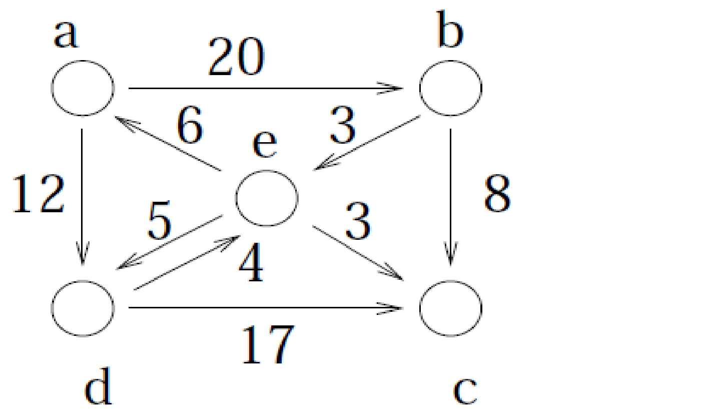
Dynamic Programming
All Pair Shortest Path

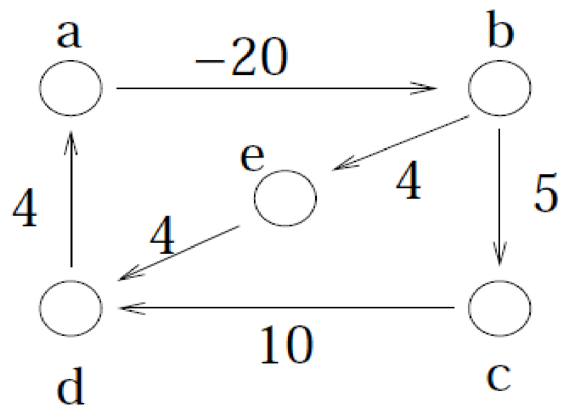
Manoj Kumar
DTU, Delhi

The All-Pairs Shortest Paths Problem

Given a weighted digraph $G = (V, E)$ with weight function $w : E \rightarrow \mathbf{R}$, ($R$ is the set of real numbers), determine the length of the shortest path (i.e., distance) between all pairs of vertices in G. Here we assume that there are no cycles with zero or negative cost.



without negative cost cycle



with negative cost cycle

Solution 1: Using Dijkstra's Algorithm

If there are no negative cost edges apply Dijkstra's algorithm to each vertex (as the source) of the digraph.

- Recall that D's algorithm runs in $\Theta((n+e)\log n)$. This gives a

  $$\Theta(n(n+e)\log n) = \Theta(n^2 \log n + ne \log n)$$

  time algorithm, where $n = |V|$ and $e = |E|$.

- If the digraph is dense, this is an $\Theta(n^3 \log n)$ algorithm.

- With more advanced (complicated) data structures D's algorithm runs in $\Theta(n \log n + e)$ time yielding a $\Theta(n^2 \log n + ne)$ final algorithm. For dense graphs this is $\Theta(n^3)$ time.

Solution 2: Dynamic Programming

1. How do we decompose the all-pairs shortest paths problem into sub problems?
2. How do we express the optimal solution of a sub problem in terms of optimal solutions to some sub problems?
3. How do we use the recursive relation from (2) to compute the optimal solution in a bottom-up fashion?
4. How do we construct all the shortest paths?

Solution2:Input and Output Formats

To simplify the notation, we assume that $V = \{1, 2, \ldots, n\}$.

Assume that the graph is represented by an $n \times n$ matrix with the weights of the edges:

$$w_{ij} = \begin{cases} 0 & \text{if } i = j, \\ w(i,j) & \text{if } i \neq j \text{ and } (i,j) \in E, \\ \infty & \text{if } i \neq j \text{ and } (i,j) \notin E. \end{cases}$$

Output Format: an $n \times n$ matrix $D = [d_{ij}]$ where d_{ij} is the length of the shortest path from vertex i to j.

Step 1: How to Decompose the Original Problem

- Subproblems with smaller sizes should be easier to solve.
- An optimal solution to a subproblem should be expressed in terms of the optimal solutions to subproblems with smaller sizes

These are guidelines ONLY

Step 1: Decompose in a Natural Way

- Define $d_{ij}^{(m)}$ to be the length of the shortest path from i to j that contains at most m edges.
  Let $D^{(m)}$ be the $n \times n$ matrix $[d_{ij}^{(m)}]$.

- $d_{ij}^{(n-1)}$ is the true distance from i to j (see next page for a proof this conclusion).

- Subproblems: compute $D^{(m)}$ for $m = 1, \cdots, n - 1$.

Question: Which $D^{(m)}$ is easiest to compute?

$d_{ij}^{(n-1)}$ = True Distance from i to j

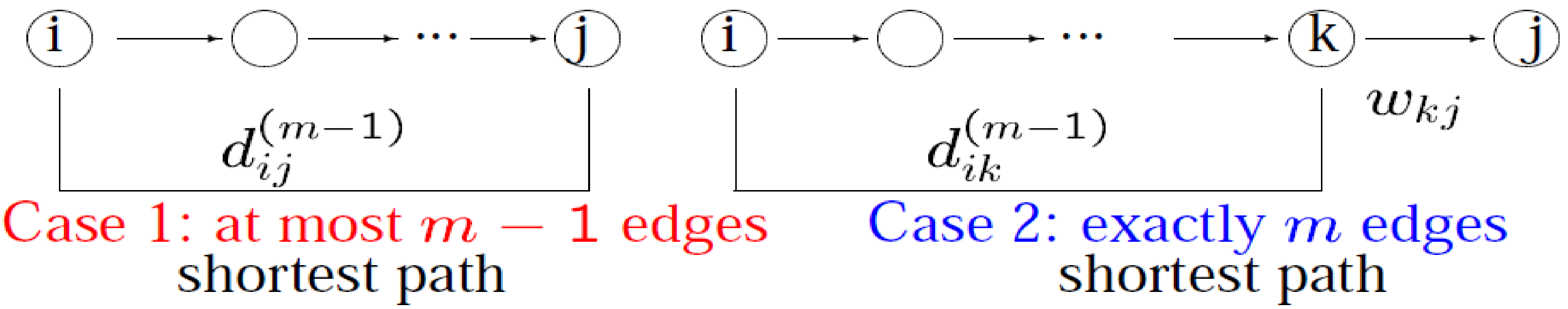
Proof: We prove that any shortest path P from i to j contains at most $n-1$ edges.

First note that since all cycles have positive weight, a shortest path can have no cycles (if there were a cycle, we could remove it and lower the length of the path).

A path without cycles can have length at most $n-1$ (since a longer path must contain some vertex twice, that is, contain a cycle).

A Recursive Formula

Consider a shortest path from i to j of length $d_{ij}^{(m)}$.



Case 1: It has at most $m-1$ edges.
Then $d_{ij}^{(m)} = d_{ij}^{(m-1)} = d_{ij}^{(m-1)}$
Case 2: It has m edges. Let k be the vertex before j on a shortest path.
Then $d_{ij}^{(m)} = d_{ik}^{(m-1)} + w_{kj}$.

A Recursive Formula

Combining the two cases,

$$d_{ij}^{(m)} = \min_{1 \le k \le n} \left\{ d_{ik}^{(m-1)} + w_{kj} \right\} .$$

Step 3: Bottom-up Computation of $D^{(n-1)}$

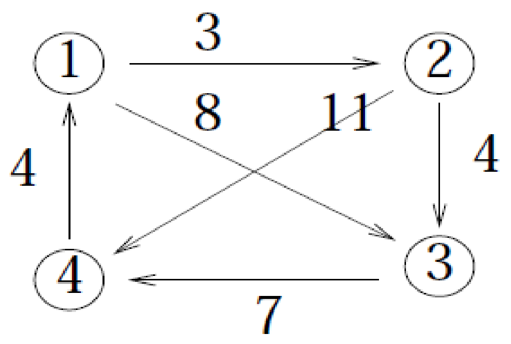
- Bottom: $D^{(1)} = [w_{ij}]$, the weight matrix.

- Compute $D^{(m)}$ from $D^{(m-1)}$, for $m = 2, ..., n-1$, using

$$d_{ij}^{(m)} = \min_{1\le k\le n} \left\{ d_{ik}^{(m-1)} + w_{kj} \right\}.$$

Example: Bottom-up Computation of $D^{(n-1)}$

Example

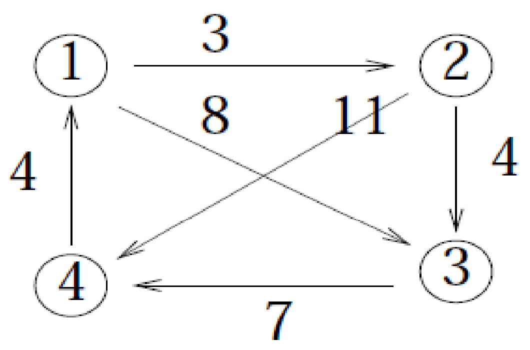


$D^{(1)} = [w_{ij}]$ is just the weight matrix:

$$D^{(1)} = \begin{bmatrix} 0 & 3 & 8 & \infty \\ \infty & 0 & 4 & 11 \\ \infty & \infty & 0 & 7 \\ 4 & \infty & \infty & 0 \end{bmatrix}$$

Example: Computing $D^{(2)}$ from $D^{(1)}$

$$d_{ij}^{(2)} = \min_{1 \le k \le 4} \left\{ d_{ik}^{(1)} + w_{kj} \right\}.$$



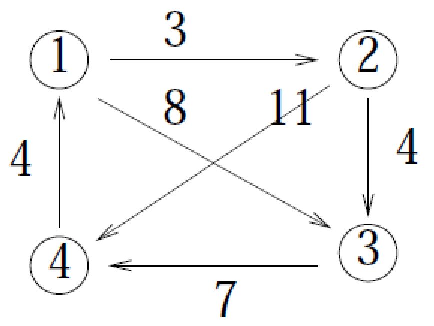
$$D^{(1)} = \begin{bmatrix} 0 & 3 & 8 & \infty \\ \infty & 0 & 4 & 11 \\ \infty & \infty & 0 & 7 \\ 4 & \infty & \infty & 0 \end{bmatrix}$$

With $D^{(1)}$ given earlier and the recursive formula,

$$D^{(2)} = \begin{bmatrix} 0 & 3 & 7 & 14 \\ 15 & 0 & 4 & 11 \\ 11 & \infty & 0 & 7 \\ 4 & 7 & 12 & 0 \end{bmatrix}$$

Example: Computing $D^{(3)}$ from $D^{(2)}$

$$d_{ij}^{(3)} = \min_{1 \le k \le 4} \left\{ d_{ik}^{(2)} + w_{kj} \right\}$$



$$D^{(2)} = \begin{bmatrix} 0 & 3 & 7 & 14 \\ 15 & 0 & 4 & 11 \\ 11 & \infty & 0 & 7 \\ 4 & 7 & 12 & 0 \end{bmatrix}$$

With $D^{(2)}$ given earlier and the recursive formula,

$$D^{(3)} = \begin{bmatrix} 0 & 3 & 7 & 14 \\ 15 & 0 & 4 & 11 \\ 11 & 14 & 0 & 7 \\ 4 & 7 & 11 & 0 \end{bmatrix}$$

$D^{(3)}$ gives the distances between any pair of vertices.

The Algorithm for Computing $D^{(n-1)}$

```
for m ← 1 to n-1
    do  for i ← 1 to n
            do  for j ← 1 to n
                    do  min ← ∞
                        for k ← 1 to n
                            do new ← $d_{ik}^{(m-1)}$ + $w_{kj}$
                                if (new < min)
                                    then  min ← new
                        $d_{ij}^{(m)}$ ← min
```

Comments on Solution 2

- Algorithm uses $\Theta(n^3)$ space; how can this be reduced down to $\Theta(n^2)$?
- How can we extract the actual shortest paths from the solution?
- Running time $O(n^4)$, much worse than the solution using Dijkstra's algorithm. Can we improve this?

Repeated Squaring

Observe that we are only interested to find $D^{(n-1)}$, all others D^i, $1 \leq i \leq n-2$ are only auxiliary. Furthermore, since the graph does not have negative cycle, we have $D^{(n-1)} = D^i$, for all $i \geq n$.

In particular, this implies that $D^{\left(2^{\lceil \log_2 n \rceil}\right)} = D^{(n-1)}$.

We can calculate $D^{\left(2^{\lceil \log_2 n \rceil}\right)}$ using "repeated squaring" to find

$$D^{(2)}, D^{(4)}, D^{(8)}, \ldots, D^{\left(2^{\lceil \log_2 n \rceil}\right)}$$

Repeated Squaring

We use the recurrence relation:

- Bottom: $D^{(1)} = [w_{ij}]$, the weight matrix.
- For $s \geq 1$ compute $D^{(2s)}$ using

$$d_{ij}^{(2s)} = \min_{1 \leq k \leq n} \left\{ d_{ik}^{(s)} + d_{kj}^{(s)} \right\}.$$

Given this relation we can calculate $D^{(2^i)}$ from $D^{(2^{i-1})}$ in $O(n^3)$ time. We can therefore calculate all of

$$D^{(2)}, D^{(4)}, D^{(8)}, \ldots, D^{\left(2^{\lceil \log_2 n \rceil}\right)} = D^{(n)}$$

in $O(n^3 \log n)$ time, improving our running time.

The Floyd-Warshall Algorithm

Step 1 : Decomposition

Definition: The vertices $v_2, v_3, ..., v_{l-1}$ are called the *intermediate vertices* of the path $p = \langle v_1, v_2, ..., v_{l-1}, v_l \rangle$.

- Let $d_{ij}^{(k)}$ be the length of the shortest path from i to j such that *all* intermediate vertices on the path (if any) are in set $\{1, 2, \ldots, k\}$.

  $d_{ij}^{(0)}$ is set to be w_{ij}, i.e., no intermediate vertex.

  Let $D^{(k)}$ be the $n \times n$ matrix $[d_{ij}^{(k)}]$.

- Claim: $d_{ij}^{(n)}$ is the distance from i to j. So our aim is to compute $D^{(n)}$.

- Subproblems: compute $D^{(k)}$ for $k = 0, 1, \cdots, n$.

Step2: Structure of shortest path

Observation 1: A shortest path does not contain the same vertex twice. Proof: A path containing the same vertex twice contains a cycle. Removing cycle gives a shorter path.

Step2: Structure of shortest path

Observation 2: For a shortest path from i to j such that any intermediate vertices on the path are chosen from the set $\{1, 2, \ldots, k\}$, there are two possibilities:

1. k is not a vertex on the path,
The shortest such path has length $d_{ij}^{(k-1)}$.

2. k is a vertex on the path.
The shortest such path has length $d_{ik}^{(k-1)} + d_{kj}^{(k-1)}$.

Step2: Structure of shortest path

Consider a shortest path from i to j containing the vertex k. It consists of a subpath from i to k and a subpath from k to j.
Each subpath can only contain intermediate vertices in $\{1, \ldots, k-1\}$, and must be as short as possible, namely they have lengths $d_{ik}^{(k-1)}$ and $d_{kj}^{(k-1)}$.

Hence the path has length $d_{ik}^{(k-1)} + d_{kj}^{(k-1)}$.

Combining the two cases we get

$$d_{ij}^{(k)} = \min\left\{d_{ij}^{(k-1)},\, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right\}.$$

Step 3: the Bottom-up Computation

- Bottom: $D^{(0)} = [w_{ij}]$, the weight matrix.

- Compute $D^{(k)}$ from $D^{(k-1)}$ using

$$d_{ij}^{(k)} = \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right)$$

for $k = 1, ..., n.$

The Floyd-Warshall Algorithm: Version 1

```
Floyd-Warshall($w, n$)
{ for $i = 1$ to $n$ do                                   initialize
    for $j = 1$ to $n$ do
    { $D^0[i,j] = w[i,j]$;
      $pred[i,j] = nil$;
    }

  for $k = 1$ to $n$ do                                   dynamic programming
    for $i = 1$ to $n$ do
      for $j = 1$ to $n$ do
        if $\left(d^{(k-1)}[i,k] + d^{(k-1)}[k,j] < d^{(k)}[i,j]\right)$
            $\{d^{(k)}[i,j] = d^{(k-1)}[i,k] + d^{(k-1)}[k,j]$;
            $pred[i,j] = k;\}$
        else $d^{(k)}[i,j] = d^{(k-1)}[i,j]$;
  return $d^{(n)}[1..n, 1..n]$;
}
```

Comments on the Floyd-Warshall Algorithm

- The algorithm's running time is clearly $\Theta(n^3)$.
- The predecessor pointer `pred[i, j]` can be used to extract the final path (see later).
- Problem: the algorithm uses $\Theta(n^3)$ space.
  It is possible to reduce this down to $\Theta(n^2)$ space by keeping only one matrix instead of n.
  Algorithm is on next page. Convince yourself that it works.

The Floyd-Warshall Algorithm: Version 2

```
Floyd-Warshall($w, n$)
{ for $i = 1$ to $n$ do                    initialize
    for $j = 1$ to $n$ do
    { $d[i, j] = w[i, j];$
      $pred[i, j] = nil;$
    }

  for $k = 1$ to $n$ do                    dynamic programming
    for $i = 1$ to $n$ do
      for $j = 1$ to $n$ do
        if $(d[i, k] + d[k, j] < d[i, j])$
            $\{d[i, j] = d[i, k] + d[k, j];$
             $pred[i, j] = k;\}$
  return $d[1..n, 1..n];$
}
```

Extracting the Shortest Paths

The predecessor pointers `pred[i, j]` can be used to extract the final path. The idea is as follows.

Whenever we discover that the shortest path from i to j passes through an intermediate vertex k, we set $pred[i, j] = k$.

If the shortest path does not pass through any intermediate vertex, then $pred[i, j] = nil$.

Extracting the Shortest Paths

To find the shortest path from i to j, we consult $pred[i, j]$. If it is nil, then the shortest path is just the edge (i, j). Otherwise, we recursively compute the shortest path from i to $pred[i, j]$ and the shortest path from $pred[i, j]$ to j.

The Algorithm for Extracting the Shortest Paths

```
Path($i, j$)
{
    if ($pred[i, j] = nil$)    single edge
        output ($i, j$);
    else                compute the two parts of the path
    {
        Path($i, pred[i, j]$);
        Path($pred[i, j], j$);
    }
}
```

Backtracking

- A backtracking algorithm tries to build a solution to a computational problem incrementally.
- Suppose you have to make a series of *decisions*, among various *choices*, where
  - You don't have enough information to know what to choose
  - Each decision leads to a new set of choices
  - Some sequence of choices (possibly more than one) may be a solution to your problem
- Backtracking is a methodical way of trying out various sequences of decisions, until you find one that "works"

Backtracking

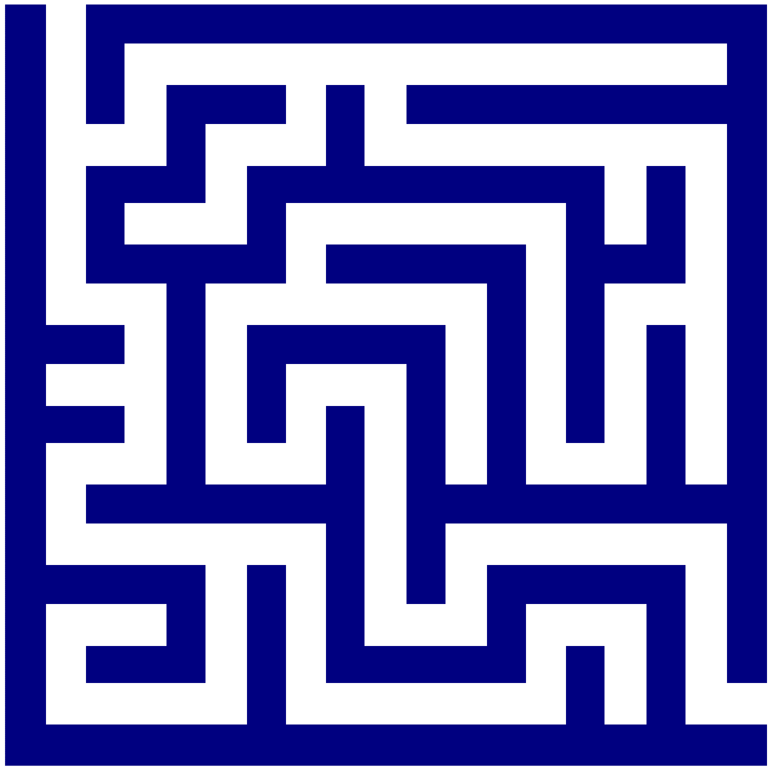
- Is used to solve problems for which a sequence of objects is to be selected from a set such that the sequence satisfies some constraint
- Traverses the state space using a depth-first search with pruning

Backtracking

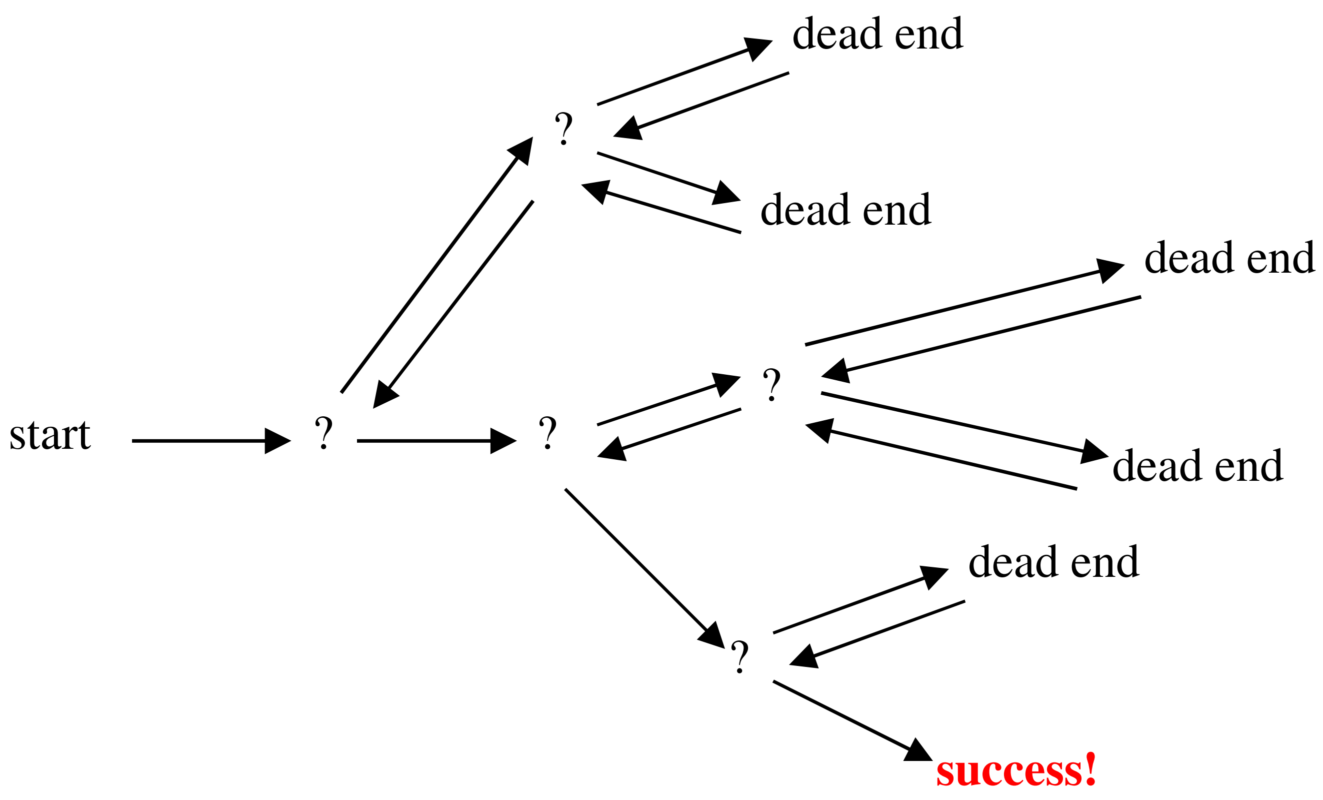
- Performs a depth-first traversal of a tree
- Continues until it reaches a node that is non-viable or non-promising
- Prunes the sub tree rooted at this node and continues the depth-first traversal of the tree

Solving a maze

- Given a maze, find a path from start to finish
- At each intersection, you have to decide between three or fewer choices:
  - Go straight
  - Go left
  - Go right
- You don't have enough information to choose correctly
- Each choice leads to another set of choices
- One or more sequences of choices may (or may not) lead to a solution.
- Many types of maze problem can be solved with backtracking



Backtracking



The Backtracking Method

- **A given problem has a set of constraints and possibly an objective function**
- **The** solution **optimizes an objective function, and/or is feasible.**
- **We can represent the solution space for the problem using a state space tree**
  - **The *root* of the tree represents 0 choices,**
  - **Nodes at depth 1 represent first choice**
  - **Nodes at depth 2 represent the second choice, etc.**
  - **In this tree a *path* from a root to a leaf represents a candidate solution**

SAMSUNG

DTU Delhi Technological University

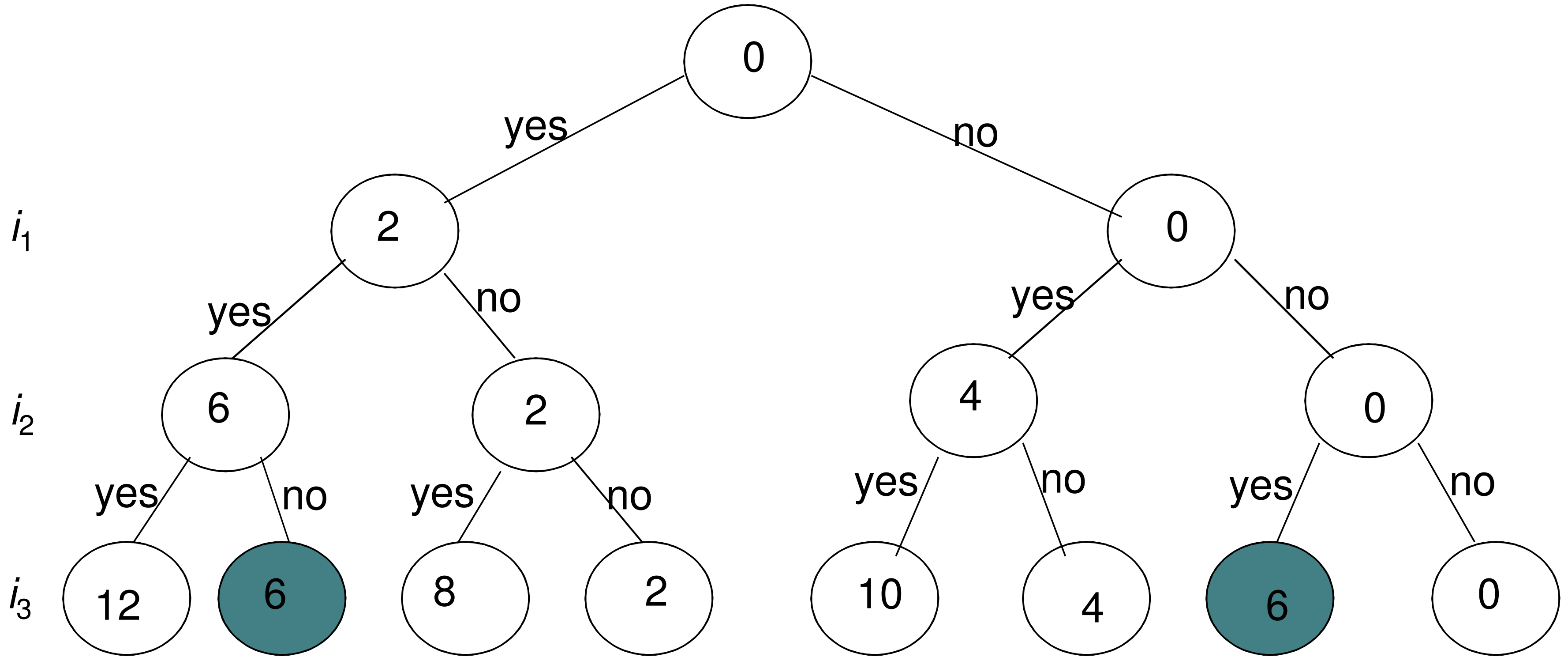
Sum of subsets

- **Problem**: Given n positive integers $w_{1,} \ldots w_n$ and a positive integer S. Find all subsets of $w_{1,} \ldots w_n$ that sum to S.
- **Example**:
  n=3, S=6, and $w_1=2$, $w_2=4$, $w_3=6$
- **Solutions**:
  {2,4} and {6}

- We will assume a binary state space tree.
- The nodes at depth 1 are for including (yes, no) item 1, the nodes at depth 2 are for item 2, etc.
- The left branch includes w_i, and the right branch excludes w_i.
- The nodes contain the sum of the weights included so far

Sum of subset Problem: State SpaceTree for 3 items

$w_1 = 2, \quad w_2 = 4, \quad w_3 = 6$ and $S = 6$



The sum of the included integers is stored at the node.

A Depth First Search solution

- Problems can be solved using depth first search of the (implicit) state space tree.

- Each node will save its depth and its (possibly partial) current solution

- DFS can check whether node v is a leaf.
  - If it is a leaf then check if the current solution satisfies the constraints
  - Code can be added to find the optimal solution

A DFS solution

- Such a DFS algorithm will be very slow.
- It does not check for every solution state (node) whether a solution has been reached, or whether a *partial* solution can lead to a *feasible* solution
- Is there a more efficient solution?

Backtracking solution

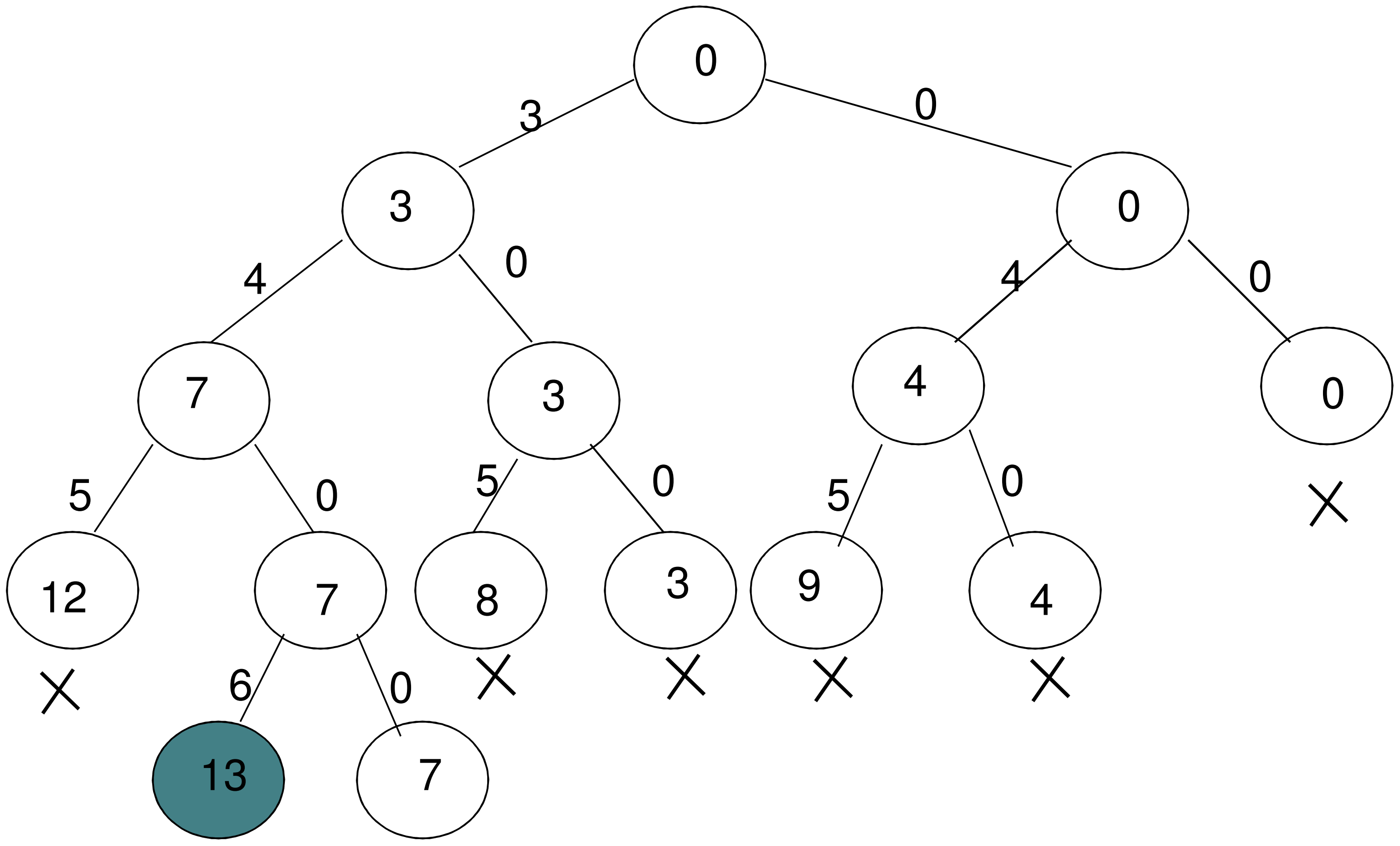
- **Definition:** We call a node *nonpromising* if it cannot lead to a feasible (or optimal) solution, otherwise it is *promising*

- **Main idea:** Backtracking consists of doing a DFS of the state space tree, checking whether each node is promising and if the node is nonpromising backtracking to the node's parent

Backtracking solution

- The state space tree consisting of expanded nodes only is called the *pruned state space tree*
- The following slide shows the pruned state space tree for the sum of subsets example
- There are only 15 nodes in the pruned state space tree
- The full state space tree has 31 nodes

A Pruned State Space Tree (find all solutions)
$w_1 = 3, w_2 = 4, w_3 = 5, w_4 = 6; \; S = 13$



Sum of subsets problem

Backtracking algorithm

```
void checknode (node v) {
  node u
  if (promising ( v ))
      if (aSolutionAt( v ))
            write the solution
      else                            //expand the node
            for ( each child u of v )
                  checknode ( u )
```

Checknode

- Checknode uses the functions:
  - *promising*(*v*) which checks that the partial solution represented by *v* can lead to the required solution
  - *aSolutionAt*(*v*) which checks whether the partial solution represented by node *v* solves the problem.

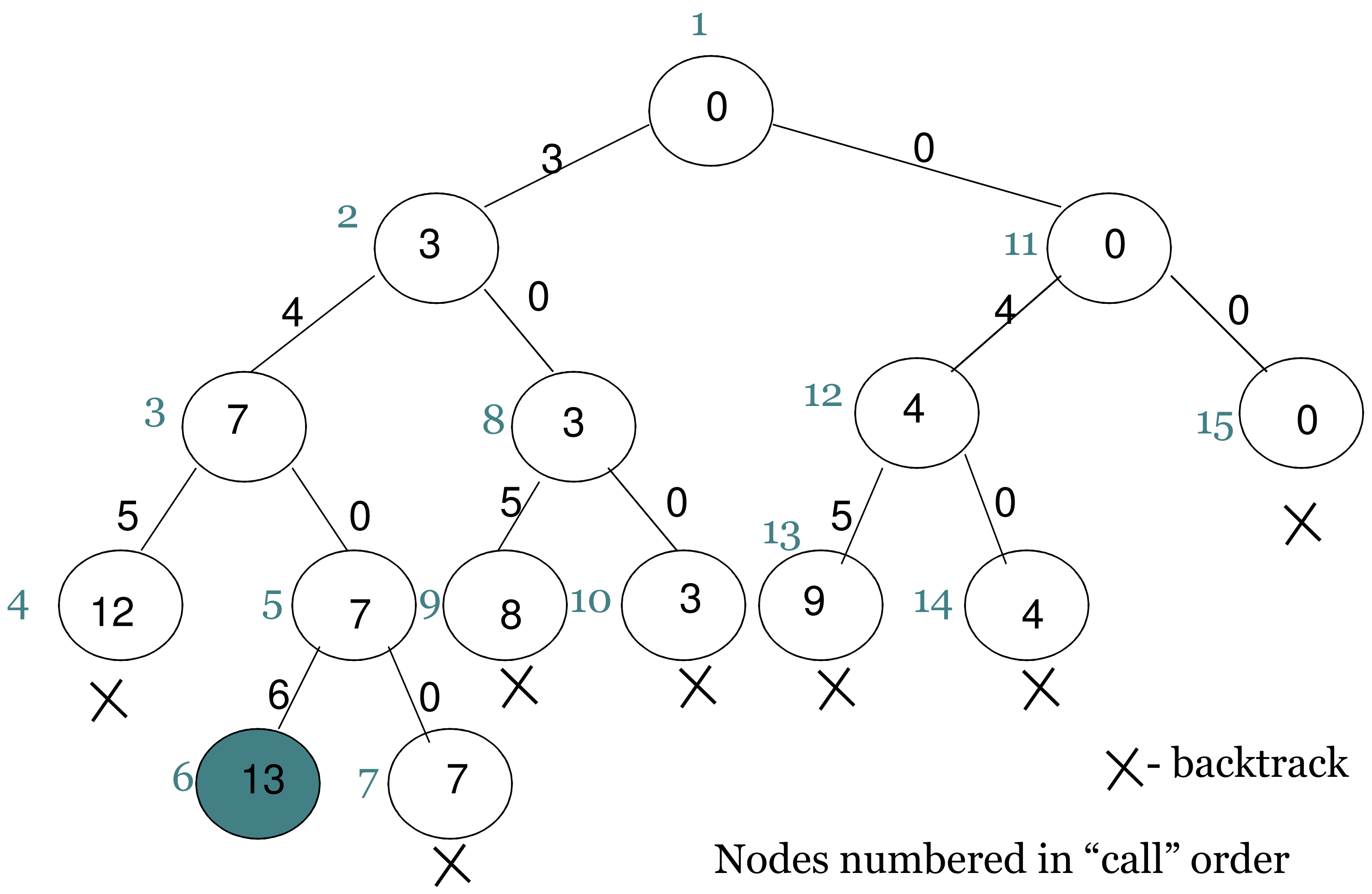
Sum of subsets – when is a node “promising”?

- Consider a node at depth i
- *weightSoFar* = weight of node, i.e., sum of numbers included in partial solution node represents

- *totalPossibleLeft* = weight of the remaining items i+1 to n (for a node at depth i)
- A node at depth i is non-promising
  if (*weightSoFar* + *totalPossibleLeft* < S)
  or (*weightSoFar* + w[i+1] > S)
- To be able to use this “promising function” the w_i must be sorted in non-decreasing order

A Pruned State Space Tree

$w_1 = 3, w_2 = 4, w_3 = 5, w_4 = 6;\ S = 13$



╳- backtrack

Nodes numbered in “call” order

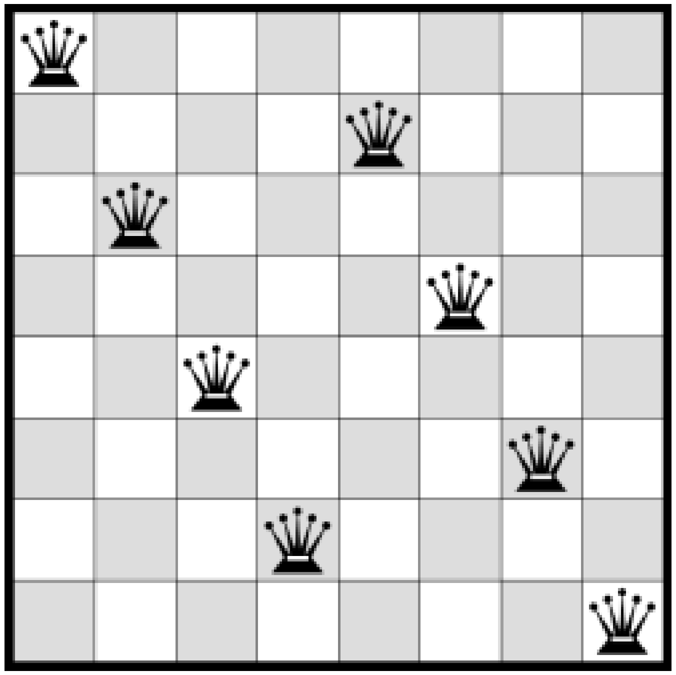
```
sumOfSubsets ( i, weightSoFar, totalPossibleLeft )
   1. if (promising (i))                                //may lead to solution
   2.    then if ( weightSoFar == S )
   3.       then print include[ 1 ] to include[ i ]     //found solution
   4.    else       //expand the node when weightSoFar < S
   5.       include [ i + 1 ] = "yes"                   //try including
   6.       sumOfSubsets ( i + 1,
                        weightSoFar + w[i + 1],
                        totalPossibleLeft - w[i + 1] )
   7.       include [ i + 1 ] = "no"                    //try excluding
   8.       sumOfSubsets ( i + 1,   weightSoFar ,
                              totalPossibleLeft - w[i + 1] )

boolean promising (i )
   1.  return ( weightSoFar + totalPossibleLeft ≥ S)  &&
              ( weightSoFar == S  ||  weightSoFar +  w[i + 1] ≤ S )
```

Prints all solutions!

Initial call sumOfSubsets(0, 0, $\sum_{i=1}^{n} w_i$)

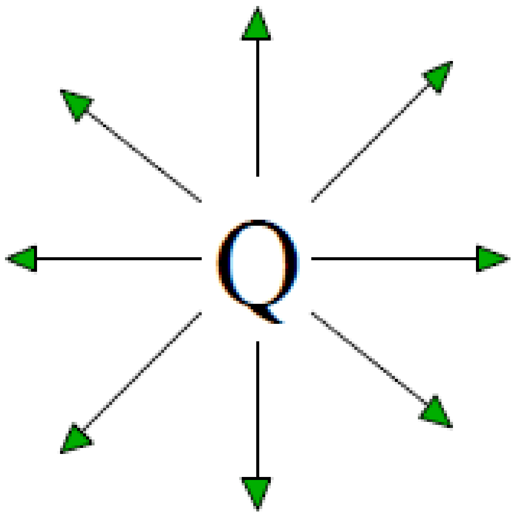
The 8 Queens Problem

- Given is a chess board. A chess board has 8x8 fields. Is it possible to place 8 queens on this board, so that no two queens can attack each other?



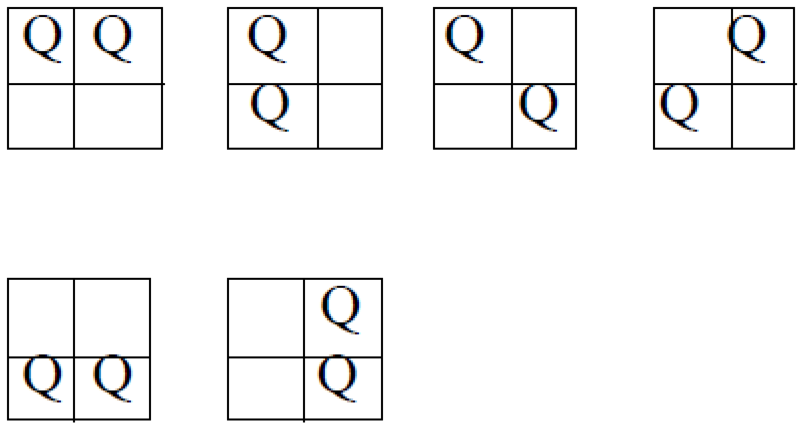
The 8 Queens Problem

- NOTES: A queen can attack horizontally, vertically, and on both diagonals, so it is pretty hard to place several queens on one board so that they don't attack each other.



The n Queens problem:

- The **n Queens problem:**
- Given is a board of **n by n squares. Is it** possible to place **n queens (that behave** exactly like chess queens) on this board, without having any one of them attack any other queen?
- Example: 2 Queens problem is not solvable.

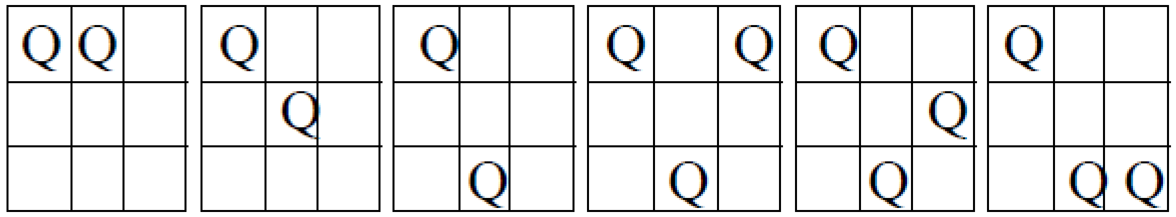


Example 2: The 4-queens problem is solvable

Basic idea of solution:

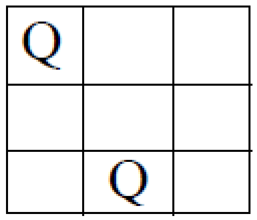
- Start with one queen in the first column, first row.
- Start with another queen in the second column, first row.
- Go down with the second queen until you reach a permissible situation.
- Advance to the next column, first row, and do the same thing.
- If you cannot find a permissible situation in one column and reach the bottom of it, then you have to go back to the previous column and move one position down there. (This is the backtracking step.)
- If you reach a permissible situation in the last column of the board, then the problem is solved.
- If you have to backtrack BEFORE the first column, then the problem is not solvable.

A slow example:

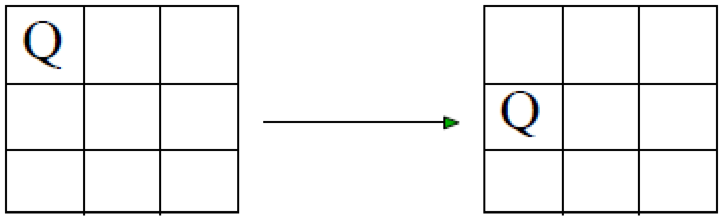


illegal illegal legal illegal illegal illegal

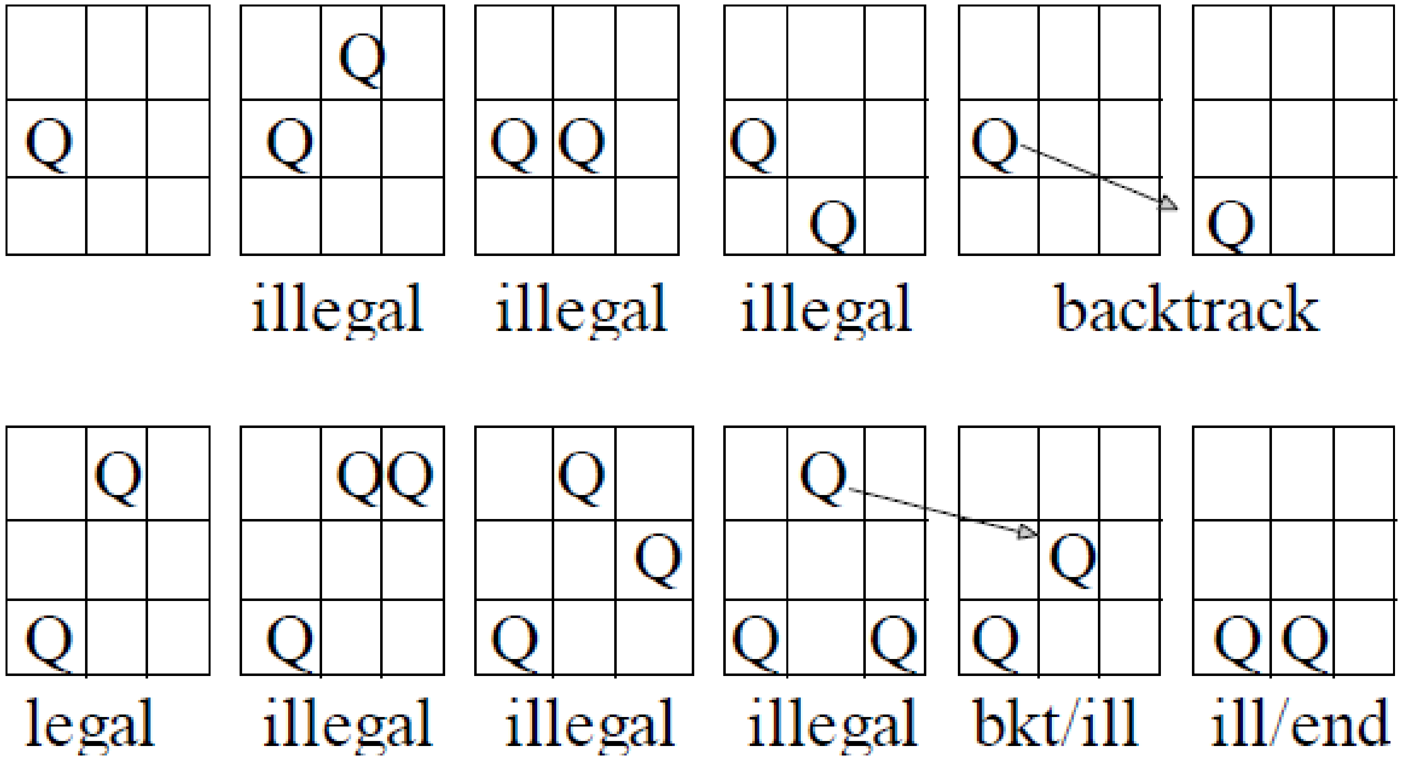
- cannot go further down in row 3. I must backtrack!



However, I cannot go further down in column 2 either. I must backtrack one more step.



Now I start again in the second column.



At this point I am at the end of the first column. I would have to backtrack again, but that's impossible, so the problem is unsolvable.

We will work out a successful example also:

QQ++	Q+++	Q+++	Q+Q+	Q+++	Q+++
++++	+Q++	++++	++++	++Q+	++++
++++	++++	+Q++	+Q++	+Q++	+QQ+
++++	++++	++++	++++	++++	++++
illegal	illegal	legal	illegal	illegal	illegal

Q+++	Q+Q+	Q+++	Q+++	Q+++
++++	++++	++Q+	++++	++++
+Q++	++++	++++	++Q+	++++
++Q+	+Q++	+Q++	+Q++	+QQ+
illegal	illegal	illegal	illegal	illegal

Backtrack to column 2. Then backtrack to column 1. Then go down in column 1.

```
+Q++   3       ++++   ++Q+   ++QQ   ++Q+   ++Q+
Q+++   ill     Q+++   Q+++   Q+++   Q++Q   Q+++
++++   steps   ++++   ++++   ++++   ++++   +++Q
++++           +Q++   +Q++   +Q++   +Q++   +Q++
illegal                      ill    ill    legal
```

I placed 4 queens on a 4x4 board.
Problem solved.

Are there other solutions? There must be, due to summetry:

	Q		
			Q
Q			
		Q	

We can continue the program to find this and other solutions.

Complexity Classes: P and NP

- The **P versus NP problem** is a major unsolved problem in computer science.
- Informally, it asks whether every problem whose solution can be quickly verified by a computer can also be quickly solved by a computer.
- The informal term *quickly* used above means the existence of an algorithm for the task that runs in polynomial time.

P and NP

- The general class of questions for which some algorithm can provide an answer in polynomial time is called "class P" or just "**P**".
- For some questions, there is no known way to find an answer quickly, but if one is provided with information showing what the answer is, it may be possible to verify the answer quickly.
- The class of questions for which an answer can be verified in polynomial time is called **NP**.

NP:Example

- Consider the subset sum problem, an example of a problem that is easy to verify, but whose answer may be difficult to compute.
- Given a set of integers, does some nonempty subset of them sum to 0?
- For instance, does a subset of the set {−2, −3, 15, 14, 7, −10} add up to 0?
- The answer "yes, because{−2, −3, −10, 15} add up to zero" can be quickly verified with three additions.
- However, there is no known algorithm to find such a subset in polynomial time (there is one, however, in exponential time, which consists of 2^n-1 tries), and indeed such an algorithm cannot exist if the two complexity classes are not the same; hence this problem is in **NP** (quickly checkable) but not necessarily in **P** (quickly solvable).

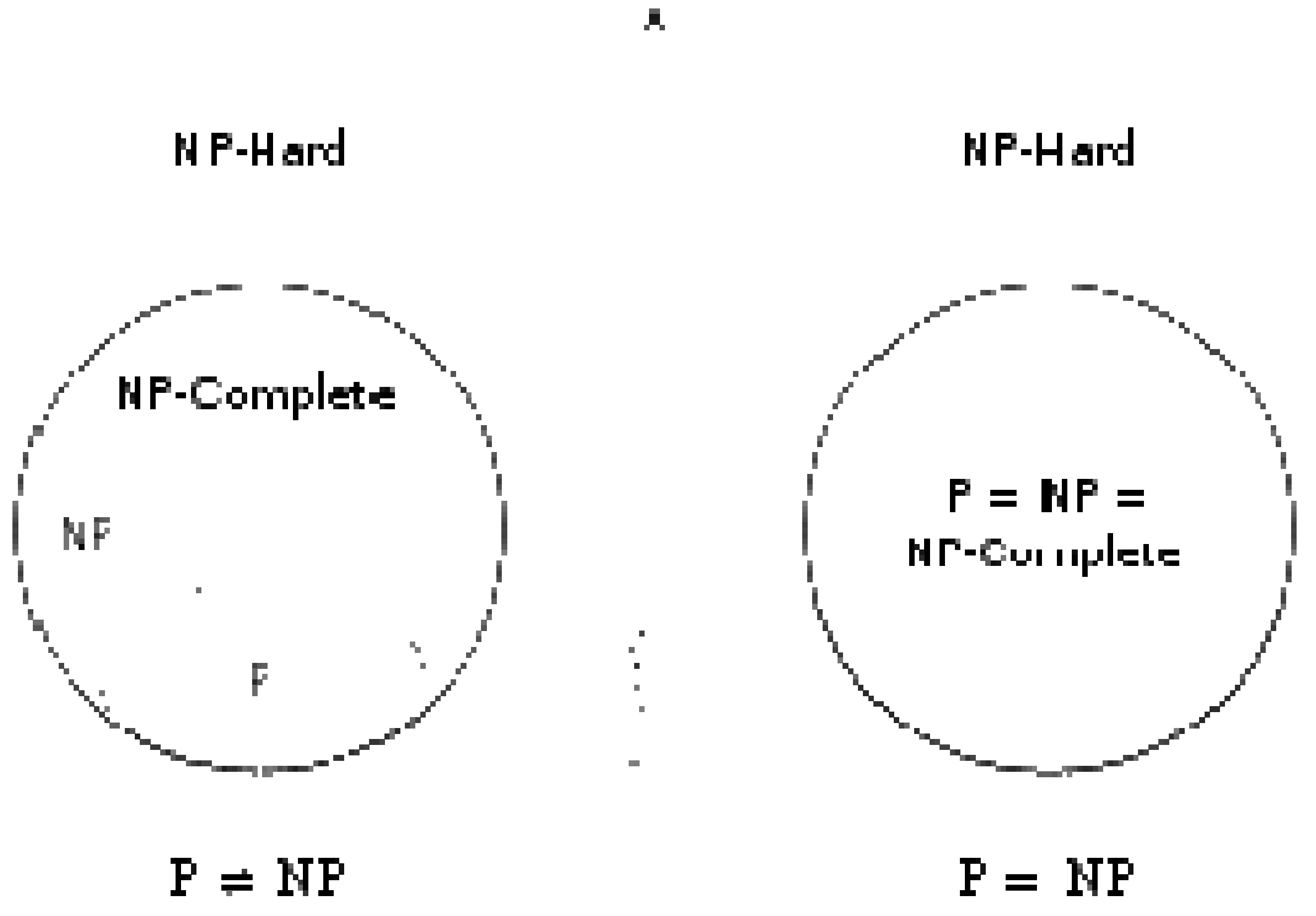
SAMSUNG

DTU Delhi Technological UNIVERSITY

- An answer to the **P = NP** question would determine whether problems that can be verified in polynomial time, like the subset-sum problem, can also be solved in polynomial time.
- If it turned out that **P** does not equal **NP**, it would mean that there are problems in **NP** (such as NP-complete problems) that are harder to compute than to verify: they could not be solved in polynomial time, but the answer could be verified in polynomial time.

NP Complete and NP Hard

- To attack the **P** = **NP** question the concept of **NP**-completeness is very useful.
- **NP**-complete problems are a set of problems to which any other **NP**-problem can be reduced in polynomial time, and whose solution may still be verified in polynomial time.
- Informally, an **NP**-complete problem is at least as "tough" as any other problem in **NP**.
- NP-hard problems are those at least as hard as **NP**-complete problems, i.e., all **NP**-problems can be reduced (in polynomial time) to them. **NP**-hard problems need not be in **NP**, i.e., they need not have solutions verifiable in polynomial time.



Euler diagram for P, NP, NP-complete, and NP-hard set of problems

- For instance, the boolean satisfiability problem is **NP**-complete.
- So *any* instance of *any* problem in **NP** can be transformed mechanically into an instance of the boolean satisfiability problem in polynomial time.
- The boolean satisfiability problem is one of many such **NP**-complete problems. If any **NP**-complete problem is in **P**, then it would follow that **P = NP**.
- Unfortunately, many important problems have been shown to be **NP**-complete, and as of 2012 not a single fast algorithm for any of them is known.

NP Hard problems

- An example of an NP-hard problem is the decision subset sum problem, which is this: given a set of integers, does any non-empty subset of them add up to zero?
- That is a decision problem, and happens to be NP-complete.
- Another example of an NP-hard problem is the optimization problem of finding the least-cost cyclic route through all nodes of a weighted graph.
- This is commonly known as the traveling salesman problem.