Backtracking

- Backtracking is a recursive strategy that constructs a solution to a computational problem incrementally, at one small piece at a time.
- When the algorithm needs to decide between multiple options to the next component of the solution, it evalues **every** alternative recursively and chooses the best one.

2.1 N-Queens

- The problem is to place n queens on an $n \times n$ chessboard in a way that no two queens are attacking each other.
  - No two queens are in the same row, column, or diagonal.

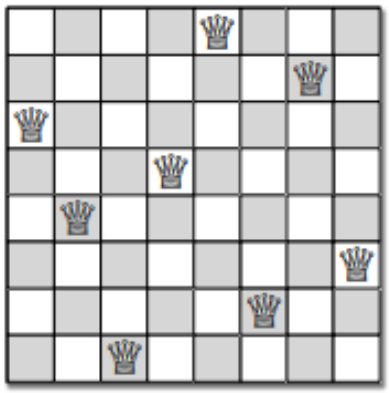

Figure 2.1. Gauss's first solution to the 8 queens problem, represented by the array $[5, 7, 1, 4, 2, 8, 6, 3]$

- Gauss's recursive strategy:
  - Place the queens on the board one row at a time starting from the top.
  - To place the rth queen in the rth row, try all n squares in the row from left to right using a loop. If a square is being attacked by an earlier queen, then ignore the square.
  - If the square is a valid placement, then we temporarily place a queen on there and recursively check later rows for a consistent placement of the queens.
- Pseudocode:

```
PlaceQueens(Q[1..n], r):
        if r = n + 1:
                print Q[1..n]
        else:
                for j = 1 to n:
                        legal = True
                        for i = 1 to r - 1:
                                if (Q[i]=j) or (Q[i]=j+r-1) or (Q[i]=j-r+i):
```

```
                    legal = False
        if legal:
            Q[r] = j
            PlaceQueens(Q[1..n], r+1)
```

- The algorithm recursively enumerates through *all* complete n-queens solutions that are consistent with a given partial solution.
- We represent the positions of the queens with the array $Q[1..n]$ where $Q[i]$ indicates which square in row i contains a queen.
- r is the index of the first empty row and $Q[1..r-1]$ contains the positions of all the queens before r.
- The outer for-loop checks for all possible placements of a queen in the row r.
- The inner for-loop checks if a temporary candidate placement on row r is consistent with the the queens that are already placed on the first $r-1$ rows.
- Our initial call of the algorithm to solve the entire tree would be `PlaceQueens(Q[1..n], 1)`.
- N-Queens solution as a recursion tree for the n=4:

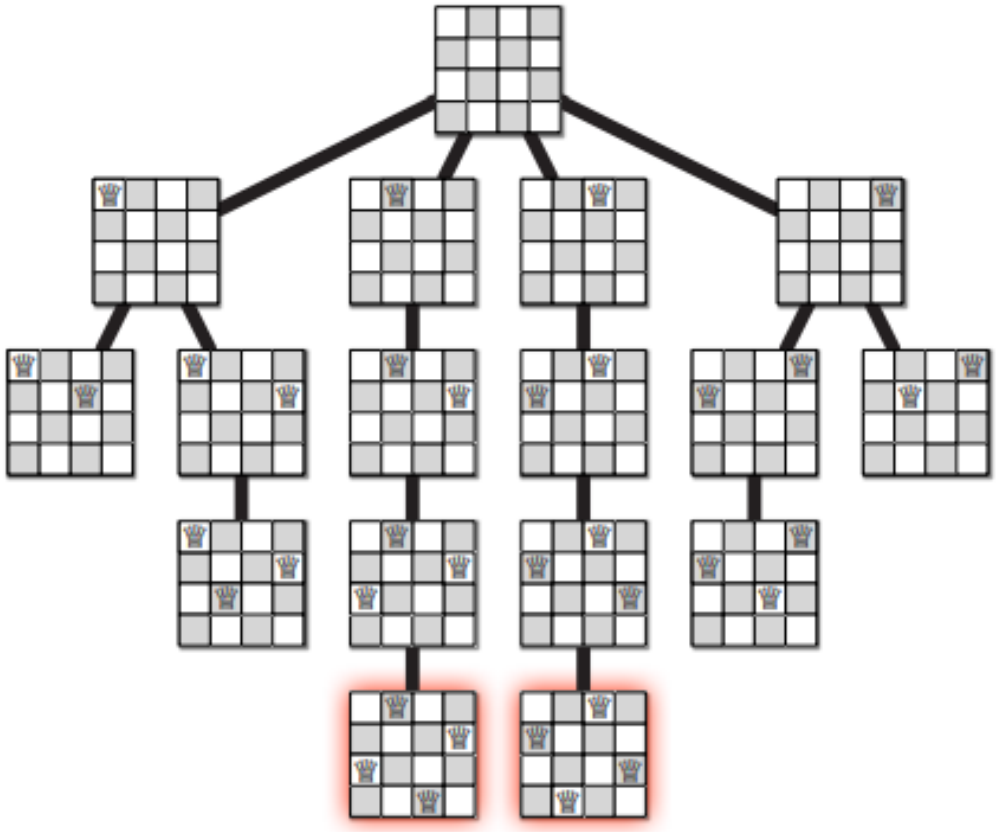

Figure 2.3. The complete recursion tree of Gauss and Laquière's algorithm for the 4 queens problem.

- Each node is a recursive subproblem and also a legal partial solution.
- Each level is the row being examined (the root is r=0 and the bottom leaves are r=4)
- The edges correspond to recursive calls.

- The leaves are the partial solutions that cannot be extended because there are no more valid placements of queens with the current board (either because every possible placement is attacked by previous queens or all the rows are already filled).

2.2 Game Trees

- For every two-player game without randomness or hidden information that ends after a finite number of moves, there exists a backtracking algorithm that can play the game perfectly.
    - If two players play perfectly, and it is possible to win against them, then an algorithm will tell you how to win.
- A **state** of the game is the locations of all the pieces and the identity of the current player.
- The states can all be connected together into a **game tree**.
  - The edges from state x to state y exists if and only if the current player in state x can legally move to state y.

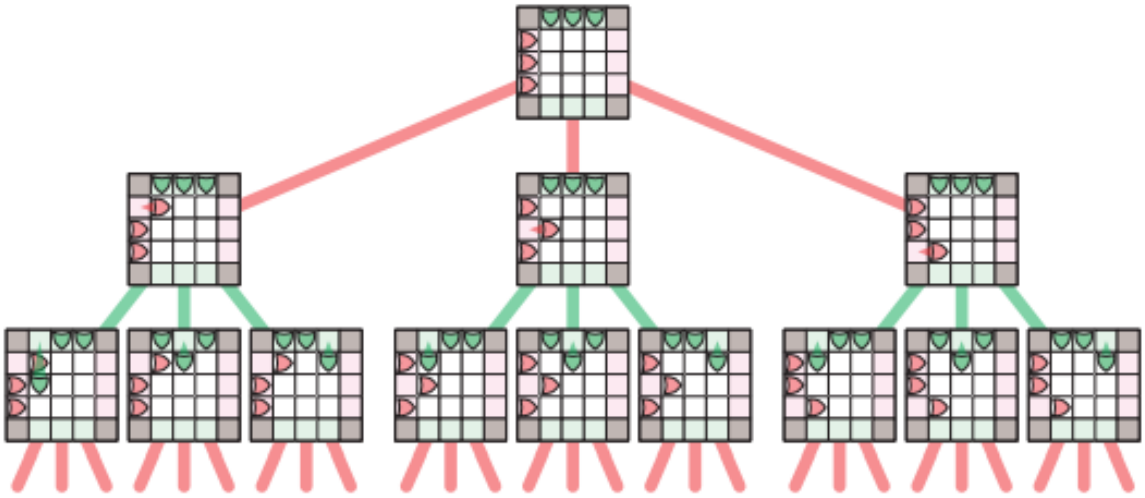

Figure 2.5. The first two levels of the fake-sugar-packet game tree.

(you should read the textbook for the example in this tree)

- To find a solution to the game and navigate through the game tree, we recursively define a game state to be **good** or **bad**.
    - a game state is **good** if the current player has already won, or if the current player can move to a bad state for the opposing player.
        - a non-leaf node is **good** if it has at least one bad child (the current player can choose to move to a bad game state for the opposing player).
    - a game state is **bad** if the current player has already lost, or if every single available move leads to a good state for the opposing player.
        - a non-leaf node is **bad** if every child is good (the current player can only move to a good game state for the opposing player).
- By induction, any player that finds the game in a good state on their turn can win the game even if their opponent plays perfectly.

    - When starting in a bad state, a player can only win if their opponent makes a mistake.
    - This is ONLY if the game has a finite number of moves.
- Pseudocode:

```
PlayAnyGame(X, player):
        if player has already won in state X:
                return Good
        if player has already lost in state X:
                return Bad
        for all legal moves X to Y:
                if PlayAnyGame(Y, ~player) = Bad:
                        return Good     // X to Y is a good move
        return Bad    // There are no good moves
```

- `X` is the current game state, and `Y` is the next potential state.
- `player` is the current player.
- The backtracking algorithm is just a Depth-First Search of the game tree.
- The game tree is the recursion tree of the algorithm.
- In practice, game trees are extremely huge to the point that traversing it entirely is not feasible, so game programs will use other algorithms to prune the game tree to ignore states that are obviously good or bad, or better or worse than other game states by cutting off the tree at a certain depth.

2.3 Subset Sum

- Subset Sum problem: Given a set X of positive integers and a target integer T, is there a subset of elements in X that add up to T?
    - There can be more than one valid answer, if one exists.
- Example: Let $X = \{8, 6, 7, 5, 3, 10, 9\}$ and $T = 15$.
  - The answer is `True` because

$$\begin{aligned} 8 + 7 &= 15, \quad \{8, 7\} \subseteq X \\ 7 + 5 + 3 &= 15, \quad \{7, 5, 3\} \subseteq X \\ 6 + 9 &= 15, \quad \{6, 9\} \subseteq X \\ 5 + 10 &= 15, \quad \{5, 10\} \subseteq X \end{aligned}$$

  are all subsets of X whose sums add up to T.
- Example: Let $X = \{11, 6, 5, 1, 7, 13, 12\}$ and $T = 15$.
    - The answer would be `False` because no subsets exist such that the sum will equal $T = 15$.
- Two trivial cases:
    - If $T = 0$, then we return `True` because we can take $\emptyset \subseteq X$ where all the elements add up to 0.

    - If $T < 0$, then we return `False` because no such subset exists where the elements add up to a negative number.
- The general case: Consider an element $x \in X$. Then, there exists a set$A \subseteq X$ that sums to T if and only if one of the following statements are true:
    - There exists an A such that $x \in A$ and whose sum is T.
    - There exists an A such that $x \notin A$ and whose sum is T.
    - In the first case, $A \subseteq X - \{x\}$ where A sums to $T - x$.
    - In the second case, $A \subseteq X - \{x\}$ where A sums to T (because $x \notin A$).
- We can then reduce the problem to two smaller subproblems by solving the first and second case.
- Pseudocode:

```
SubsetSum(X, T):    //checks if any subset of X sums to T
        if T = 0:
                return True
        else if T<0 or X is empty:
                return False
        else:
                x = any element of X
                with = SubsetSum(X-{x}, T-x)
                without = SubsetSum(X-{x}, T)
                return (with or without)    //boolean logic
```

Correctness

- If $T = 0$, then $\emptyset \subseteq X$ sums to 0, so it outputs `True`, which is correct.
- If $T < 0$ or $\emptyset = X$, then it outputs `False` because no subset of X sums to T, which is correct.
- Otherwise, we have our general case; Our algorithm checks if there exists a subset that sums to T. This subset either contains $X[n]$ or it doesnt, and the algorithm correctly checks for each of those possibilities.

Analysis

Variants

2.4 The General Pattern

2.5 Text Segmentation (Interpunctio Verborum)

2.6 Longest Increasing Subsequence

2.7 Longest Increasing Subsequence, Take 2

2.8 Optimal Binary Search Trees