Ask a Question

Introduction to Backtracking

Let’s go over the Backtracking pattern, its real-world applications, and some problems we can solve with it.

We'll cover the following...

About the pattern

Imagine we’re planning an exciting road trip through a city, aiming to visit all the places we want to see while covering the shortest distance possible. However, there are some conditions we must follow: we can’t revisit the same place more than once, and we must end up back where we started. This problem, known as the city road trip problem, requires finding the optimal route that satisfies these conditions. It’s a classic example where the concept of backtracking comes into play, allowing us to explore different paths until we find the shortest one that fulfills all the conditions.

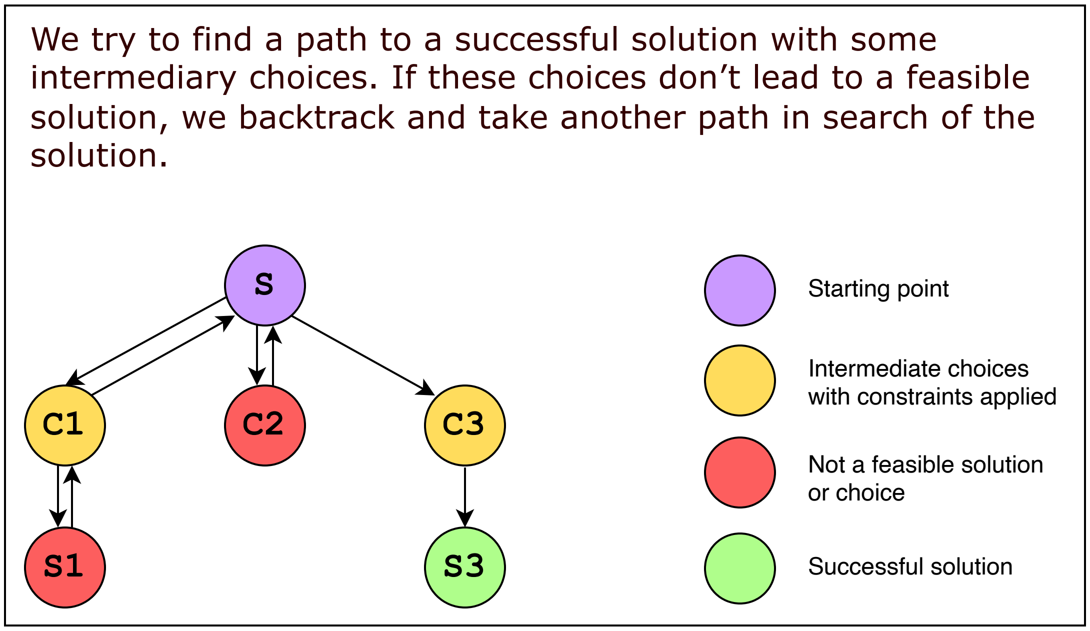
Let’s first see how this problem can be solved using a brute-force approach. We can do this by exploring routes in every single way we can visit the places. We have to write down every possible route, check how long each one is, and then pick the shortest one. But as our list of places grows, it makes this approach computationally impractical for a large number of routes.

Now, let’s look at a backtracking approach to solve the same problem. With backtracking, we can start by picking a place and choose the next place to visit that’s close and follows our conditions. We move back (backtrack) to the previous place if the current place has been visited before or if we cannot move forward to any place from here. We check these conditions on each of our choices because we do not want to break any of our road trip rules. We keep doing this, choosing, checking conditions, and backtracking until we've visited all the places according to the requirements. At every step, we choose the closest place, ensuring we have chosen the shortest path to visit all the places we want to see.

Backtracking is an algorithmic technique for solving problems by incrementally constructing choices to the solutions. We abandon choices as soon as it is determined that the choice cannot lead to a feasible solution. On the other side, brute-force approaches attempt to evaluate all possible solutions to select the required one. Backtracking avoids the computational cost of generating and testing all possible solutions. This makes backtracking a more efficient approach. Backtracking also offers efficiency improvements over brute-force methods by applying constraints at each step to prune non-viable paths.

As seen in the above example, backtracking works by exploring all potential routes toward a solution step-by-step. It can be visualized as traversing a state space tree, where each node represents a partial solution. Starting from the root (an empty solution), backtracking moves deeper into the tree, exploring branches (choices) until it finds a feasible solution or reaches a leaf node that cannot be extended into a complete solution. Upon reaching a dead end, the algorithm backtracks to the previous state and explores a different branch. This process is repeated, with constraints applied at each step to avoid exploring paths that cannot lead to a successful, feasible solution.

Let's look at the visualization of the space state tree below to better understand the working:


We try to find a path to a successful solution with some intermediary choices. If these choices don’t lead to a feasible solution, we backtrack and take another path in search of the solution.

S
C1
C2
C3
S1
S3
Starting point
Intermediate choices with constraints applied
Not a feasible solution or choice
Successful solution

In the visualization above, we start with an initial point, S. From this point, we proceed to explore a potential solution, $\mathrm{S1}$, via an intermediate choice, $\mathrm{C1}$. After evaluating, we determine that $\mathrm{S1}$ does not satisfactorily solve our problem. Therefore, we backtrack to S and then shift our exploration towards another potential choice, $\mathrm{C2}$. This process of exploration and backtracking continues until we identify a successful feasible solution.

In the scenario above, both $\mathrm{S1}$ and $\mathrm{C2}$ fail to provide feasible solutions and only $\mathrm{S3}$ emerges as a successful solution to the problem. This illustrates that the backtracking approach examines all potential combinations until it discovers a successful feasible solution.

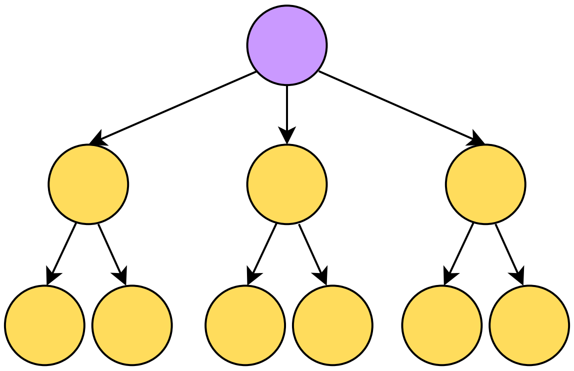
The backtracking algorithm can be implemented using recursion. We use recursive calls where each call attempts to move closer towards a feasib solution. This can be outlined as follows after starting from the initial poi as the current point:

- **Step 1:** If the current point represents a feasible solution, declare success and terminate the search.

- **Step 2:** If all paths from the current point have been explored (i.e., the current point is a dead-end) without finding a feasible solution, backtrack to the previous point.
- **Step 3:** If the current point is not a dead-end, keep progressing towards the solution, and reiterate all the steps until a solution is found or all possibilities are exhausted.

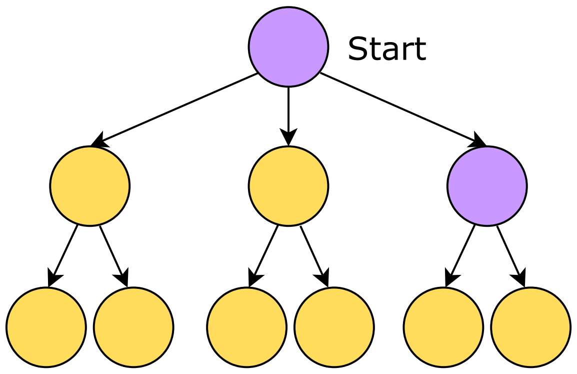
If all the points have been explore without finding a feasible solution, declare failure; no solution exists.

We start with an initial point of value(s). We check if our required conditions are met.



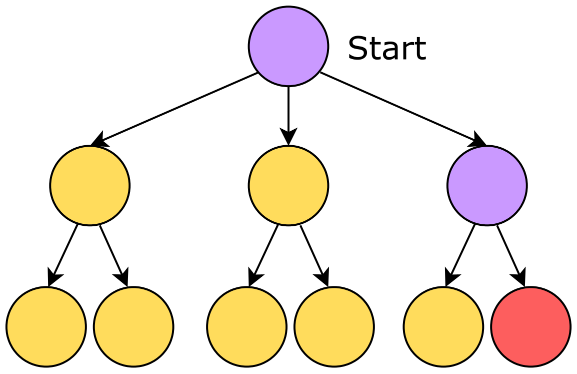
1 of 7

The required conditions are met. Therefore, we move to the next node of a path.



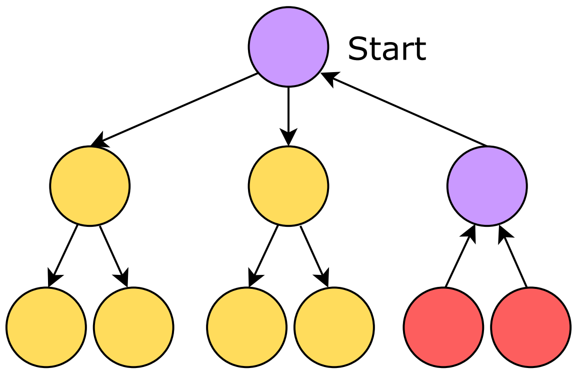
2 of 7

The required conditions are not met, therefore, we backtrack and explore another path.



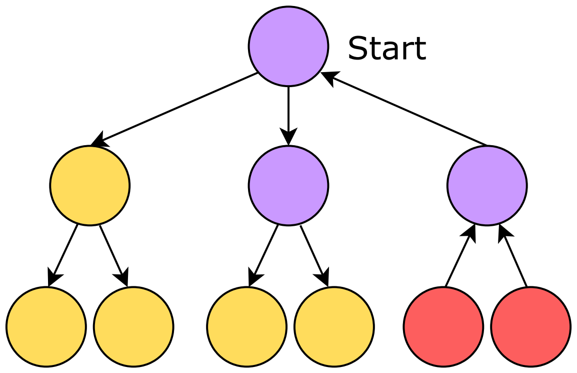
3 of 7

The required conditions are not met, therefore, we backtrack and explore another path.



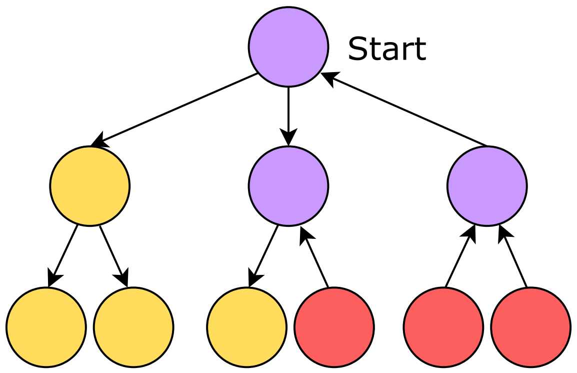
4 of 7

The required conditions are met. Therefore, we move to the next node of a path.



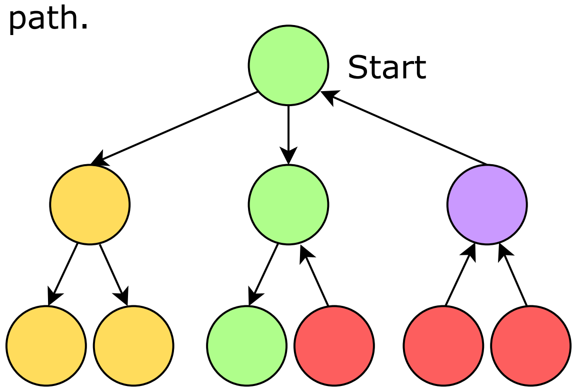
5 of 7

The required conditions are not met, therefore, we backtrack and explore another path.



6 of 7

The required conditions are met, and this is the feasible solution. Therefore, we have reached our desired solution from the green path.



7 of 7

Examples

The following examples illustrate some problems that can be solved with this approach:

1. Path in binary matrix: Find a path of 1s from top-left to bottom-right in an $n \times n$ binary maze. We are only allowed to move to the right or downward.

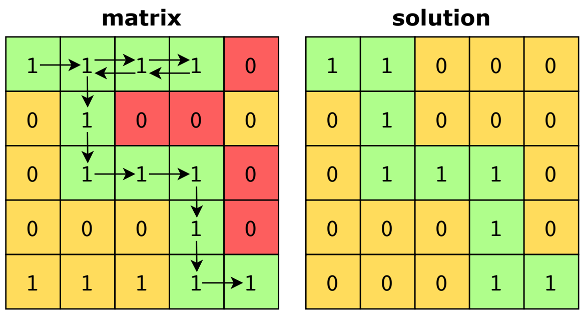
We declare a **solution** matrix initialized to **0** to store the path we find.

1. We start from the top-left cell of the matrix and check if the position is valid, i.e., whether the cell contains **1**.

2. If the position is valid, we recursively keep moving to the right and mark the corresponding cell in the solution matrix to **1**.

3. If we encounter a **0**, we can't move to the right anymore, so we try to move down from there.

4. If we can't go in either direction, we backtrack to a **1** from which we can still move downward and again try to recursively find a valid path from there. When backtracking, we need to unmark the corresponding position in the **solution** matrix to **0**, since that cell will not be part of the path leading to the destination.



matrix

1	1	1	1	0
0	1	0	0	0
0	1	1	1	0
0	0	0	1	0
1	1	1	1	1

solution

1	1	0	0	0
0	1	0	0	0
0	1	1	1	0
0	0	0	1	0
0	0	0	1	1

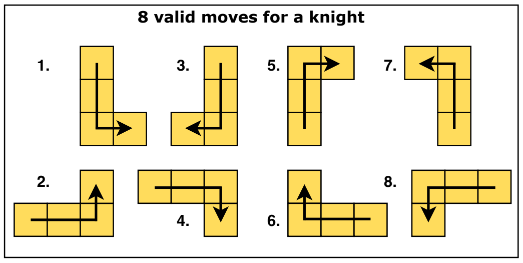
2. Check knight tour configuration: Check if a knight can cover all possible squares once in an $n \times n$ chessboard. The initial position of the knight is at the top-left square of the board.

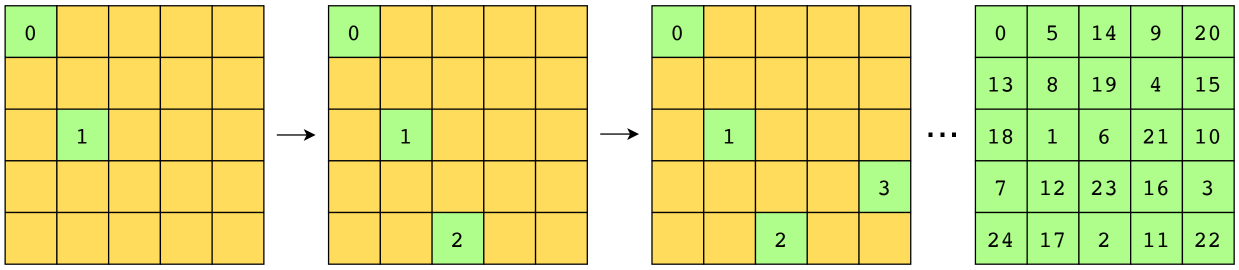
1. We recursively check if the knight can move from its current square to another unvisited square using one of its eight valid moves given below.

2. If a valid move is made, we mark the square as visited.

3. If we cannot move to an unvisited square with any of the eight valid moves, we mark the current square as unvisited and backtrack to the last visited square. We then attempt to make another valid move from this square.

4. If we have achieved our goal of visiting every square, meaning no unvisited squares are left; we've found a valid configuration of the knight's tour, and we return **TRUE.** Otherwise, we return FALSE.





In the 5x5 board above, **0** is the initial position of the knight, **1** is the second position after the first move, **3** is the third position after the second move, and so on. It took **24** moves in total to visit all squares.

Does your problem match this pattern?

- Yes, if any of these conditions is fulfilled:
  - **Complete exploration is needed for any feasible solution:** The problem requires considering every possible choice to find any feasible solution.
  - **Selecting the best feasible solution:** When the goal is not just to find any feasible solution but to find the best one among all feasible solutions.
- No, if the following condition is fulfilled:
  - **Solution invalidity disqualifies other choices:** In problems where failing to meet specific conditions instantly rules out all other options, backtracking might not add value.

Real-world problems

Many problems in the real world share the backtracking pattern. Let's look at some examples.

- **Syntax analysis:** In compilers, we use recursive descent parsing. It is a form of backtracking, to analyze the syntax of the program. This analysis involves matching the sequence of tokens (basic symbols of programming language) against the grammar rules of the language. When a mismatch occurs during the analysis, the parser backtracks to a previous point to try a different rule of the grammar. This ensures that even complex nested structures can be accurately understood and compiled.
- **Game AI (Artificial Intelligence):** In games like chess or Go, AI algorithms use backtracking to try out different moves and see what happens. If a move doesn't work out well, the AI goes back and tries something else. This helps the AI learn strategies that might be better than those used by humans because it can think about lots of different moves and figure out which ones are likely to work best.
- **Pathfinding algorithms:** In pathfinding problems like finding the way through a maze or routing in a network, backtracking is used. It tries out different paths to reach the destination. If it hits a dead end or a spot it can't pass through, it goes back and tries another path. This keeps happening until it finds a path that works and leads to the destination.

Strategy time!

Match the problems that can be solved using the backtracking pattern.

> **Note:** Select a problem in the left-hand column by clicking it, and