

# 10 - Backtracking

[KOMS119602] & [KOMS120403]

Design and Analysis of Algorithm (2021/2022)

Dewi Sintiar

Prodi S1 Ilmu Komputer  
Universitas Pendidikan Ganesha

Week 9-13 May 2022

# Table of contents

- Principal of Backtracking
- State-space tree in backtracking algorithm
- $n$ -Queens problem
- Hamiltonian circuit problem
- Subset-sum problem

# Principal of Backtracking

- The *exhaustive-search* technique suggests generating all candidate solutions and then identifying the one (or the ones) with a desired property.
- Backtracking algorithm improves exhaustive search.
- In exhaustive search, all possible solutions are explored and evaluated one-by-one
- In backtracking, we do not examine all possibilities, only the possibilities that lead to the solution. Other nodes that do not lead to the solution are pruned.

## Central idea:

To cut off a branch of the problems state-space tree, as soon as we can deduce that it cannot lead to a solution.

# Principal of Backtracking

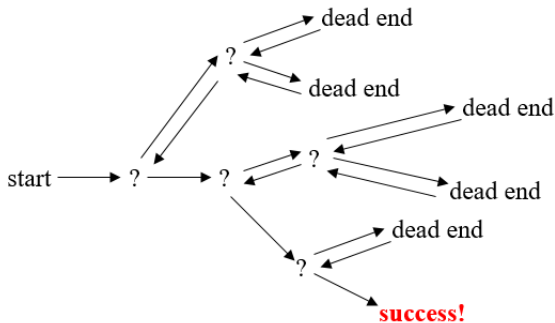


Figure: Illustration of backtracking (sumber: <https://miro.medium.com/>)

# Representation of solution

- **Representation**: an output can be thought of as  $n$ -tuple  $(x_1, x_2, \dots, x_n)$  where each coordinate  $x_i$  is an element of some finite linearly ordered set  $S_i$ .
- **Tuples**: all solution tuples can be of the same length (the  $n$ -queens and the Hamiltonian circuit problem) and of different lengths (the Subset-sum problem).

# Backtracking in DFS

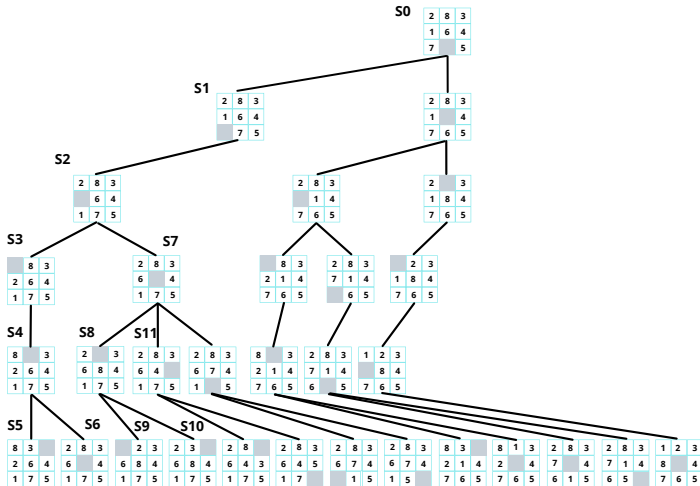
Backtracking in DFS is used in solution-searching problems that have many possibilities of solution.

The solution is obtained by looking in a depth-first approach

- You do not have enough information to know the next step.
- Each decision leads you to several/many new choices.
- Several sequence of choices may be the problem's solution.

In DFS, backtracking is used as a methodological way to try several sequences of decision.

## Example of backtracking in DFS



# State-space tree



# State-space tree (1)

Backtracking can be seen as searching in a tree from the root to the leaves (solution node).

## State-space tree

It is a tree representing all the possible states (solution or non-solution) of the problem from the root as an initial state to the leaf as a terminal state.

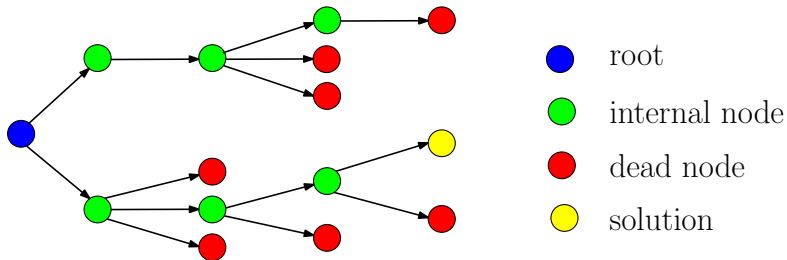
A backtracking algorithm generates, explicitly or implicitly, a **state-space tree**:

- its nodes represent partially constructed tuples with the first  $i$  coordinates defined by the earlier actions of the algorithm;
- if such a tuple  $(x_1, x_2, \dots, x_i)$  is not a solution, the algorithm finds the next element in  $S_{i+1}$  that is consistent with the values of  $(x_1, x_2, \dots, x_i)$  and the problems constraints, and adds it to the tuple as its  $(i + 1)$ st coordinate;
- if such an element does not exist, the algorithm backtracks to consider the next value of  $x_i$ , and so on.

# State-space tree (2)

- **Root** represents an initial state before the search begins;
- **Internal nodes**
  - the nodes of the first level in the tree represent the choices made for the first component of a solution;
  - the nodes of the second level represent the choices for the second component;
  - and so on...;
- **Leaves** represent either non-promising dead ends or complete solutions found by the algorithm.

## State-space tree (3)



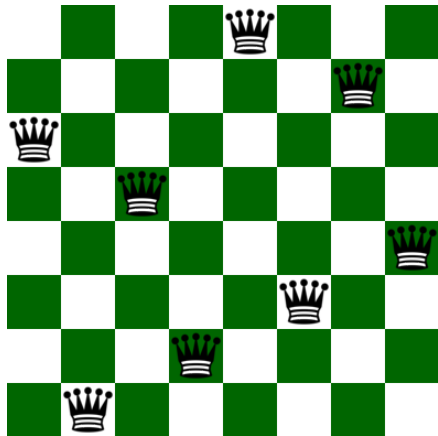
### Types of nodes in the state-space tree

- **Promising node**: corresponds to a partially constructed solution that may still lead to a complete solution;
- **Non-promising node**: dead node

# State-space tree (4)

- The solution is searched by generating the **state nodes**, so that it produces paths from the root to the leaves;
- To generate the nodes, the DFS rule is followed;
- The generated nodes are called **live node**;
- The live node being expanded is called **expand-node**;
- Each time the expand-node is expanded, the generated path gets longer;
- Function that is used to “kill” an expand-node is called **bounding function**;
- When a node is killed, then automatically all its children nodes are pruned;
- If the paths-generation ends up with dead node, the searching is **backtrack** to the parents nodes;
- These parents nodes become the new expand-nodes;
- The searching is stopped if we find a solution.

# $n$ -Queens problem

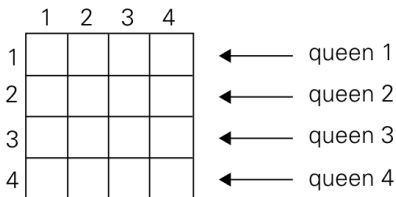


# $n$ -Queens problem

## Problem

*Place  $n$  queens on an  $n \times n$  chessboard, so that no two queens attack each other (by being in the same row, in the same column, or in the same diagonal).*

- For  $n = 1$ , the problem has a trivial solution
- For  $n = 2, 3$ , the problem has no solution
- What if  $n = 4$ ?

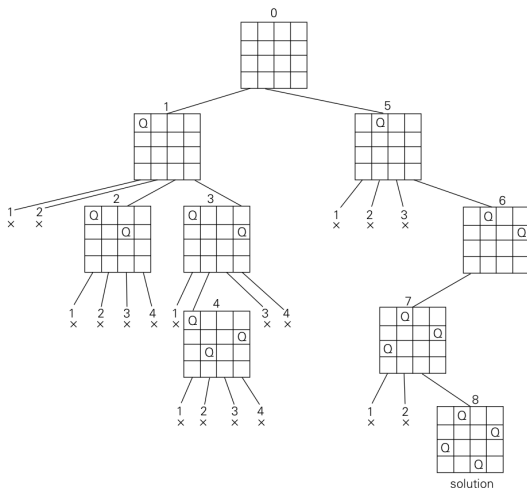


## START

- ① begin from the leftmost column
- ② if all the queens are placed, return true/ print configuration
- ③ check for all rows in the current column
  - ① if queen placed safely, mark row and column; and recursively check if we approach in the current configuration, do we obtain a solution or not
  - ② if placing yields a solution, return true
  - ③ if placing does not yield a solution, unmark and try other rows
- ④ if all rows tried and solution not obtained, return false and backtrack

## END

# $n$ -Queens problem



**Figure:** State-space tree of solving the four-queens problem by backtracking;  $\times$  denotes an unsuccessful attempt to place a queen in the indicated column. The numbers above the nodes indicate the order in which the nodes are generated.



# $n$ -Queens problem

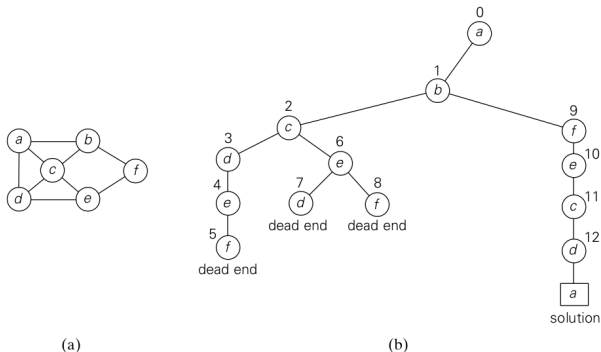
- 1 We start with the empty board and then place queen 1 in the first possible position of its row, which is in column 1 of row 1.
- 2 Then we place queen 2, after trying unsuccessfully columns 1 and 2, in the first acceptable position for it, which is square (2,3), the square in row 2 and column 3.
- 3 This proves to be a dead end because there is no acceptable position for queen 3.
- 4 So, the algorithm backtracks and puts queen 2 in the next possible position at (2,4).
- 5 Then queen 3 is placed at (3,2), which proves to be another dead end.
- 6 The algorithm then backtracks all the way to queen 1 and moves it to (1,2).
- 7 Queen 2 then goes to (2,4), queen 3 to (3,1), and queen 4 to (4,3), which is a solution to the problem.

# Other problems

# Hamiltonian circuit problem

## Problem

Given a connected graph  $G$ , find a Hamiltonian circuit in  $G$ . (Recall that a Hamiltonian circuit is a circuit that visits all vertices of  $G$  exactly once.)



**Figure:** (a) Graph. (b) State-space tree for finding a Hamiltonian circuit. The numbers above the nodes of the tree indicate the order in which the nodes are generated.

# Subset-sum problem (1)

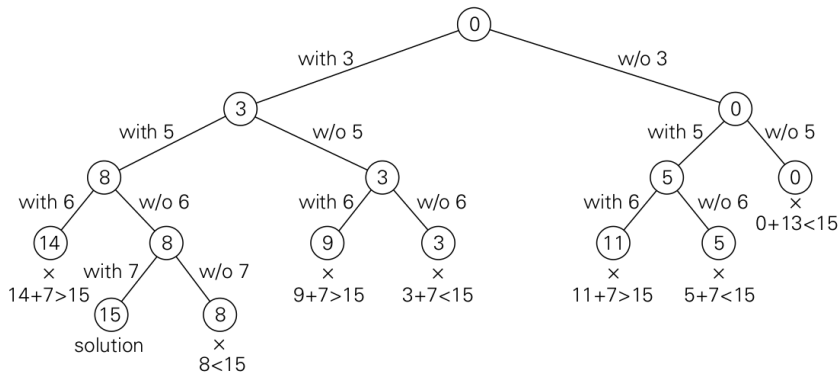
## Problem

*Find a subset of a given set  $A = \{a_1, \dots, a_n\}$  of  $n$  positive integers whose sum is equal to a given positive integer  $d$ .*

**Example 1:** Given  $A = \{1, 2, 5, 6, 8\}$ ,  $d = 9$ , the solution are:  $\{1, 2, 6\}$  and  $\{1, 8\}$ .

**Example 2:** Given  $A = \{3, 5, 6, 7\}$ ,  $d = 15$ , the solution are:  $\{3, 5, 7\}$ .

# Subset-sum problem (2)



**Figure:** Complete state-space tree of the backtracking algorithm applied to the instance  $A = \{3, 5, 6, 7\}$  and  $d = 15$  of the Subset-sum problem. The number inside a node is the sum of the elements already included in the subsets represented by the node. The inequality below a leaf indicates the reason for its termination.

# Subset-sum problem (1)

A path from the root to a node on the  $i$ th level of the tree indicates which of the first  $i$  numbers have been included in the subsets represented by that node.

We record the value of  $s$ , the sum of these numbers, in the node.

- If  $s$  is equal to  $d$ , we have a solution to the problem. We can either report this result and stop or,
- If all solutions need to be found, continue by backtracking to the nodes parent.
- If  $s$  is not equal to  $d$ , we can terminate the node as non-promising if either of the following two inequalities holds:

$$s + a_{i+1} > d \text{ (the sum } s \text{ is too large)}$$

$$s + \sum_{j=i+1}^n a_j < d \text{ (the sum } s \text{ is too small)}$$

# Backtracking framework

---

## Algorithm 1 Backtracking

---

```
1: procedure BACKTRACK( $X[1..i]$ )
2:   input:  $X[1..i]$ : the first  $i$  promising components of a solution
3:   output: all the tuples representing the problem's solution
4:   if  $X[1..i]$  is a solution then
5:     write ( $X[1..i]$ )
6:   else
7:     for each  $x \in S_{i+1}$  consistent with  $X[1..i]$  and the constraints do
8:        $X[i + 1] \leftarrow x$ 
9:       BACKTRACK( $X[1..i + 1]$ )
10:    end for
11:  end if
12: end procedure
```

---



Backtracking is basically an exhaustive search performed over the search space. So the time complexity of a backtracking algorithm is **defined by the size of the search space**.

For example, in the  $n$ -queens problem and Hamiltonian problem, the size of the search space is about  $\mathcal{O}(n!)$ .

*Intuitively*, the first queen has  $n$  placements, the second queen must not be in the same column as the first as well as at an oblique angle, so the second queen has  $n - 1$  possibilities, and so on, with a time complexity of  $\mathcal{O}(n!)$ .

# Advantages & drawbacks

## Advantages

- Typically applied to difficult combinatorial problems for which no efficient algorithms for finding exact solutions possibly exist.
- Unlike the exhaustive-search approach, backtracking at least holds a hope for solving some instances of nontrivial sizes in an acceptable amount of time (especially for optimization problems).
- Even if backtracking does not eliminate any elements of a problems state space and ends up generating all its elements, it provides a specific technique for doing so.

## Drawbacks

- Backtracking is **not** a very efficient technique (even though it was succeeded to use in the previous problems).
- In the worst case, it may have to generate all possible candidates in an exponentially (or faster) growing state space of the problem.