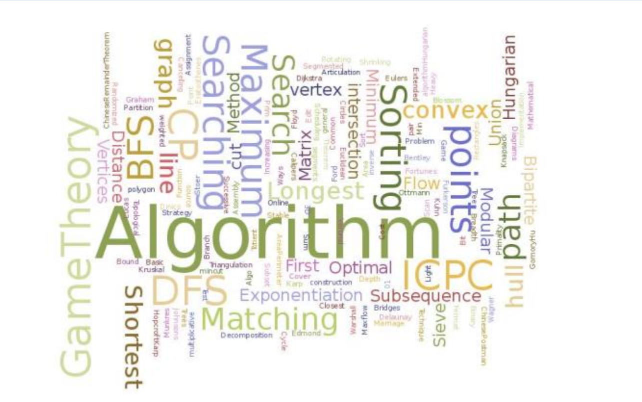
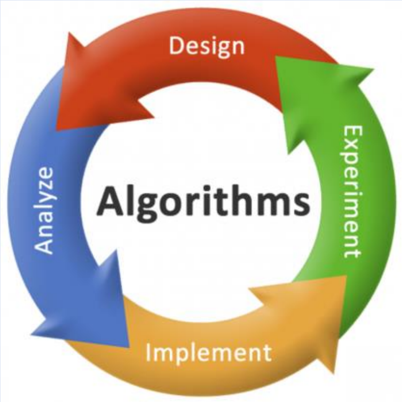
Backtracking

COP 3503
Fall 2021

Department of Computer Science
University of Central Florida
Dr. Steinberg

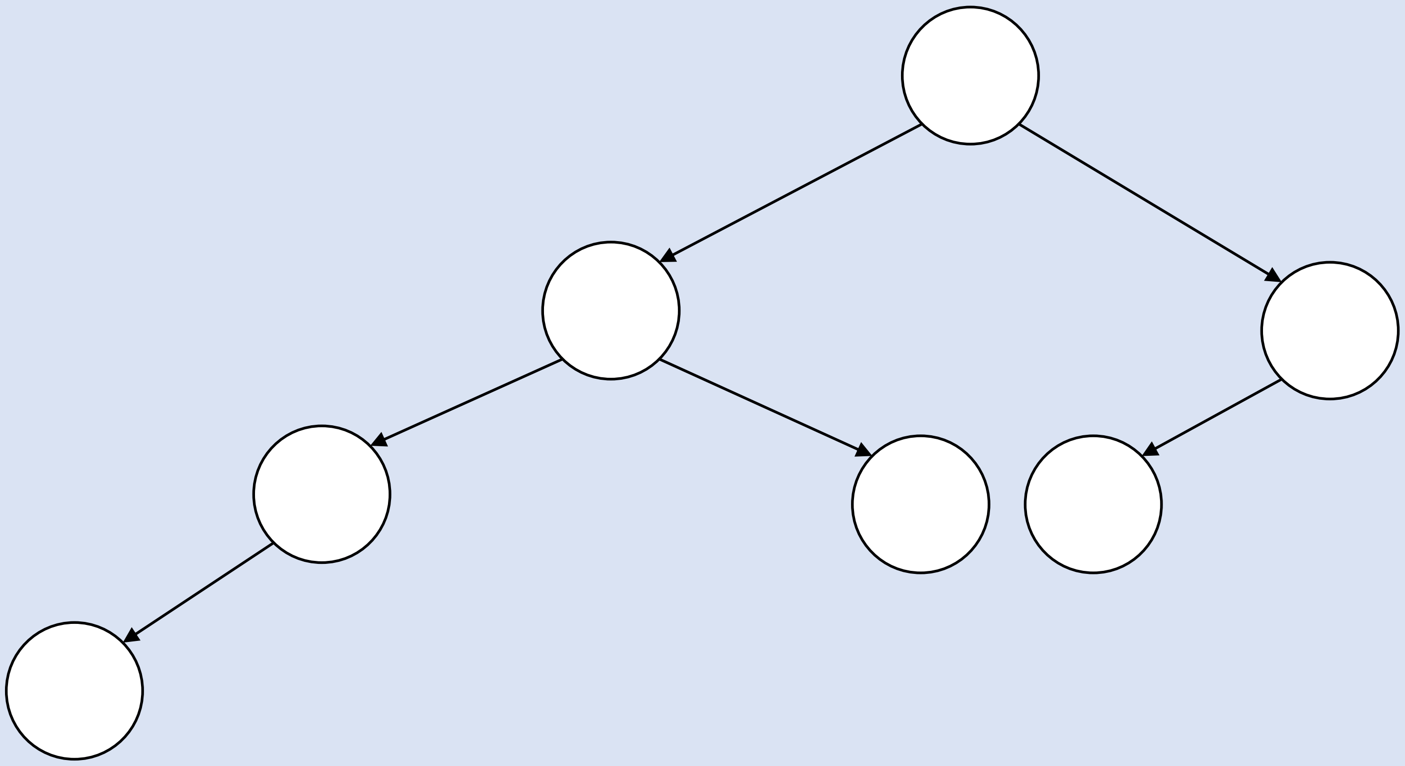




Introduction

- We just observed two common approaches to designing algorithms
  - Brute Force (BF)
  - Divide and Conquer (DC)
- BF and DC both yield correct results to problems. Both have their pros and cons that were discussed in previous lectures.
- There can be more than one correct result, however one may be better than the other.
- In this lecture we will discuss a technique called Backtracking

Quick Review of Tree Terminology



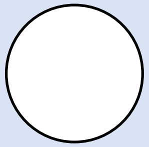
Backtracking

- Uses a Search Tree
- The Root is the starting state before the search for solutions
- Nodes on the first level, choices made for the first component of the solution
- Nodes on the second level, choices for second component of the solution
- The pattern continues…
- Each node on the tree is considered a potential solution, if it corresponds to a partially constructed solution that may still lead to a full solution.
- Leaves in the tree are dead ends or complete solution.

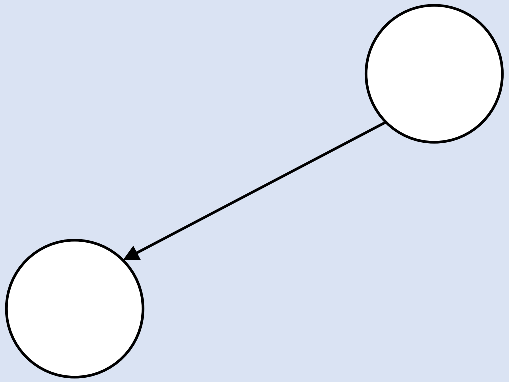
The Search Tree Construction

- If the current node is considered a potential solution, its child is generated by adding the first remaining legitimate option for the next component of a solution
  - The algorithm moves onto the child
- If the current node is NOT considered a potential solution, then the algorithm “backtracks” to the parent to consider the next potential solution.
  - If no option is possible, then the algorithm backtracks up on more level in the search tree.
- If the potential solution turns out to be the complete solution, then the algorithm stops (assuming we are searching for one solution).

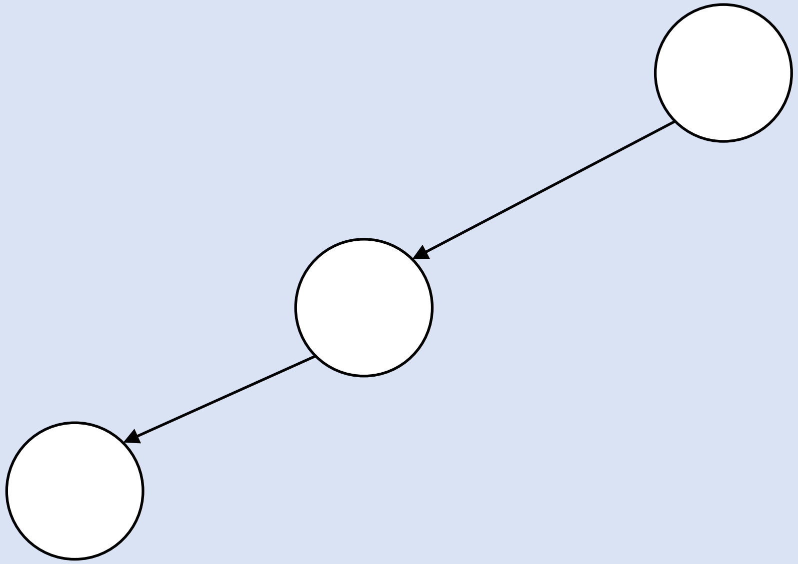
The Big Picture of Backtracking



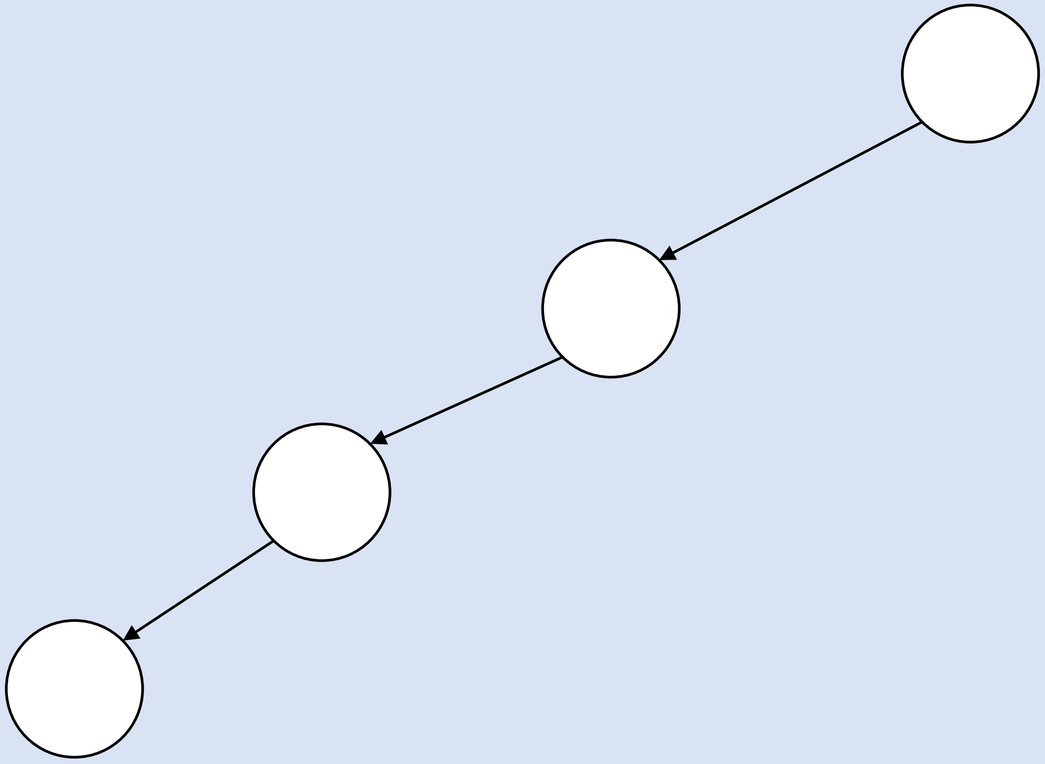
The Big Picture of Backtracking



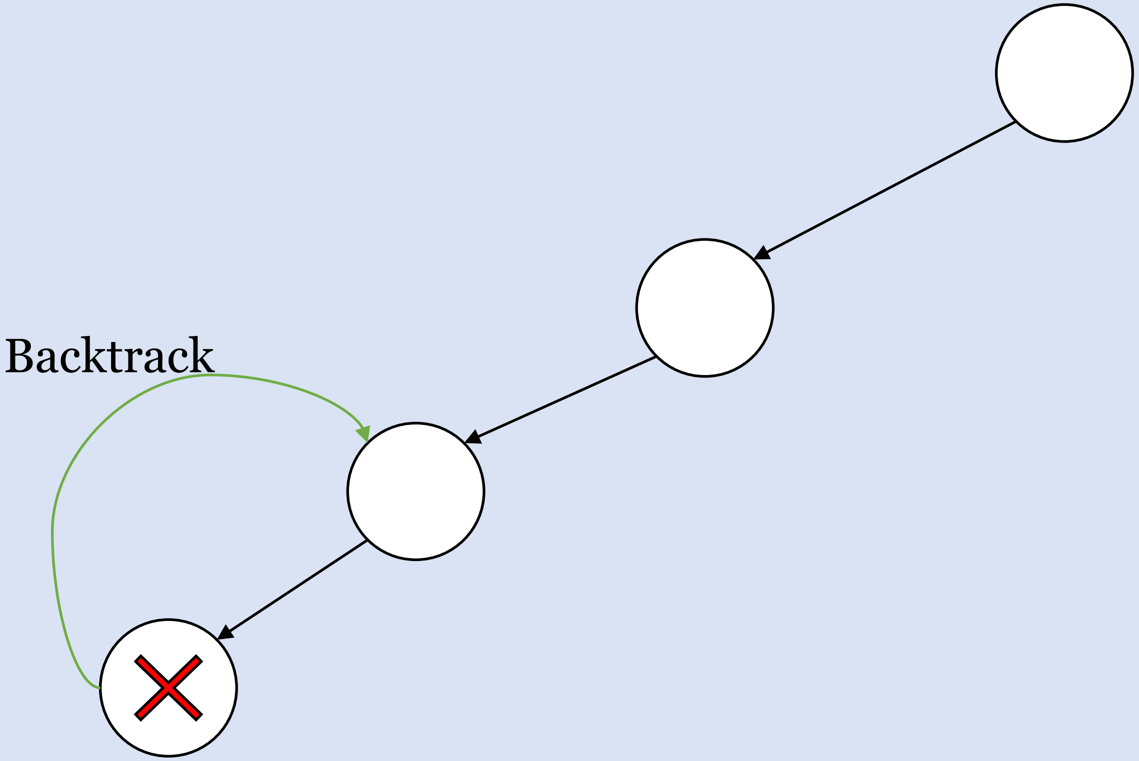
The Big Picture of Backtracking



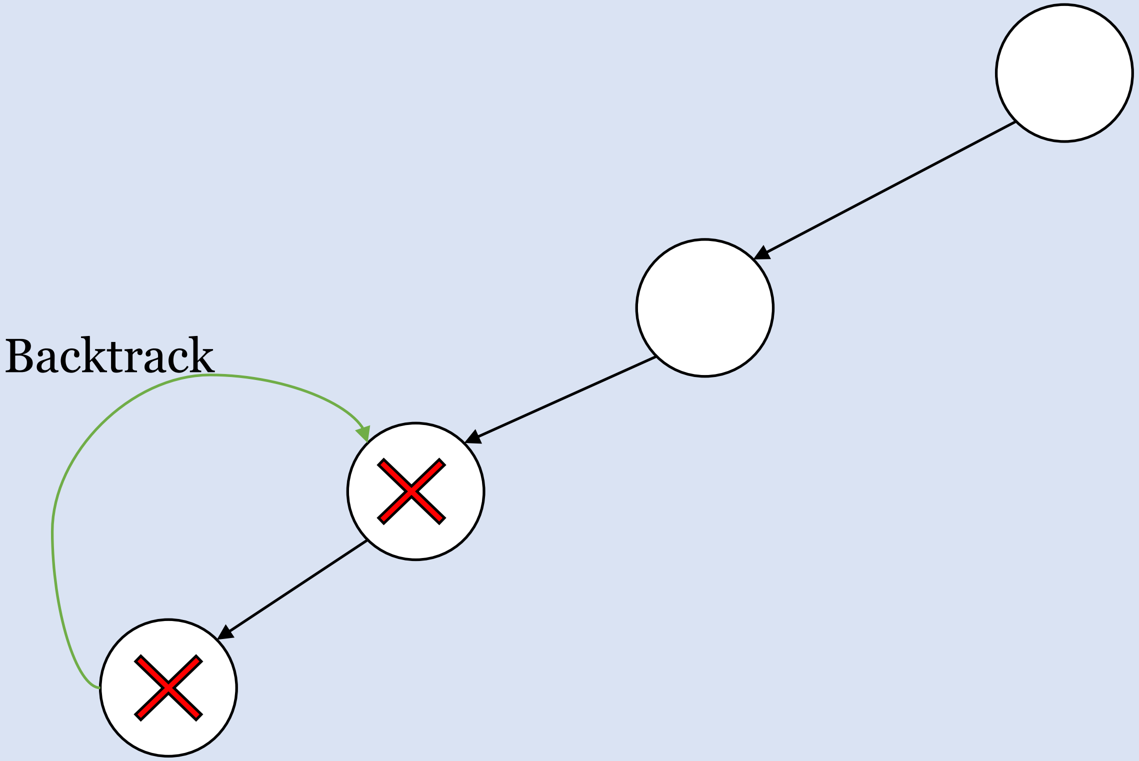
The Big Picture of Backtracking



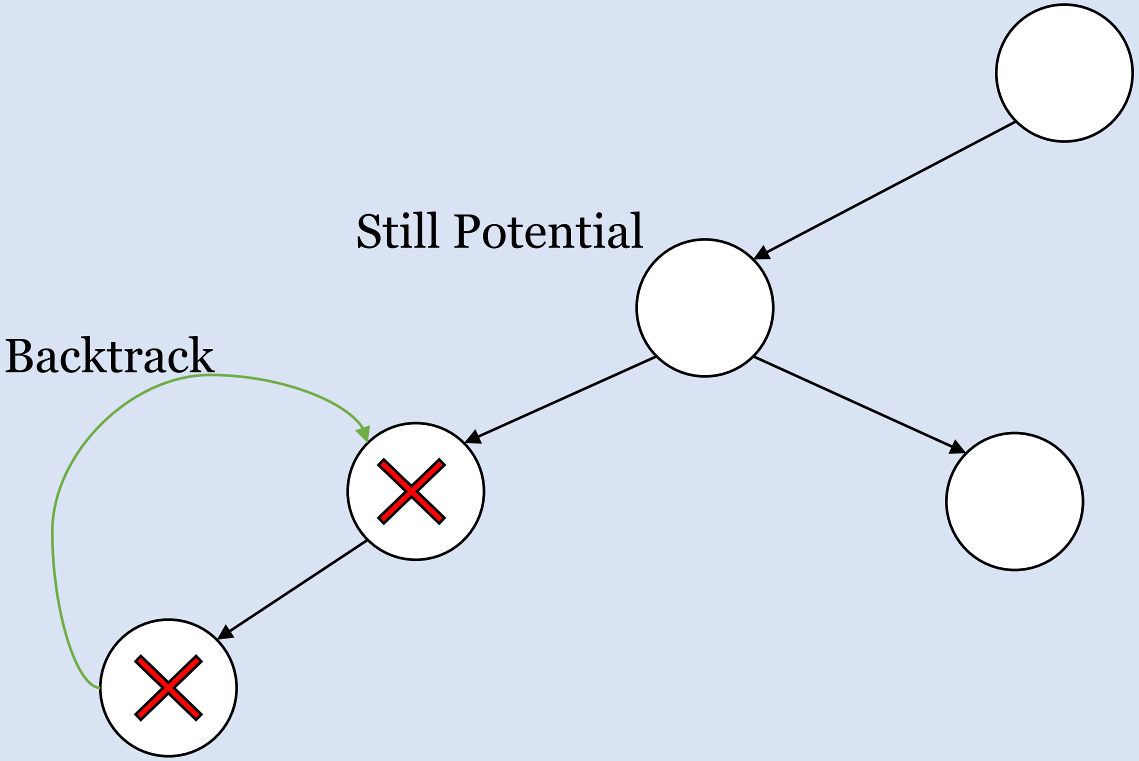
The Big Picture of Backtracking



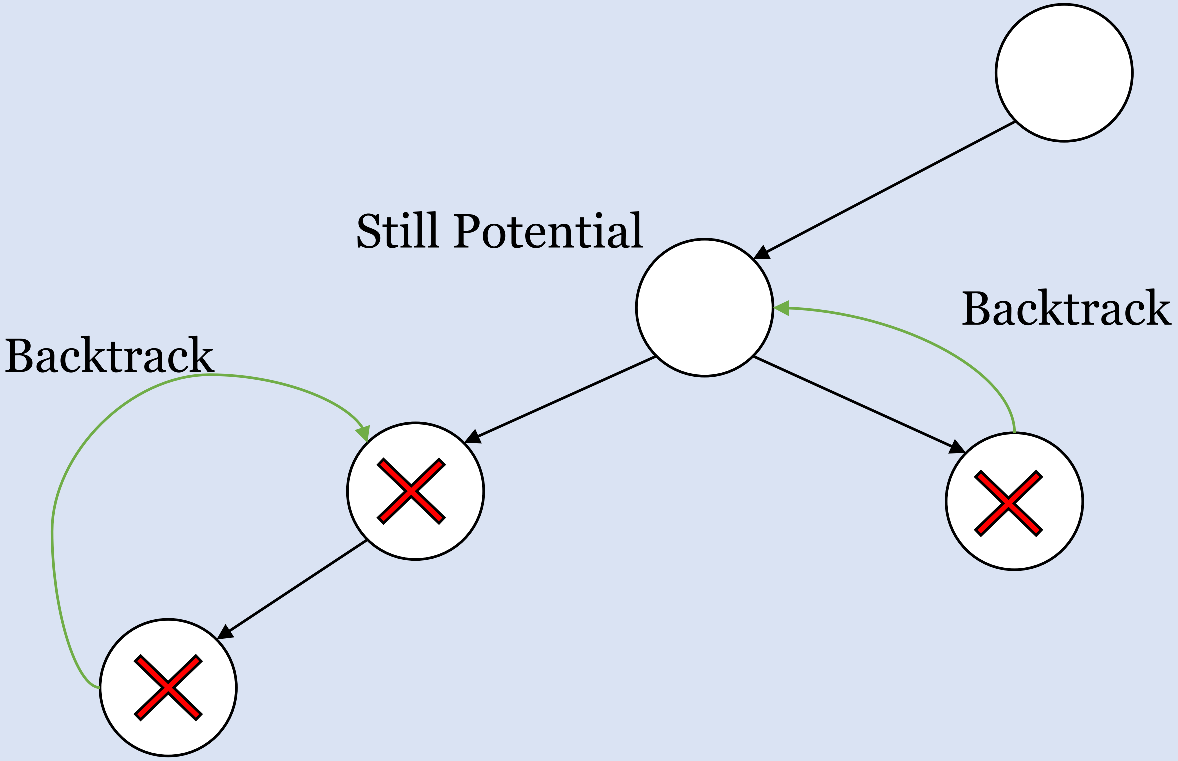
The Big Picture of Backtracking



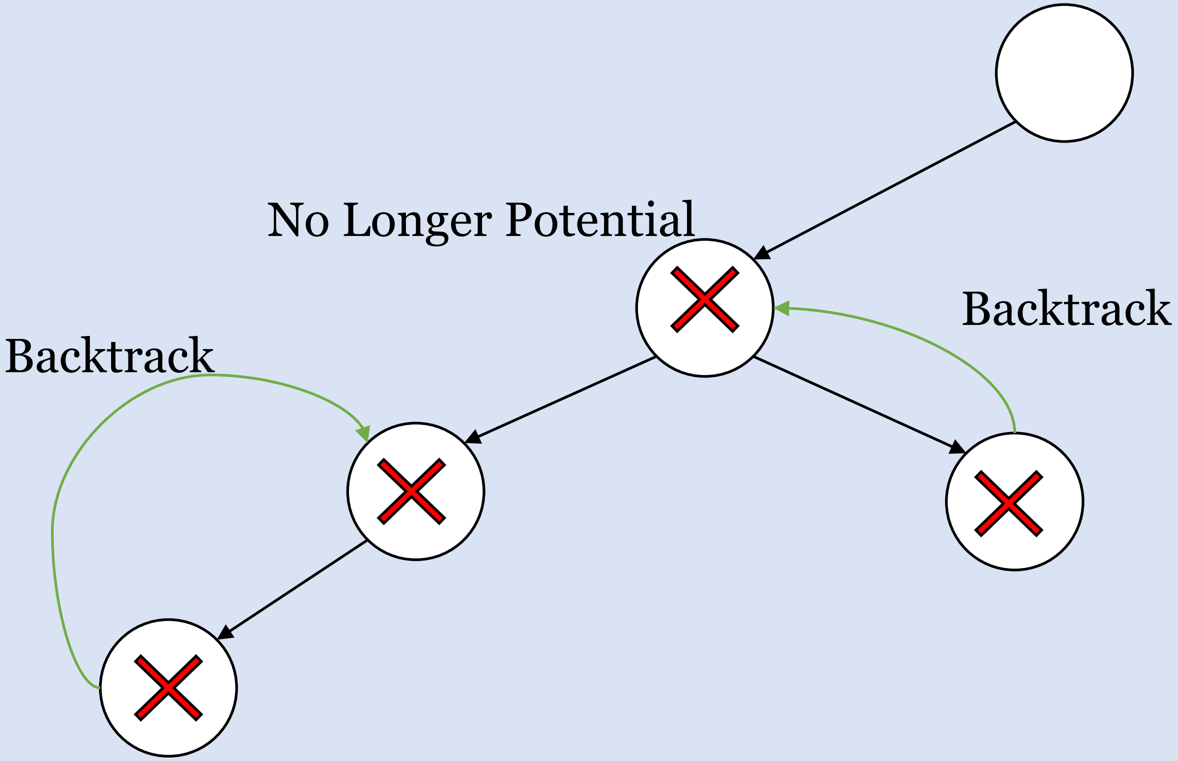
The Big Picture of Backtracking



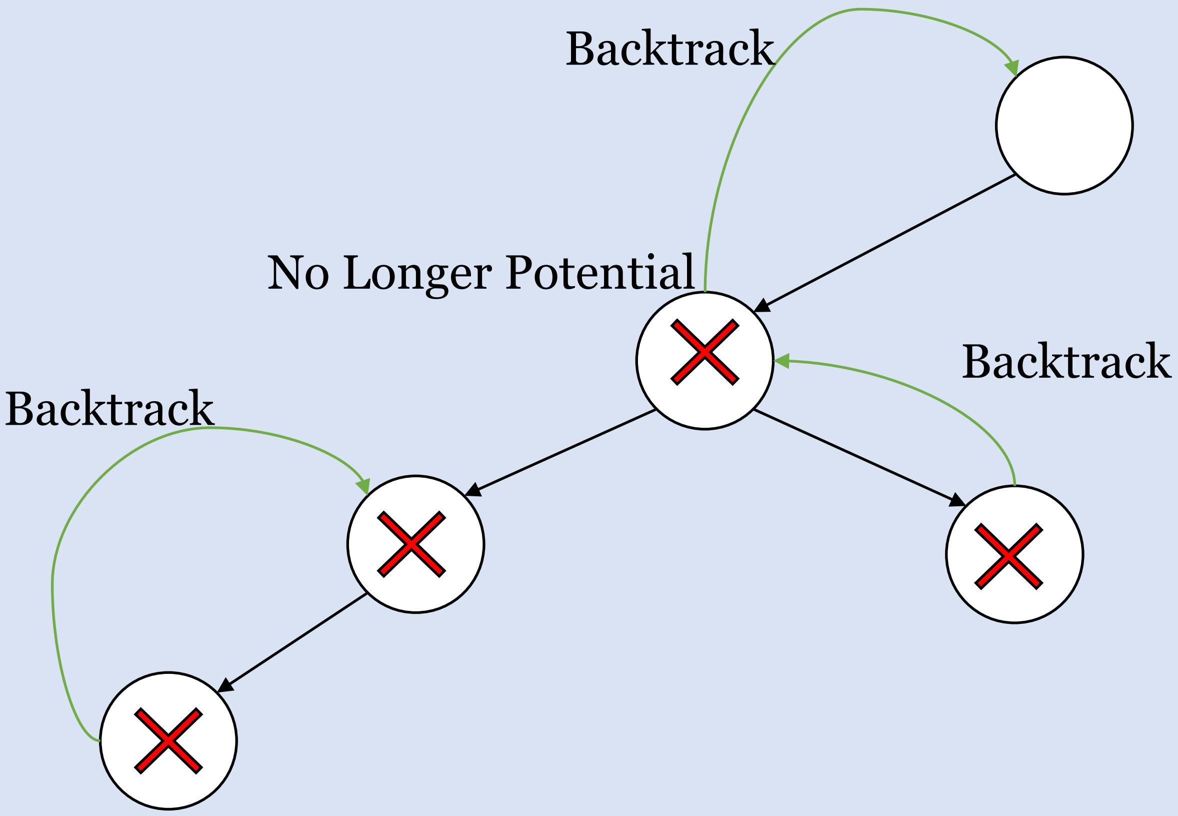
The Big Picture of Backtracking



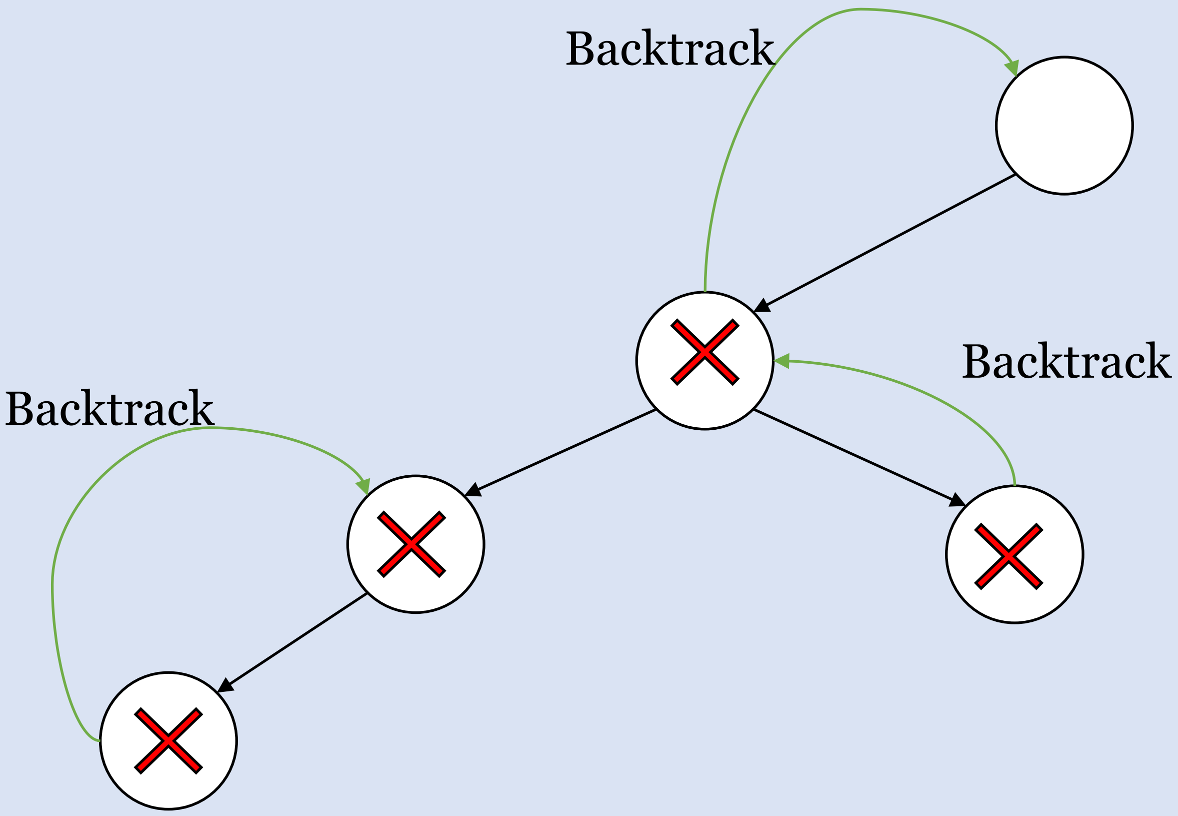
The Big Picture of Backtracking



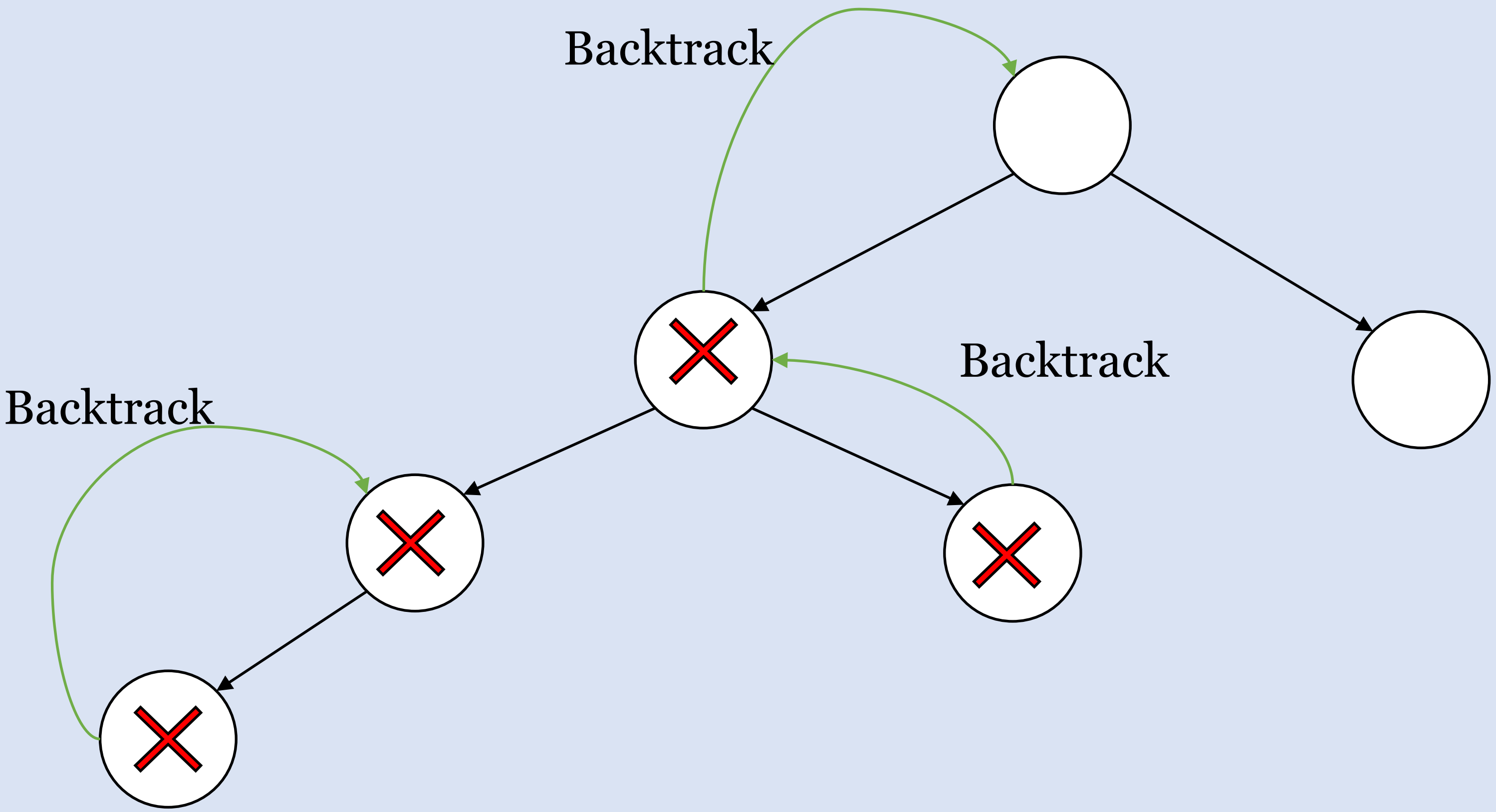
The Big Picture of Backtracking



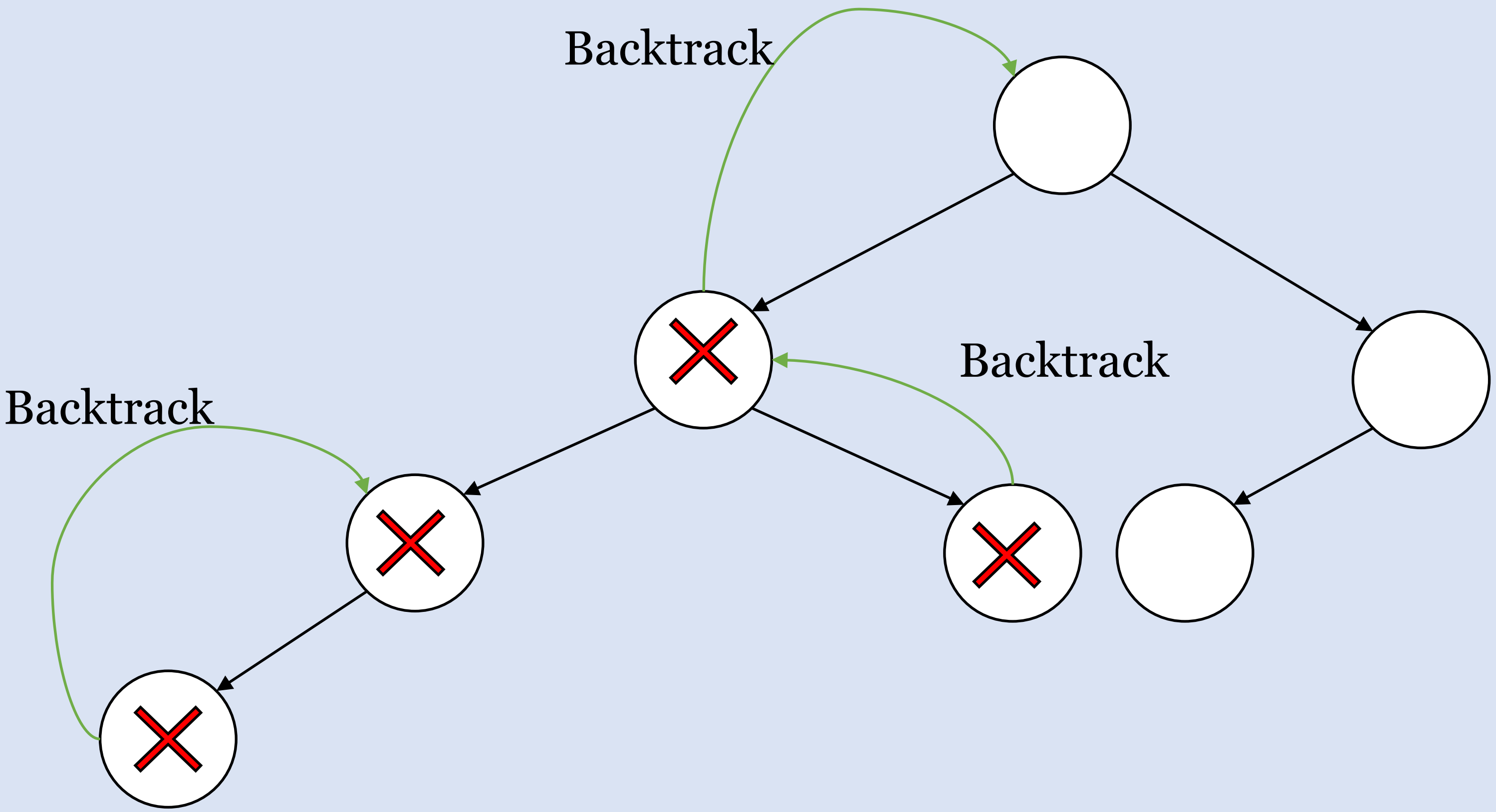
The Big Picture of Backtracking



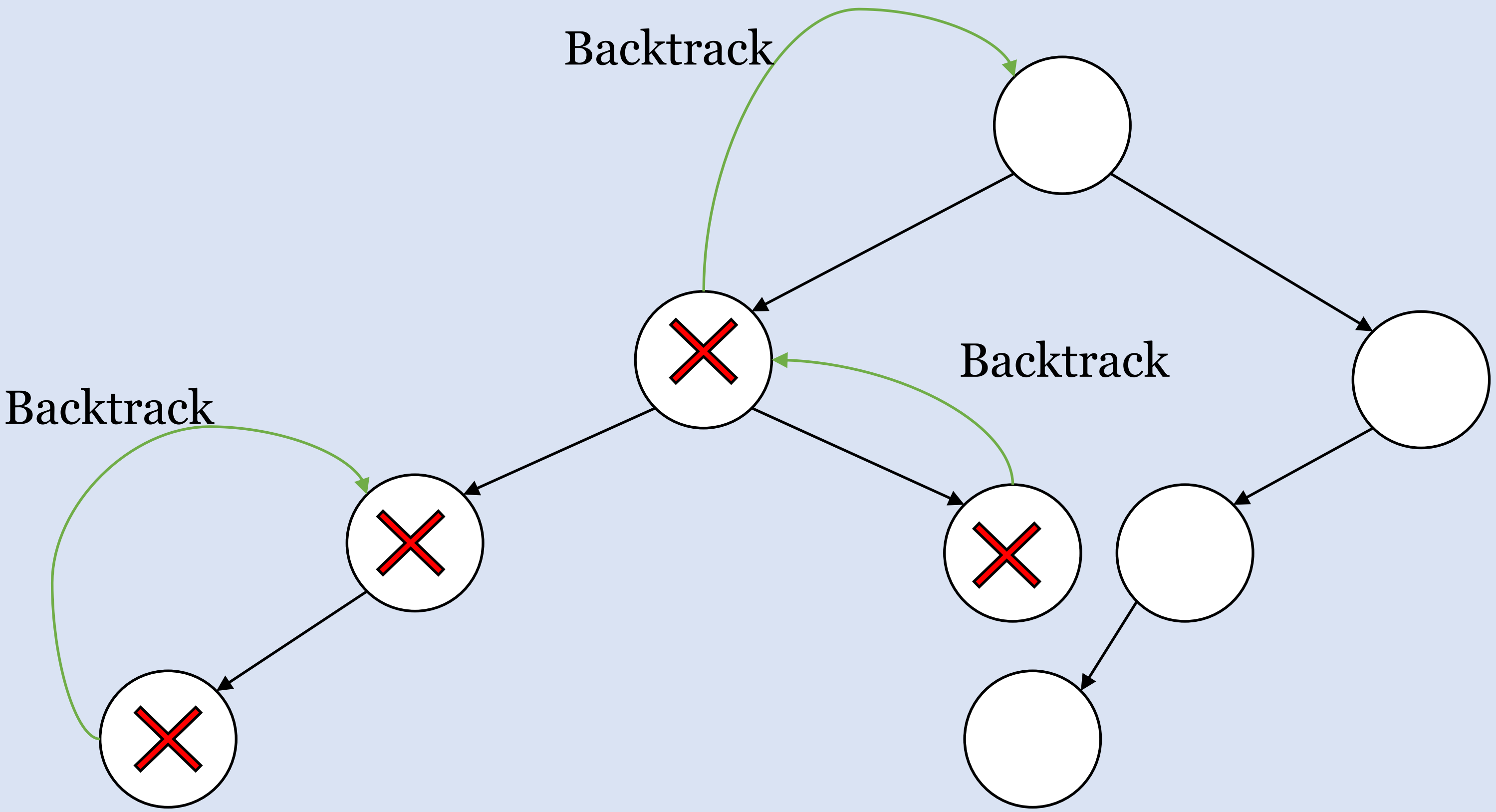
The Big Picture of Backtracking



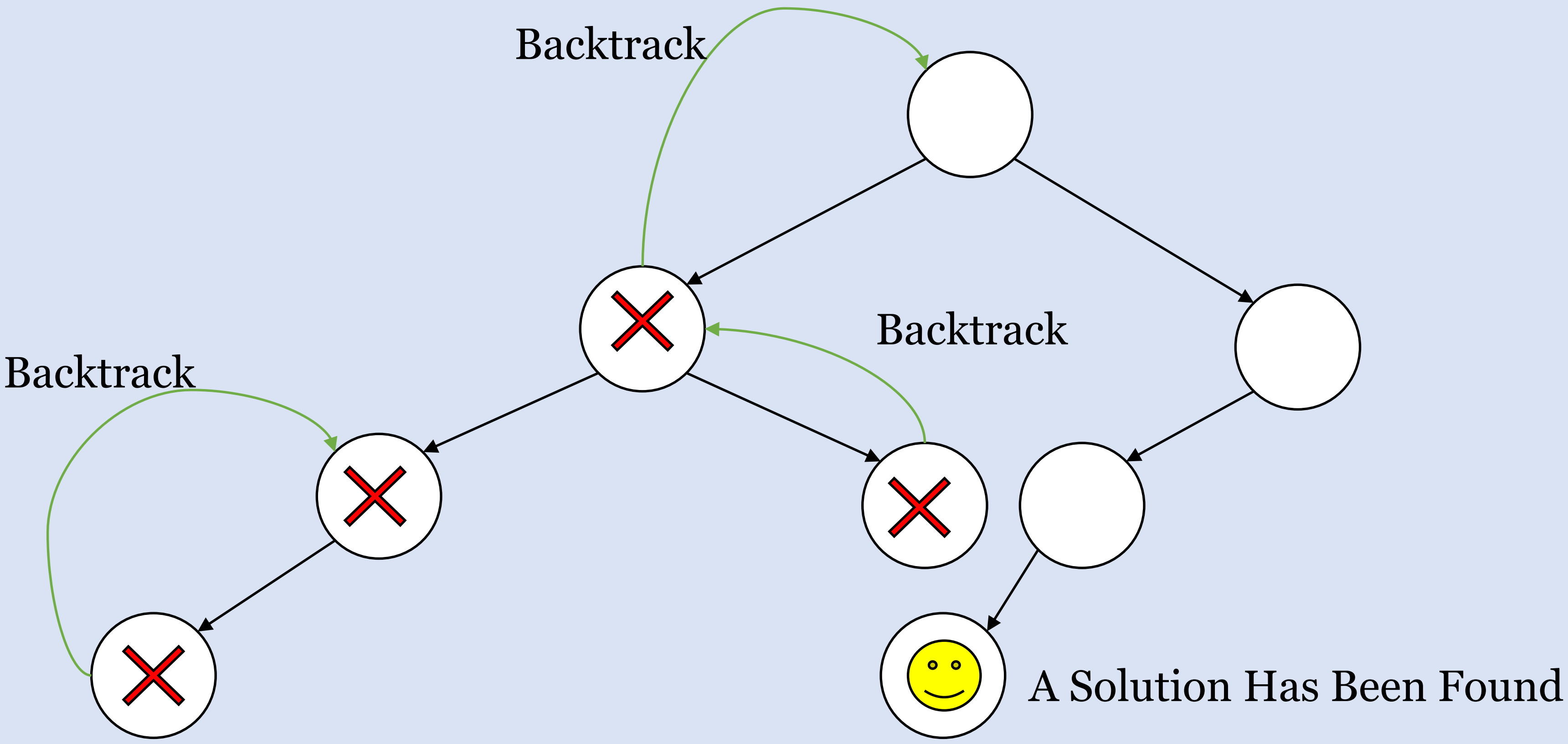
The Big Picture of Backtracking



The Big Picture of Backtracking



The Big Picture of Backtracking



Based on observation of Backtracking what kind of problems can we apply this technique too? From the description, are there limitations?

Let’s Observe the Classic N-Queens Problem

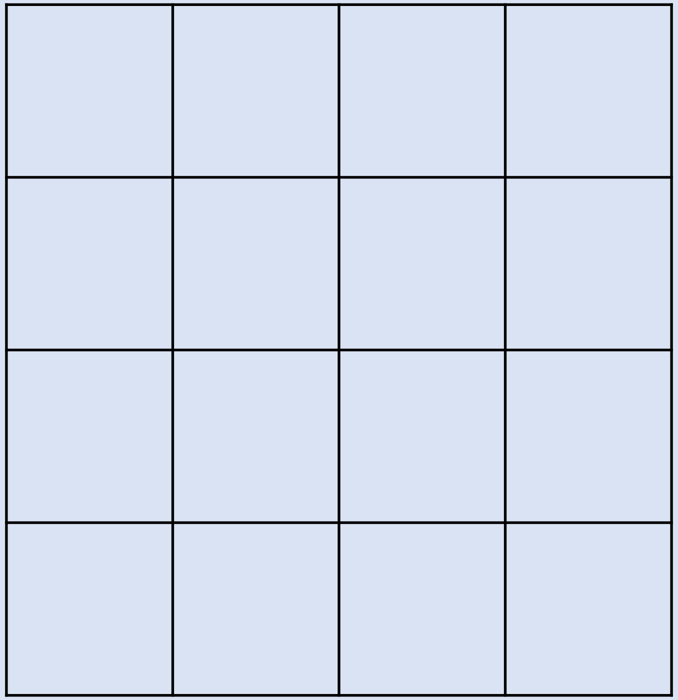
Problem Definition

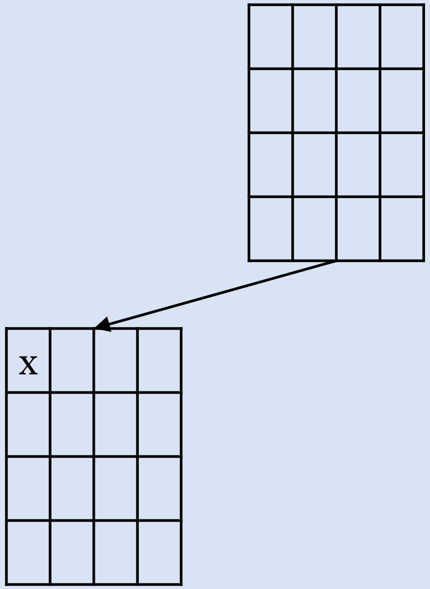
The n-queens problem is to place *n* queens on an *n x n* board so that no two queens are in the same row, column, or diagonal.

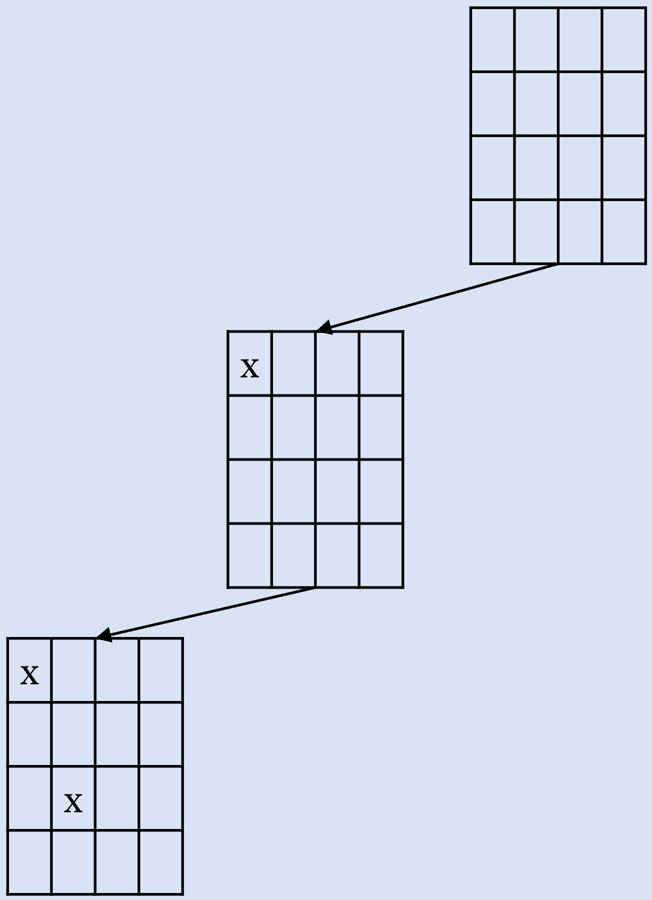
Sample
4 queens on 4 x 4

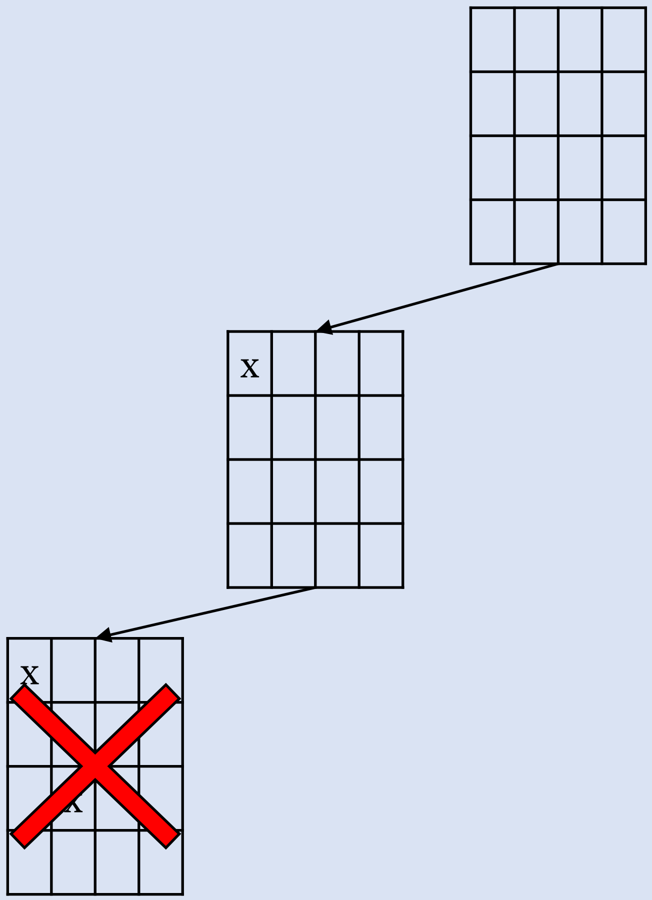
		X	
X			
			X
	X		

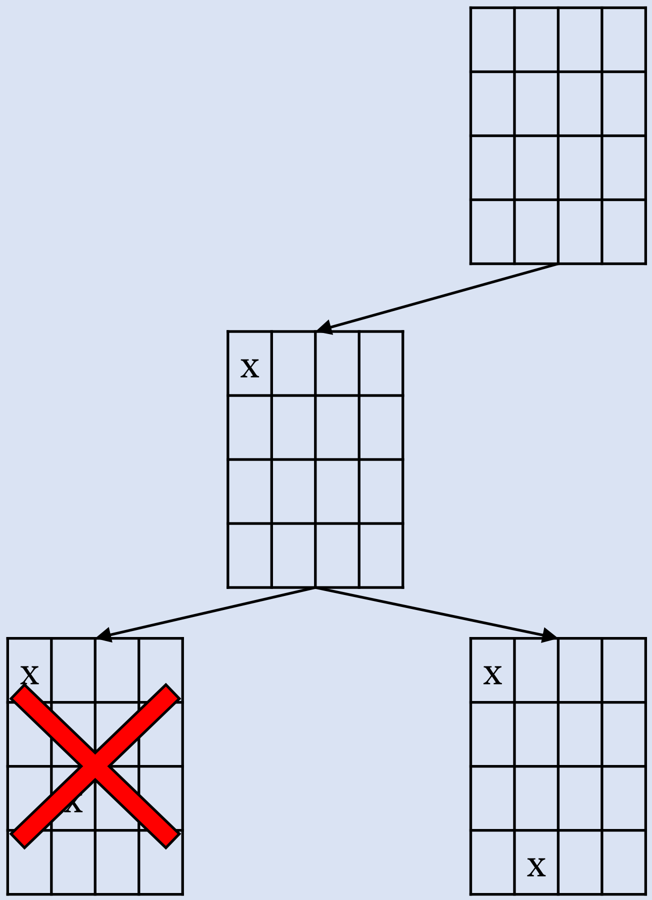
Search Tree Example of 4 queens on 4 x 4

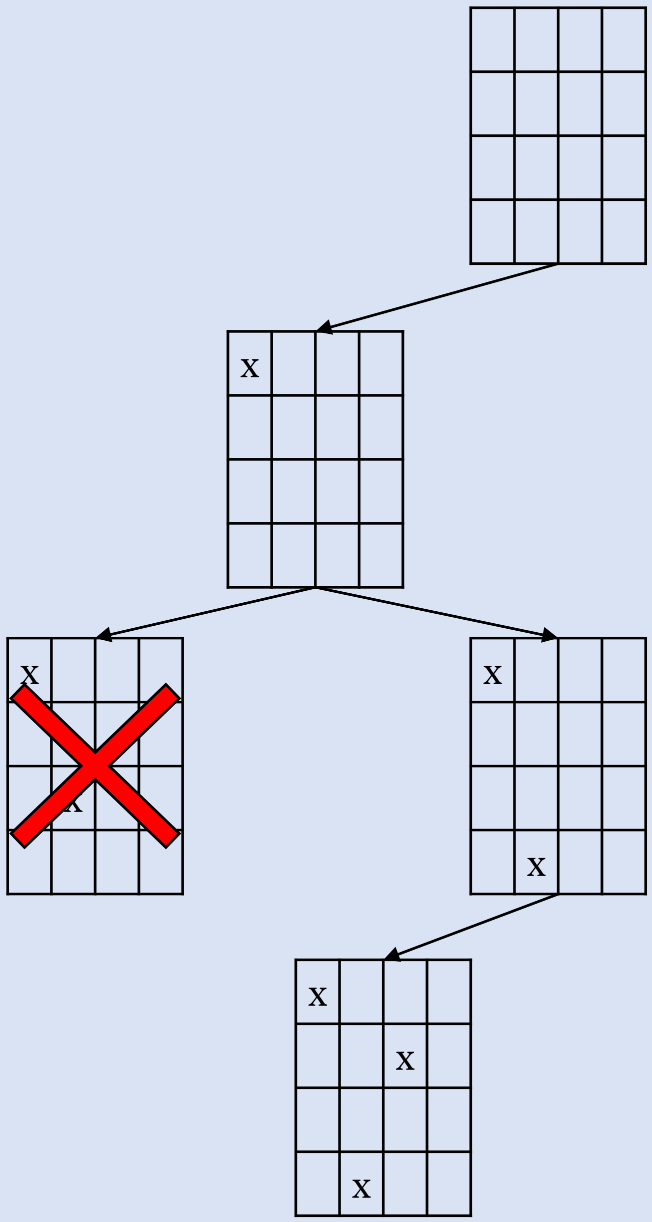


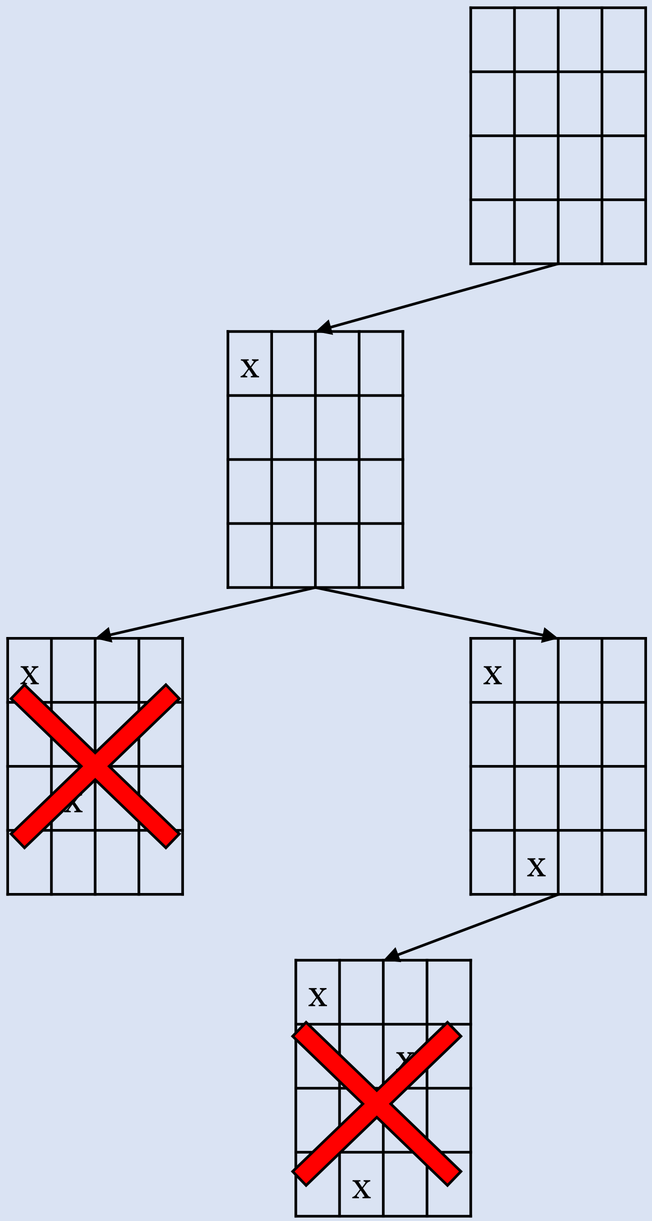

x

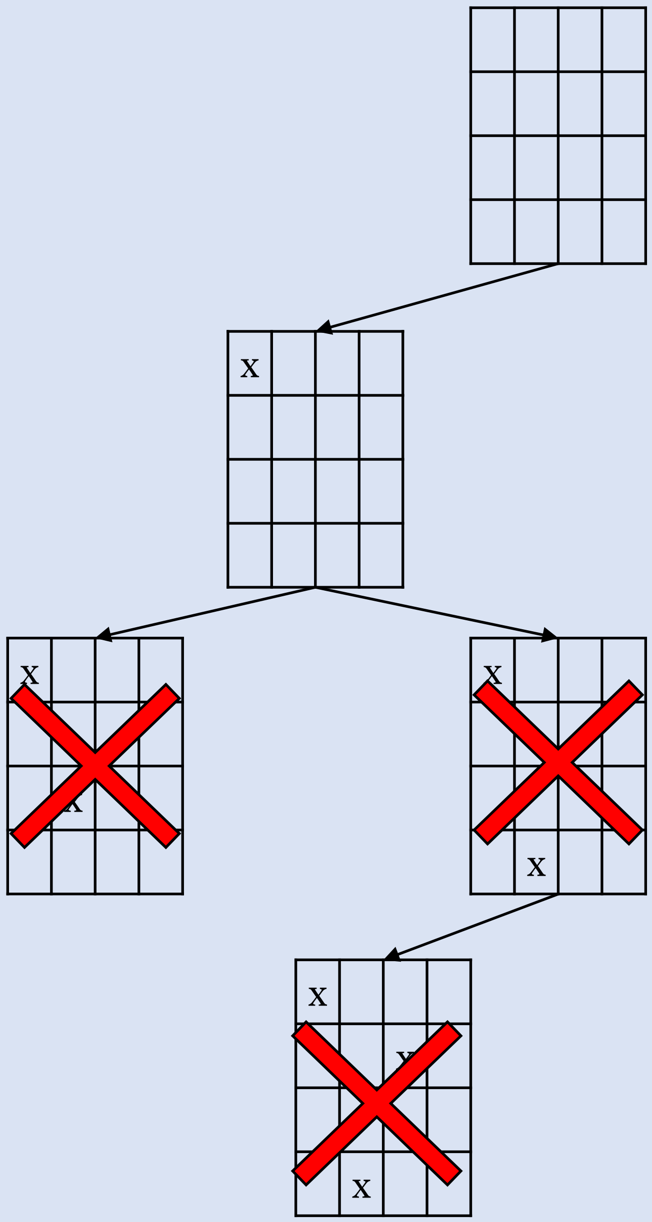

x
x
x

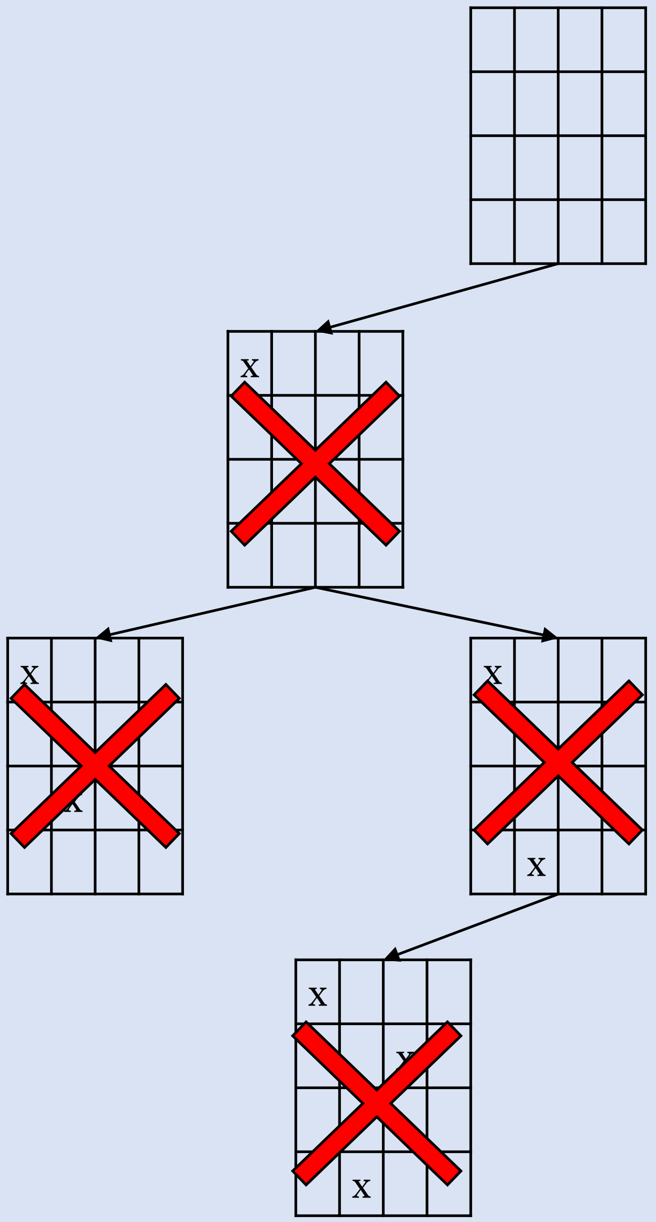

x
x

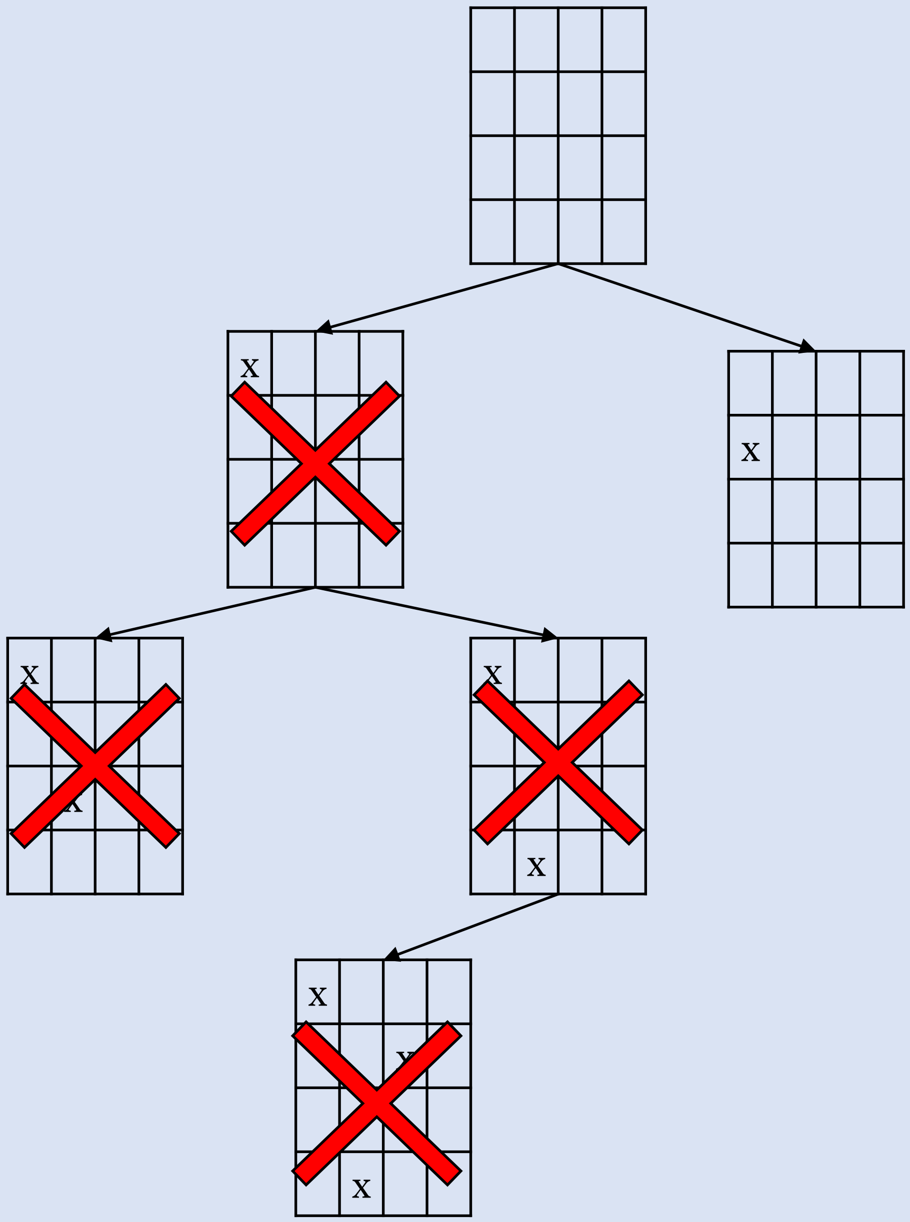

x
x
x
x
x

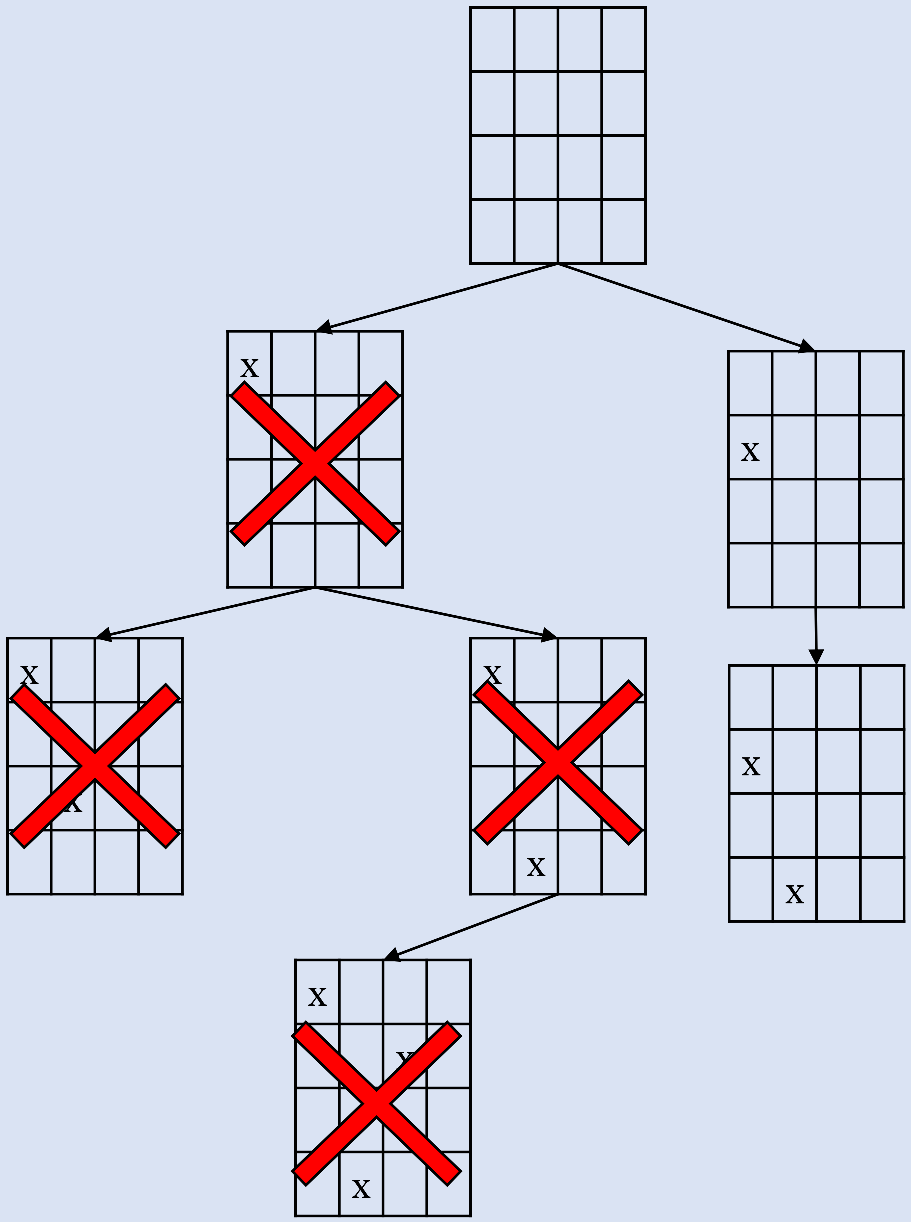

x
x
x
x
x
x
x
x
x

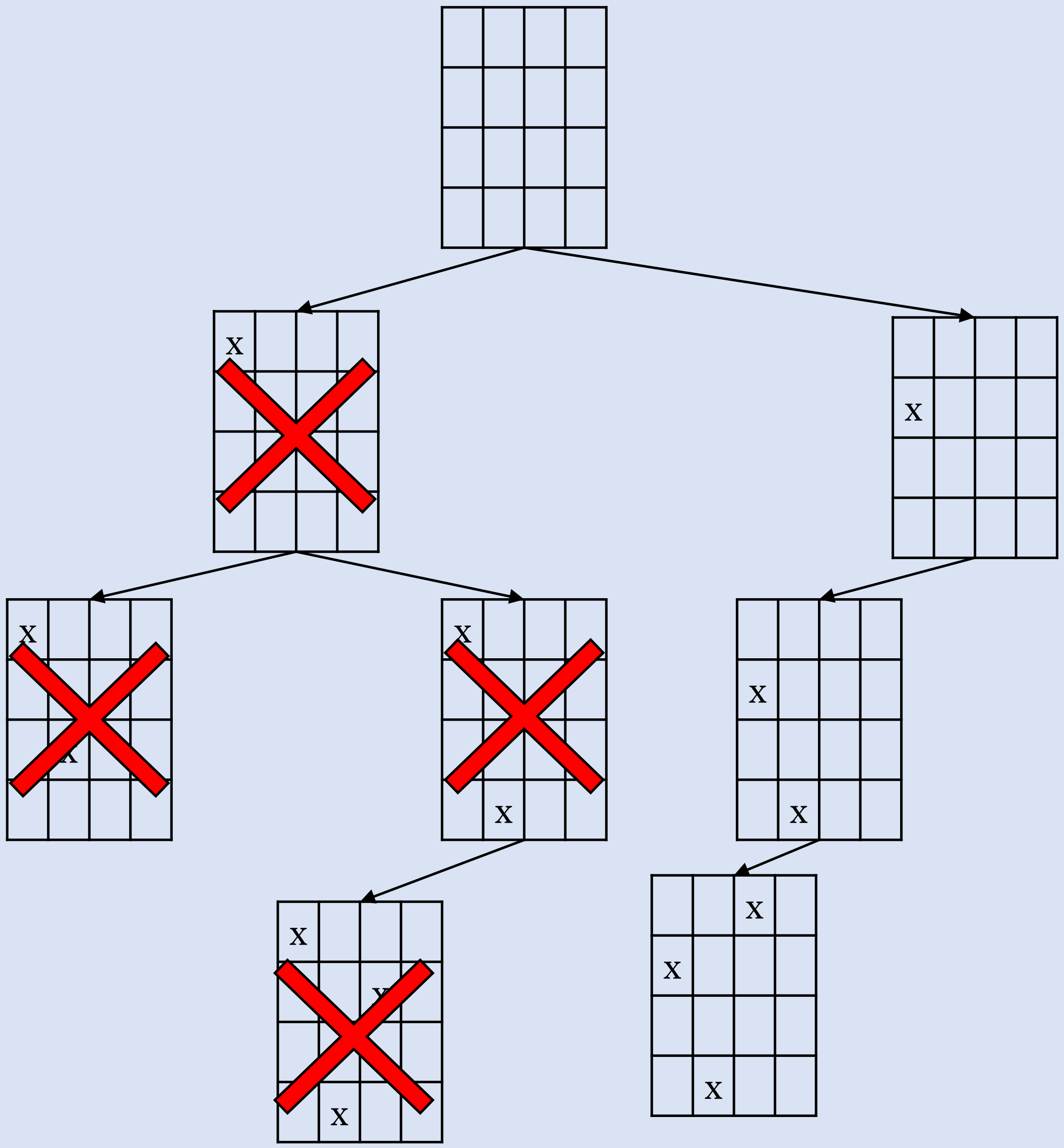


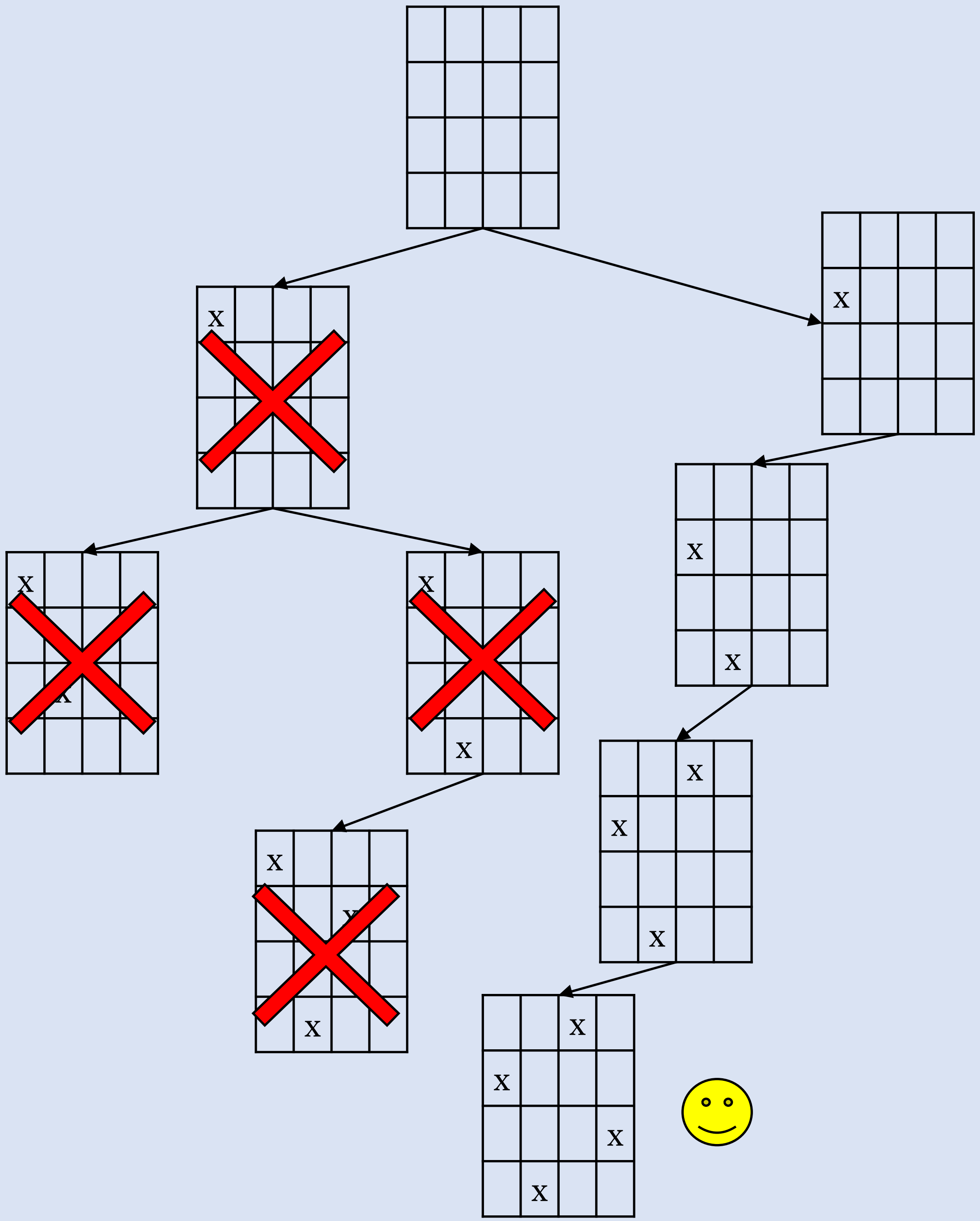



x
x
x
x
x
x
x
x









Now Lets Derive the Backtracking Algorithm for N-Queens

Now Lets Derive the Running Time Analysis

The Skeleton Backtracking Algorithm

- This is a skeleton backtracking algorithm you can utilize in design your backtracking solutions to various problems
- bound() tests for a partial solution

```
backtrack(n)
    rbacktrack(1,n)
```

```
rbacktrack(k, n)
    for each x[k] ∈ S
        if (bound(k))
            if (k == n)
                //output the solution
               for i = 1 to n
                   print(x[i] + " ")
                println()
            else
                rbacktrack(k+1,n)
```