CSIT113
Problem Solving

UNIT 9
COPING WITH COMPLEXITY USING BACKTRACKING

UNIVERSITY OF WOLLONGONG AUSTRALIA

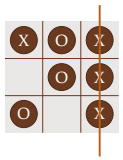
SIM GLOBAL EDUCATION

Overview

- Introducing State-Space Tree and State-Space Graph
- Investigating Brute-Force Complexity (Time) using State-Space Tree
- Backtracking - a means to cope with time complexity, more specifically, it helps to design combinatorial algorithms that find required solutions faster.

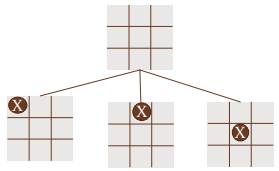
State-Space Tree and State-Space Graph for Games

- Consider the game of Tic-Tac-Toe (Noughts and Crosses)
  - Two players X and O alternate play on a 3 × 3 grid.
  - Each player puts their symbol in one of the empty squares.
  - The winner is the first player to establish a line of three of their symbol.
    - Horizontal
    - Vertical
    - Diagonal
  - A draw is possible
  - Consider the following sample game:
    - ...In this case, X wins!

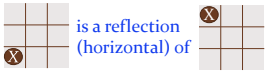


State-Space Tree and State-Space Graph for Game

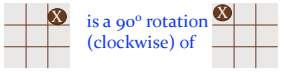
- We can construct a tree showing all possible positions in the game.
- The root is an empty board.
- The next level shows all possible first moves



- Note that there are only three possible first moves as:
  - The rest are reflections or rotations of these: example of reflections or rotations:



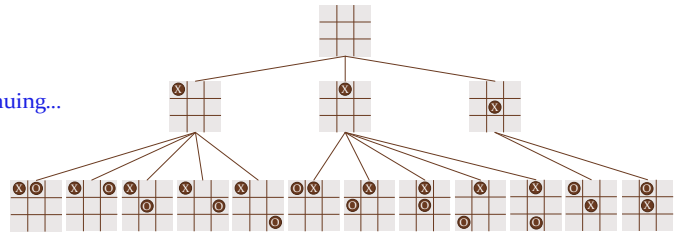
is a reflection (horizontal) of



is a 90° rotation (clockwise) of

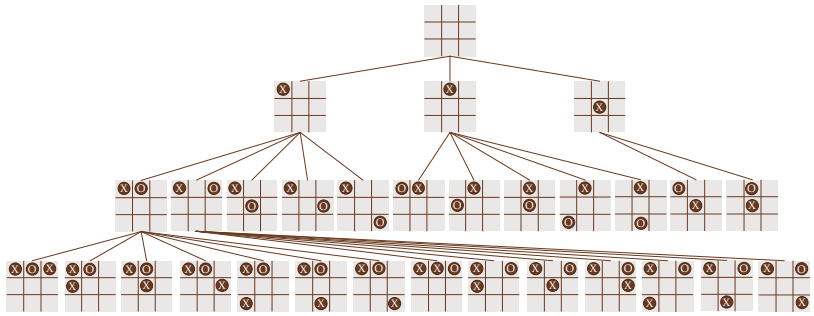
State-Space Tree and State-Space Graph for Game

- Continuing…



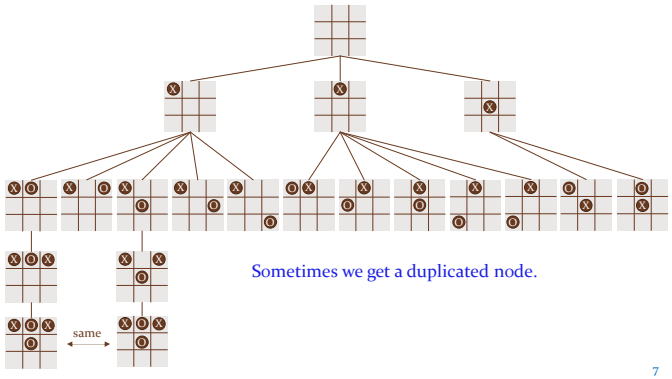
- Excluding reflections or rotations of these
- And so on…

State-Space Tree and State-Space Graph for Game



As you can see this is going to be a pretty big tree!

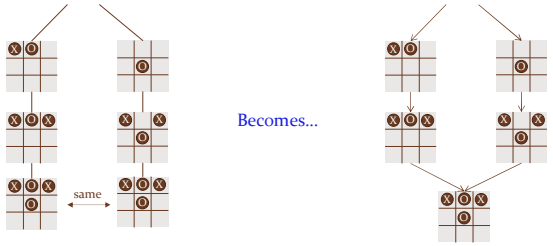
State-Space Tree and State-Space Graph for Game



Sometimes we get a duplicated node.

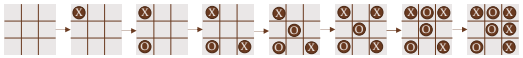
State-Space Tree and State-Space Graph for Game

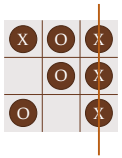
- We can eliminate duplicated nodes
- This turns our state-space tree into a state-space graph
- This state-space graph is directed and acyclic



State-Space Tree and State-Space Graph for Game

- The terminal nodes (leaves) of the state-space graph correspond to winning/losing or drawn positions.
- Thus, our sample game is one path from the root to a leaf...





- ...is one possible complete game.

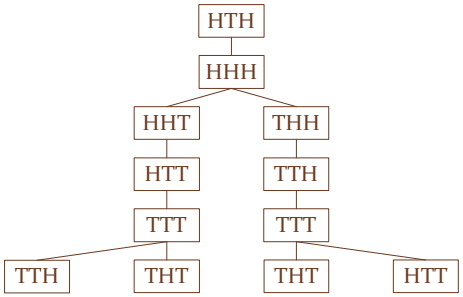
A Solitaire Game.

- Let us now look at a game with a much simpler state-space graph.
- We start with three coins...

- We can flip the centre coin at any time.
- We can flip either end coin only if the other two coins show the same face.
- Starting Position is HTH
- Ending Position is THT

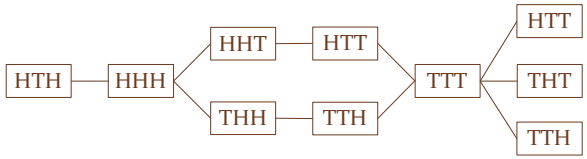
A Solitaire Game.

- We can construct the state-space tree by starting at the initial position and trying all valid moves:



A Solitaire Game.

- We can simplify this by combining duplicated nodes to get the equivalent state-space graph.



- Any path from the start node to the end node is a solution to the game.

state-space trees and state-space graphs

- We can create a tree or graph for any problem – not just for games.
- Each node represents a partial attempt at a solution.
- The leaves represent final states for the problem – these may be:
  - Solutions
  - or Dead ends.
- The difficulty is to find a path from the root to a solution – without building the whole tree.
- Any solution strategy can be viewed as a way of choosing the next node in the tree.

State-Space Trees using Brute-Force Approach

- Consider the **discrete** backpack problem:
  - We have a set of objects, each with a weight and a value.
  - We need to assemble the greatest possible value into our backpack.
  - We must put all of an object in the backpack.
- We could construct the state-space tree in two different ways:
  1. At each level, add one object at a time to the pack.
  2. At each level, include or exclude one object at a time in the pack.

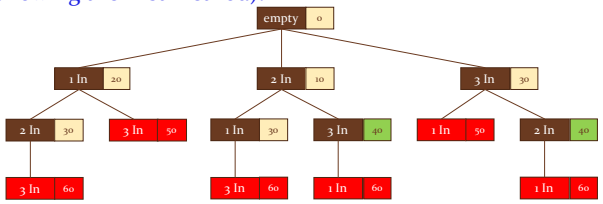
State-Space Trees by Adding One Object at a time

- Consider the following problem:

Object	1	2	3
Value	20	10	30
Weight	5	2	3

Backpack capacity = 7

- Tree 1 (following the first method):



- Red nodes are illegal. Green nodes are optimal.

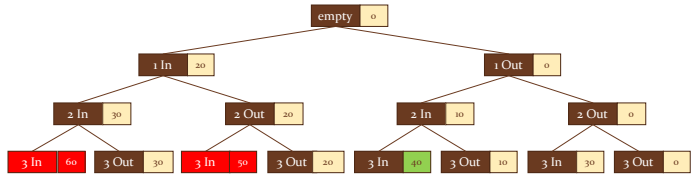
State-Space Trees by Including or Excluding One Object at a time

- Consider the following problem:

Object	1	2	3
Value	20	10	30
Weight	5	2	3

Backpack capacity = 7

- Tree 2 (following the second method):



- Again Red is illegal, green is best.

The Problem in using Brute Force

- The big problem we have here is that we have no easy way to find the best terminal node.
- We need some way of evaluating the nodes as we go.
- So, we construct the whole tree and then find the solution somewhere within it.
- If we have 10 objects we have a tree with over 1000 (2^{10}) leaves!
- Each extra object doubles the number.
- Do not even think about trying to fit the best selection from 100 objects!

How do we fix this.

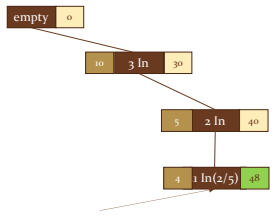
- We need some way to decide which node is the most promising at each level.
- We call this measure of "promisingness" a *metric*.
- We could then choose this node to further expand.
- This is the basis of the greedy strategy.
- Let's look at solving the **continuous** backpack problem in terms of a state-space tree.
- We will start with the same set of objects and use value/weight as our metric.

State-Space Tree of continuous backpack problem using Greedy Strategy

- Consider the following problem:

Object	1	2	3
Value	20	10	30
Weight	5	2	3
V/W	4	5	10

Backpack capacity = 7



Note the partial amounts of object 1:

- Clearly, this is a much smaller state-space tree.
- But, Greedy approach may not be able to find an optimal solution for some problems.

When Greedy Fails

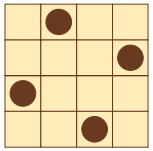
- As we have seen in previous weeks the greedy strategy does not always lead to a solution.
- Sometimes the promising node turns out to be wrong.
- What can we do in this case?
  - Brute-force?
  - Something else?
- Brute-force always works but often we can find a better strategy.
- This unit will examine one such strategy, ***backtracking***.

The N-Queens Problem

- This problem involves placing n chess queens on a square $n \times n$ board so that no two queens share:
  - the same row;
  - the same column;
  - the same diagonal.

The N-Queens Problem

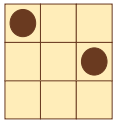
- Here is a sample 4 x 4 solution:



We Have a Problem

- One difficulty with this problem is that there is no solution for some values of n.
  - $n = 2$

  - $n = 3$

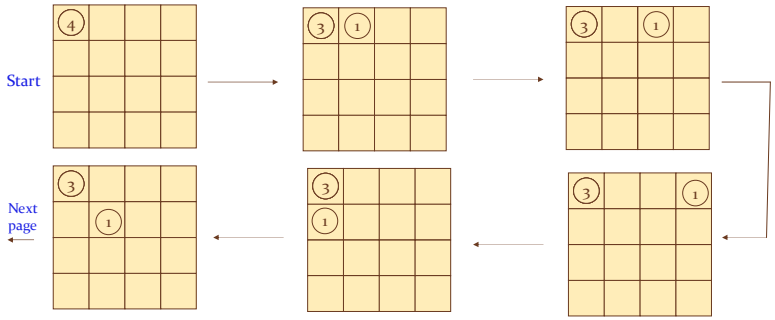


Brute-Force 1

- The simplest statement of the problem (the one that uses least information) is as follows:
  - Place *N-Queens* on a square board with n^2 squares so that no two queens threaten each other.
- If we number the squares from 1 to n^2 we can try all possible values for each queen.

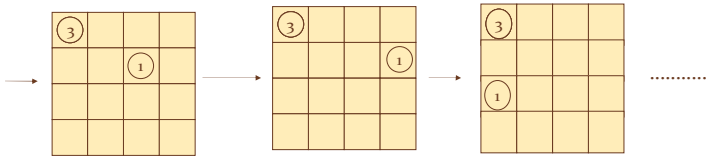
Solving 4-Queens Problem using Brute-Force 1

- Thus all queens start on square 1 and we try moving each queen to all possible squares independently.
- For a 4 x 4 board this approach can start and proceed as follows (each circle indicates the number of queens on the location): all fail



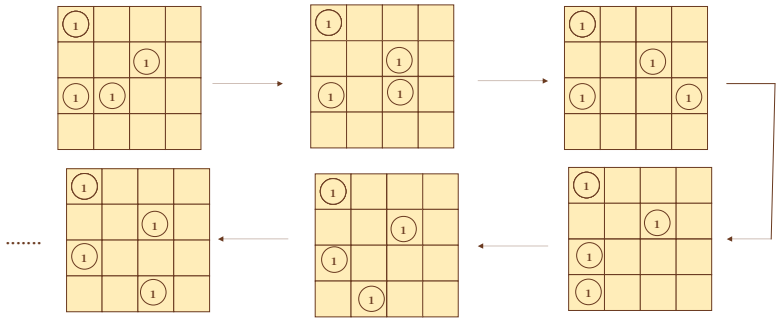
Solving 4-Queens Problem using Brute-Force 1

- Continued from previous page:



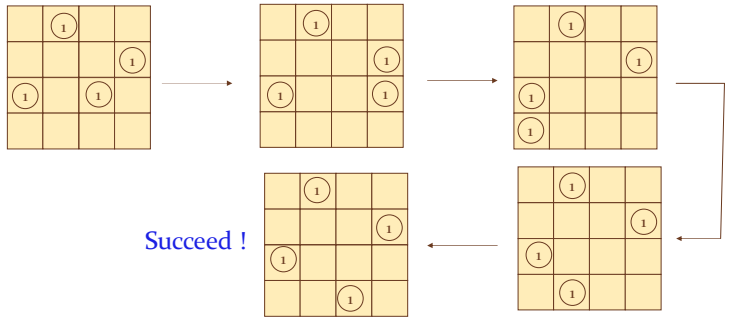
Solving 4-Queens Problem using Brute-Force 1

- ... much later... : again, all fail



Solving 4-Queens Problem using Brute-Force 1

- ... much, much later... : we get a solution finally.



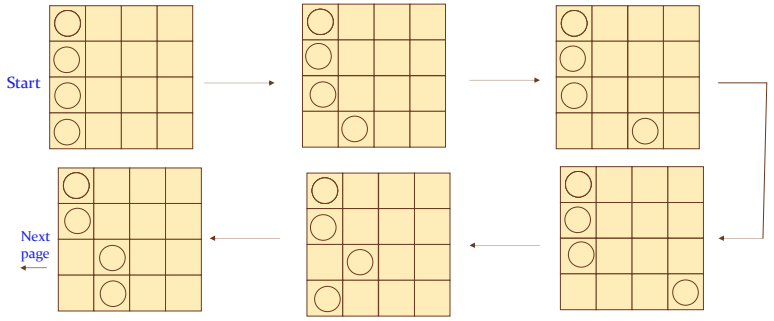
- Altogether, there are $(4 \times 4)^4$ possible cases

What Next?

- This approach is clearly stupid!
- We are using none of the information inherent in the problem.
- For example, each queen must be in a different row.
- Let's use this (Brute-Force 2) to construct a less brutish brute-force solution

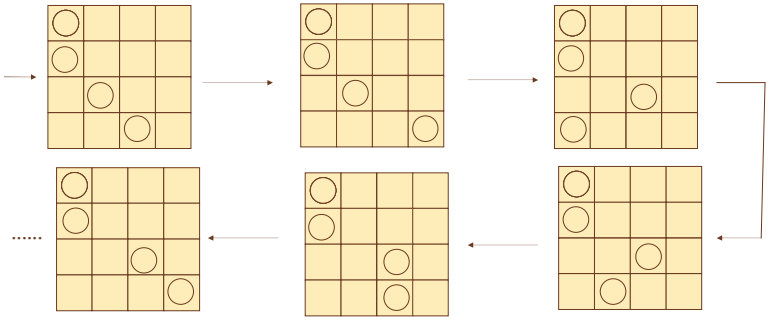
Solving 4-Queens Problem using Brute-Force 2

- Brute-Force 2 can start and proceed as follows: all fail



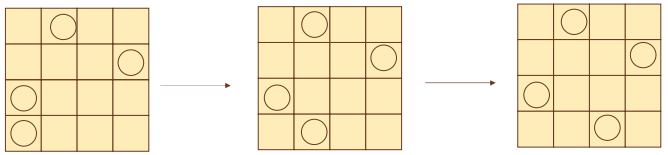
Solving 4-Queens Problem using Brute-Force 2

- Continue from previous page: again, all fail



Solving 4-Queens Problem using Brute-Force 2

- ... sometime later... :we get a solution finally.



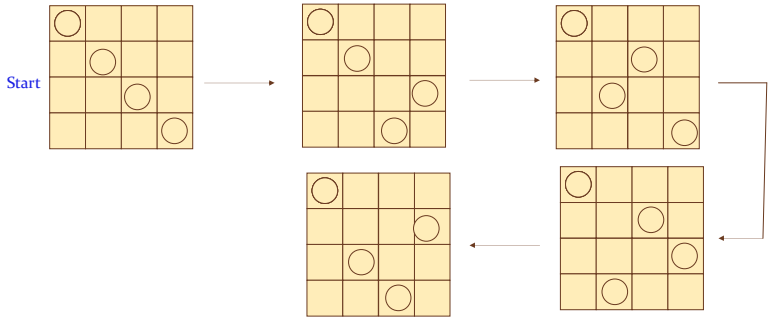
- Altogether, there are 4^4 possible cases

What Next?

- This is better but we still are doing a lot of needless work.
- Let us add the fact that each queen is in a different column.
- We can use this to build brute-force 3.

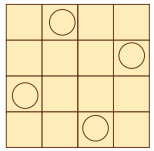
Solving 4-Queens Problem using Brute-Force 3

- Brute-Force 3 can start and proceed as follows: all fail



Solving 4-Queens Problem using Brute-Force 3

- ... sometime later... :we get a solution finally.



Succeed !

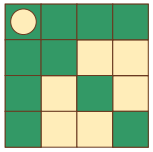
- Altogether, there are 4×3×2×1 (= 4!) possible cases

What Next?

- This is even better but we still are doing a lot of needless work.
- Perhaps our basic strategy, placing all the queens at once, is at fault.
- What if we try placing them one at a time? This method is called **backtracking**.

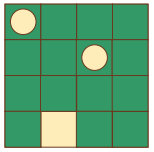
Solving 4-Queens Problem using Backtracking

- Place a queen in row 1



Solving 4-Queens Problem using Backtracking

- Place a queen in row 2



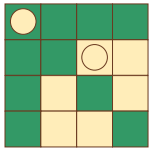
- We cannot place the next queen!
- Now what do we do?

Backtracking

- When we placed each queen we took the first available square.
- But there were other squares available.
- If the square taken does not work, what we have to do is to go back to the last place we had another move available and try it instead.
- This procedure is known as backtracking.

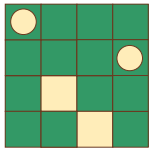
Solving 4-Queens Problem using Backtracking

- This didn't work as we cannot place the next queen.



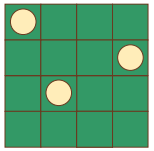
Solving 4-Queens Problem using Backtracking

- So try this



Solving 4-Queens Problem using Backtracking

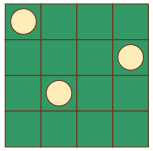
- Place a queen in row 3



- Again, we cannot continue.

Solving 4-Queens Problem using Backtracking

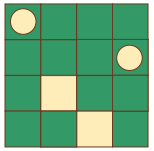
- We cannot place a queen in row 4!



- Backtrack

Solving 4-Queens Problem using Backtracking

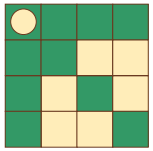
- And we have run out of possibilities on row 2



- Backtrack again!

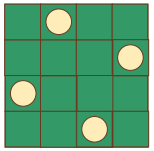
Solving 4-Queens Problem using Backtracking

- This didn't work.



Solving 4-Queens Problem using Backtracking

- So try this.



- Succeed!

Efficiency (based on what we have tried)

- Let us compare the different strategies (empirical data):

Strategy	Number of Moves
Brute-Force 1	6031
Brute-Force 2	115
Brute-Force 3	11
Backtracking	8 (partial)

- As *n* is increased, the value of backtracking becomes even greater.

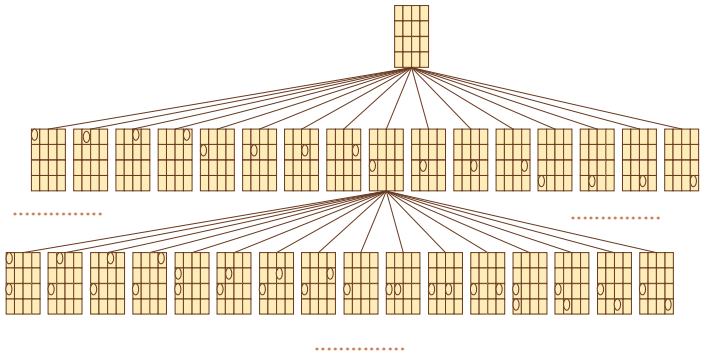
state-space trees

- We can understand what is happening for each method by analysing the underlying data structure that describes the problem.
- The ***state-space tree (game-tree)***.
- This is a tree constructed by showing each move in turn and linking it to each possible continuation.

State-Space Trees of First Brute-Force

- In the first brute-force algorithm, each node has 16 children.
- As the tree has 4 levels, there are 65,536 leaves in the tree, each corresponding to one possible arrangement of queens.
- The algorithm is to look at each leaf in turn until we find the solution.
- With *N-Queens* this gives n^{2n} leaves.

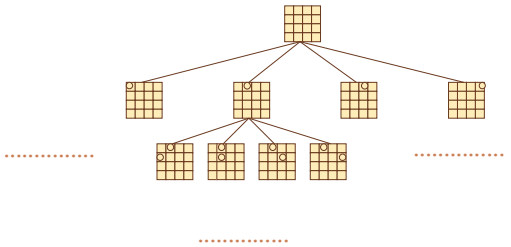
State-Space Trees of First Brute-Force



State-Space Trees of Second Brute-Force

- In the second brute-force algorithm, each node has 4 children.
- As the tree has 4 levels, there are 256 leaves in the tree, each corresponding to one possible arrangement of queens.
- This reduction in the size of the tree is what makes this much faster to solve.
- With *N-Queens* this gives n^n leaves.

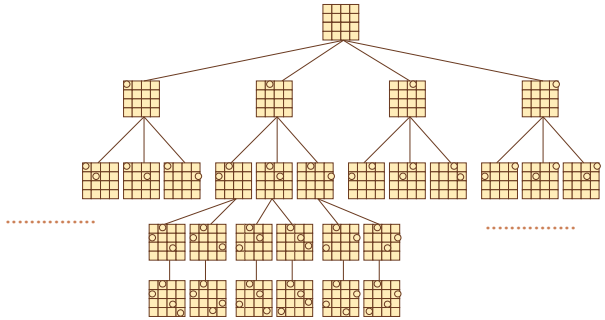
State-Space Trees of Second Brute-Force



State-Space Trees of Third Brute-Force

- In the third brute-force algorithm, the root node has 4 children.
- Each of these has 3 children.
- Each of these has 2.
- And each of these has 1.
- There are a total of 24 leaves.
- We examine each of these in turn.
- With *N-Queens* this gives *n*! leaves.

State-Space Trees of Third Brute-Force

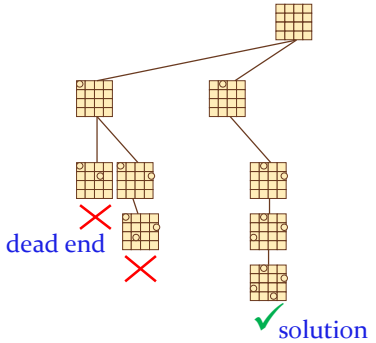


State-Space Tree of Backtracking

- The backtracking algorithm takes a different approach.
- Instead of looking only at the leaves, look at the internal nodes as we construct them.
- Stop building the sub-tree as soon as you get an illegal position.
- This dramatically reduces the work we have to do.

Solving 4-QueensProblem using Backtracking

State-Space Tree



State-Space Trees

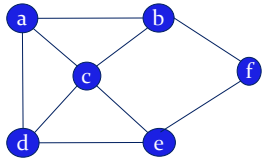
- As *n* gets bigger, the advantage becomes greater.

No	*First Brute-Force*	*First Brute-Force*	*Third Brute-Force*	Backtracking
n	n^{2n}	n^n	*n*!	Not available
4	65,536	256	24	8
5	9,765,625	3,125	120	5
6	2,176,782,336	46,656	720	34
7	678,223,072,849	823,543	5,040	10
8	281,474,976,710,656	16,777,216	40,320	?

- The numbers on backtracking are based on the strategy used (row-by-row).

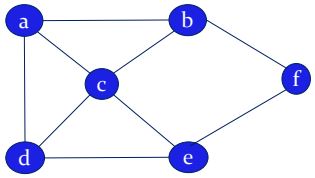
Solving Hamiltonian Circuit Problem using Backtracking

- A **Hamiltonian Circuit** in a graph is defined as a cycle that visits all the graph's vertices exactly once before returning to the starting vertex.
- Backtracking can be used to find **Hamiltonian Circuits** in connect graphs.
- Next, we shall apply backtracking to find a Hamiltonian circuit in the following connected graph starting from vertex a:

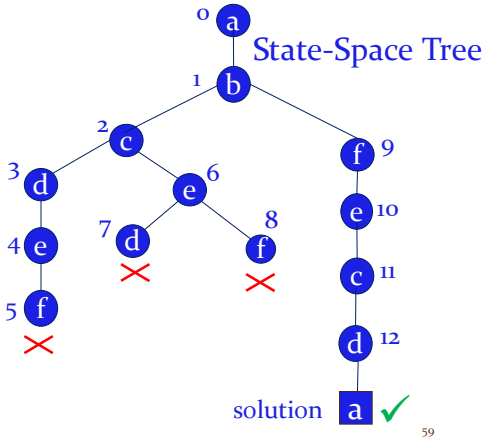


Solving Hamiltonian Circuit Problem using Backtracking

Start vertex :a
Strategy used: according to alphabetical order if there are multiple choices

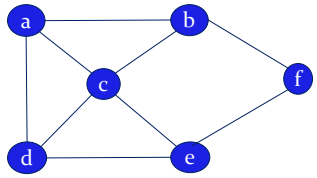


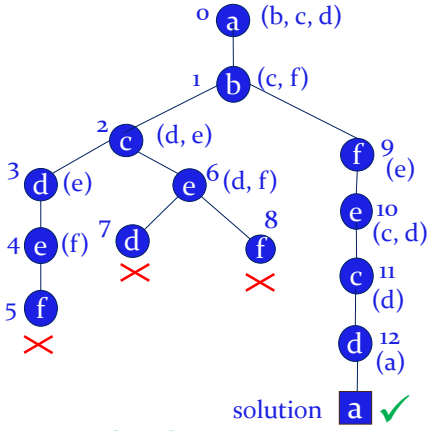
State-Space Tree



Solving Hamiltonian Circuit Problem using Backtracking

Start vertex :a
Strategy used: according to alphabetical order if there are multiple choices





State-Space Tree

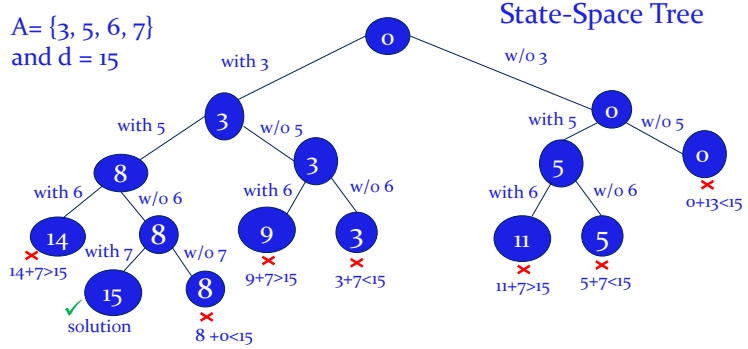
Solving Subset-Sum Problem using Backtracking

- **Subset-sum problem:** find a subset of a given set A = {a_1,...., a_n} of n positive integers whose sum is equal to a given positive integer d. For example, for A = {1, 2, 5, 6, 8} and d = 9, there are two solutions: {1, 2, 6} and {1, 8}. Some instances of this problem may have no solutions.
- Subset-sum problem can be solved using backtracking. Usually, we sort the set's elements in increasing order first before applying backtracking. Hence, $a_1 \le a_2 \le .... \le a_n$, It proceeds as follows:
  - ✓We record the value s, the sum of all the numbers included in the node.
  - ✓If s = d, we have a solution. We can either: (1) if we need to find only one solution, we report the result and stop; (2) if we need to find all solutions, we continue by backtracking to the node's parent.
  - ✓If s ≠ d, we terminate the node as nonpromising if one of following conditions holds:
    - $s + a_{i+1} > d$
    - $s + \sum_{j=i+1}^{n} a_j < d$
- Next, we shall apply backtracking for A = {3, 5, 6, 7} and d = 15, there is only one solution: {3, 5, 7}.

Solving Subset-Sum Problem using Backtracking

A= {3, 5, 6, 7}
and d = 15

State-Space Tree



- The number inside a node is the sum of the elements already included in the subset represented by the node.
- The inequality below a leaf indicates the reason for its termination.

Backtracking

- This approach of constructing a tree of solutions and pruning it as it is built can be applied to a wide range of problems. Usually, it is used for non-optimization problems as other better techniques are available for optimization problems.
- Even so, we eventually reach a point where this approach is still too time consuming.
  - Consider solving the 1,000 queens problem.
- At this point we need to introduce ways of pruning the state-space tree even more effectively.

Metrics

- In the *n*-queens problem we explored the state-space tree in order.
  - We considered possible positions, left to right, in each row.
- What if we could attach a value to each node that provided some estimate of how likely its branch was to contain the solution?
- This would provide a better order in which to select branches to explore.
  - Select the branch with highest value.
- We call such a measure a *metric*.
- Adding a metric can dramatically improve backtracking.

Metrics and Greedy Strategies

- Every greedy algorithm is based on a metric.
- For greedy to work we need the metric to be perfect.
  - It must guarantee that the solution is down the chosen path.
  - The likelihood is either zero or one.
- In other cases we have a metric that is not perfect.
  - It does not guarantee that the solution is down the chosen path.
  - The likelihood is between zero and one.
- In such cases we cannot use a greedy strategy with guaranteed success.
- We can, however, use backtracking guided by the metric.

Heuristics

- When a metric is not perfect we call it a *heuristic*.
- Heuristics can improve the performance of a number of solution strategies.
- A good heuristic can dramatically improve the performance of these strategies.
- Good:
  - Close match to reality (the metric leads quickly to the solution)
  - Easy/cheap to evaluate
- Sometimes backtracking, even with a good heuristic, is not enough.
- We need even better strategies.