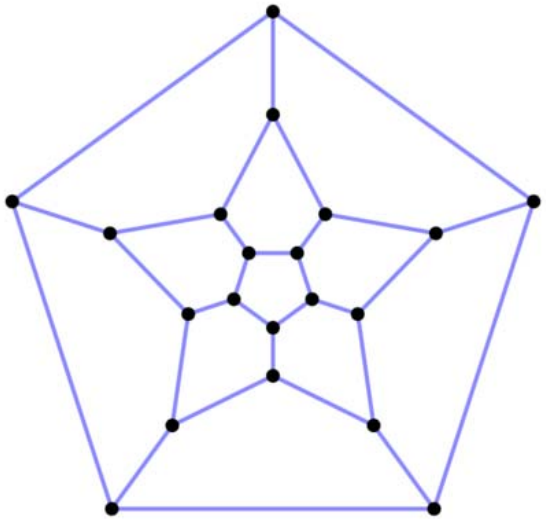

Figure 3: Cities and connections for the Icosian game.

Problem 7: Jumping Into Pairs I

There are n coins placed in a row. The goal is to form $n/2$ pairs of them by a sequence of moves. On each move a single coin can jump right or left over two coins adjacent to it (i.e., over either two single coins or one previously formed pair), to land on the next single coin; no triples are allowed. Any empty space between adjacent coins is ignored. Determine all the values of n for which the puzzle has a solution and devise an algorithm that solves the puzzle in the minimum number of moves for such ns.

Problem 8: Palindrome Counting

In how many different ways can the palindrome "WAS IT A CAT I SAW" be read in the diamond-shaped arrangement shown in Tab 1? You may start at any W and go in any direction on each step up, down, left, or right through adjacent letters. The same letter can be used more than once in the same sequence.

					W							
				W	A	W						
			W	A	S	A	W					
		W	A	S	I	S	A	W				
	W	A	S	I	T	I	S	A	W			
W	A	S	I	T	A	T	I	S	A	W		
	W	A	S	I	T	A	T	I	S	A	W	
		W	A	S	I	T	I	S	A	W		
			W	A	S	I	S	A	W			
				W	A	S	A	W				
					W	A	W					
						W						

Table 1: Letter Arrangement

Problem 9: Blocked Paths

Find the number of different shortest paths from the starting point S to the destination D in a city with perfectly horizontal streets and vertical avenues as shown in Figure 4. No path can cross the fenced off area shown in grey in the figure.

6 Challenge problems

Problem 10: Picking Up Coins

Some coins are spread in the cells of an n x m board, one coin per cell. A robot, located in the upper left cell of the board, needs to collect as many of the coins as possible and bring them to the bottom right cell. On each step, the robot can move either one cell to the right or one cell down from its current location. When the robot visits a cell with a coin, it picks up that coin. Devise an algorithm to find the maximum number of coins the robot can collect and a path it needs to follow to do this.

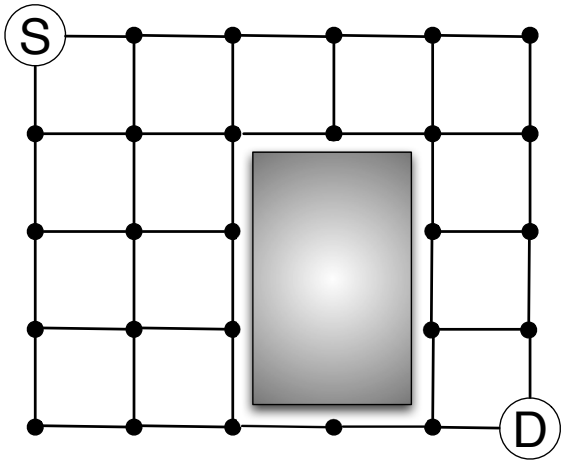

Figure 4: The city streets of the Blocked Paths problem.

Problem 11: Pile Splitting

(a) Given n counters in a pile, split the counters into two smaller piles and compute the product of the numbers of the counters in the two piles obtained. Continue to split each pile into two smaller piles and to compute the products until there are n piles of size one. Once there are n piles, sum all the products computed. How should one split the piles to maximize the sum of the products? What is this maximal sum equal to?

(b) How does the solution to the puzzle change if we are to compute the sum of the numbers of the counters in the two piles obtained after every split and have a goal of maximizing the total of such sums?