Backtracking and Branch-and-Bound

- Usually for problems with high complexity
- Exhaustive Search is too time consuming
- Cut down on some search using special methods
- Idea: Construct partial solutions and extend
- Smart method to extend partial solutions can lead to faster solutions
- If the current partial solution cannot lead to a full solution, prune
- If the current solution is worse than some earlier known solution, prune
- This approach makes it possible to sometimes solve large problems in reasonable time
- Worst case is still too much time

- Backtracking: Applies to non-optimization problems
- Branch and Bound: only for optimization problems.
- Backtracking: usually DFS
- Branch and Bound: usually BFS

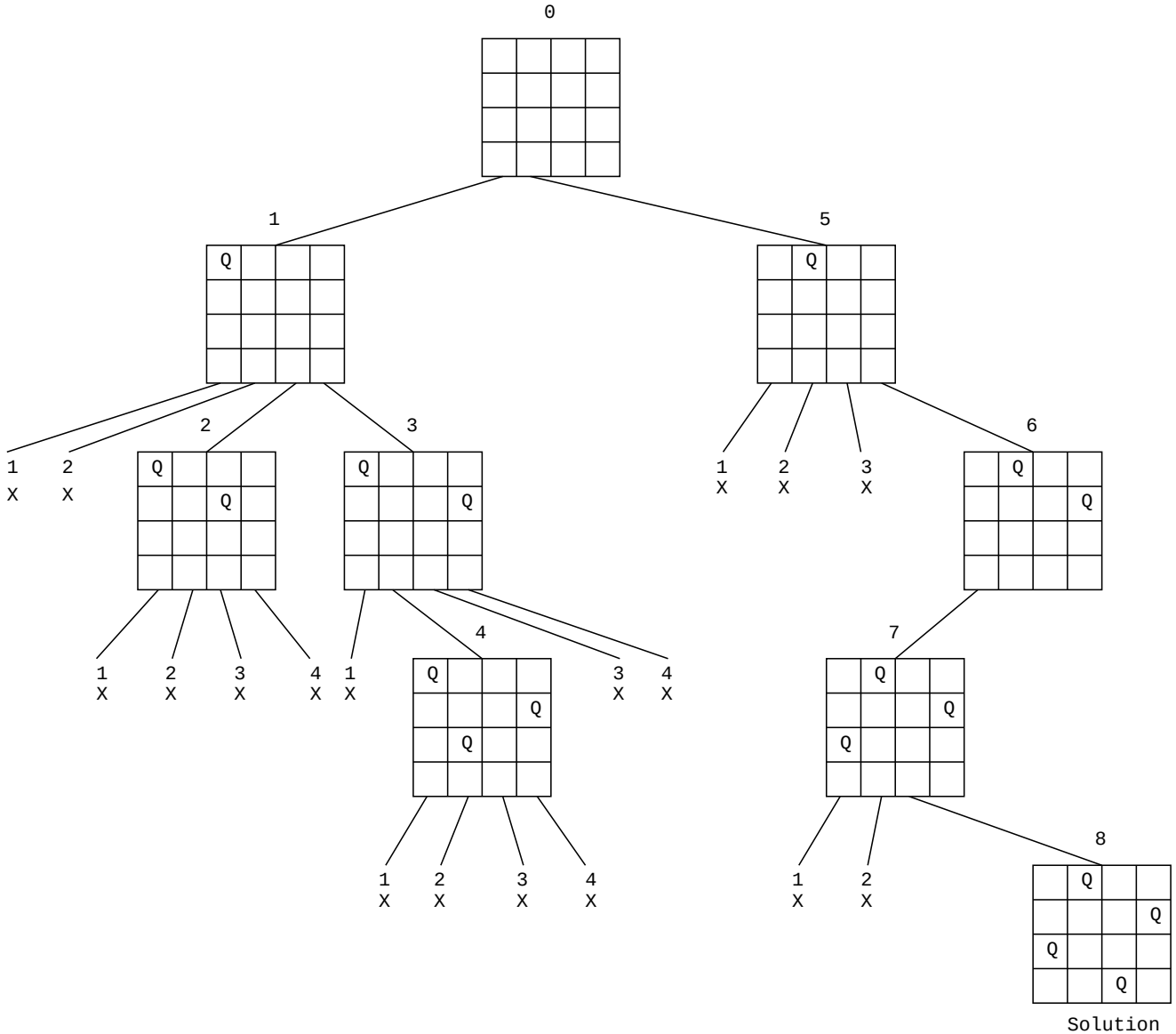
- Search Tree
- Root represents initial state,
- Each child of a node represents how the current partial solution maybe extended towards full solution
- Leaves: possible solutions
- partial solutions may be promising (can lead to a (good) solution) or non-promising (cannot lead to a solution or solution will be poor in quality)
- Prune non-promising nodes

n-Queens Problem

- Place n queens on an n-by-n chess board so that no two of them 'attack' each other, that is, no two of them are in the same row, same column or same 'diagonal'.
- Can consider Chess Board as $n \times n$ matrix. So, if two queens are at (i, j) and (i', j'), then we want $i \neq i'$, $j \neq j'$ and $i - i' \neq j - j'$.
- Ex.: $n = 4$:

	Q		
			Q
Q			
		Q	

Search Space/State Space



Travelling Salesman Problem

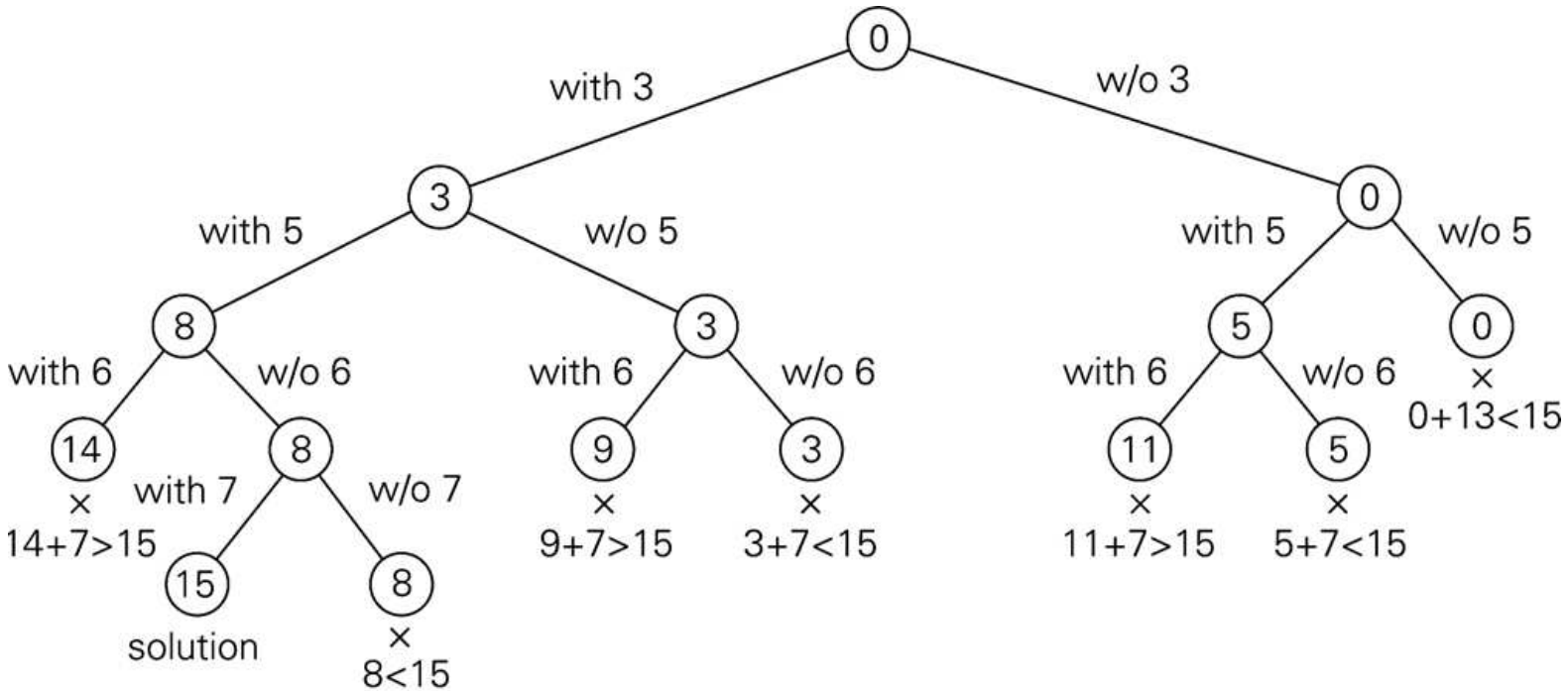
- To find a circuit going through all the vertices exactly once, with least weight or weight $\leq X$.
- Assume complete graph (where edge weights can be "large" (∞) to denote missing edges.
- Root can denote just the starting vertex.
- For any node, each child denotes adding one more vertex to the path.
- Depth of the tree is n. Root has n branches, and each intermediate node has $n - \ell$ branches based on its distance ℓ from the root.
- Cost upto a node is the sum of weights on the path so constructed upto that point.
- Each leaf represents a possible solution, and we need to find least cost leaf (or a leaf of cost $\leq X$).

Subset Sum problem

- Problem: Given a set $S = \{s_1, s_2, \ldots, s_n\}$ of n positive integers. Find a subset of S whose sum is d. Example: $S = \{1, 5, 7, 4, 3\}$, $d = 12$, then answer could be $\{5, 7\}$ or $\{4, 3, 5\}$.
- Root can be empty subset.
- Nodes at the ℓ-th level decide whether to add/not add s_i to the sum.
- If current sum $> d$, then the sum is too high, and any further search in this branch can be pruned.
- If current sum plus sum of remaining numbers is $< d$, then sum is too small, and any further search in this branch can be pruned.
- If $sum = d$, then done.

Subset-Sum Problem

- Example: $S = \{3, 5, 6, 7\}$, $d = 15$.



Backtracking Algorithm

Backtrack(node: v)
If v represents a solution, Print v
If no descendant of v can be a solution, then Halt
    (* Pruning in above case *)
Else
For each child X of node do
    Backtrack(X)
    (* Here may need to calculate some information
       regarding X *)
EndFor

- Can stop the algorithm after first solution is found
- In worst case the whole search space is searched
- Even then there is a systematic way of doing it
- Order of exploring the children can often speed up the search
- Here heuristics come into picture: which child is more promising? Usually in such cases we have a way to give value to partial solutions. For example in chess, summing up the values of pieces can give a way to compare the different partial solutions.

Branch and Bound

- For optimization Problems
- More similar to BFS than to DFS (though can be done DFS way)
- At any point, expand the most promising partial solution
- Prune if the nodes value is no better than the best solution already found
- Partial solutions can be kept in heap/queue

Knapsack Problem

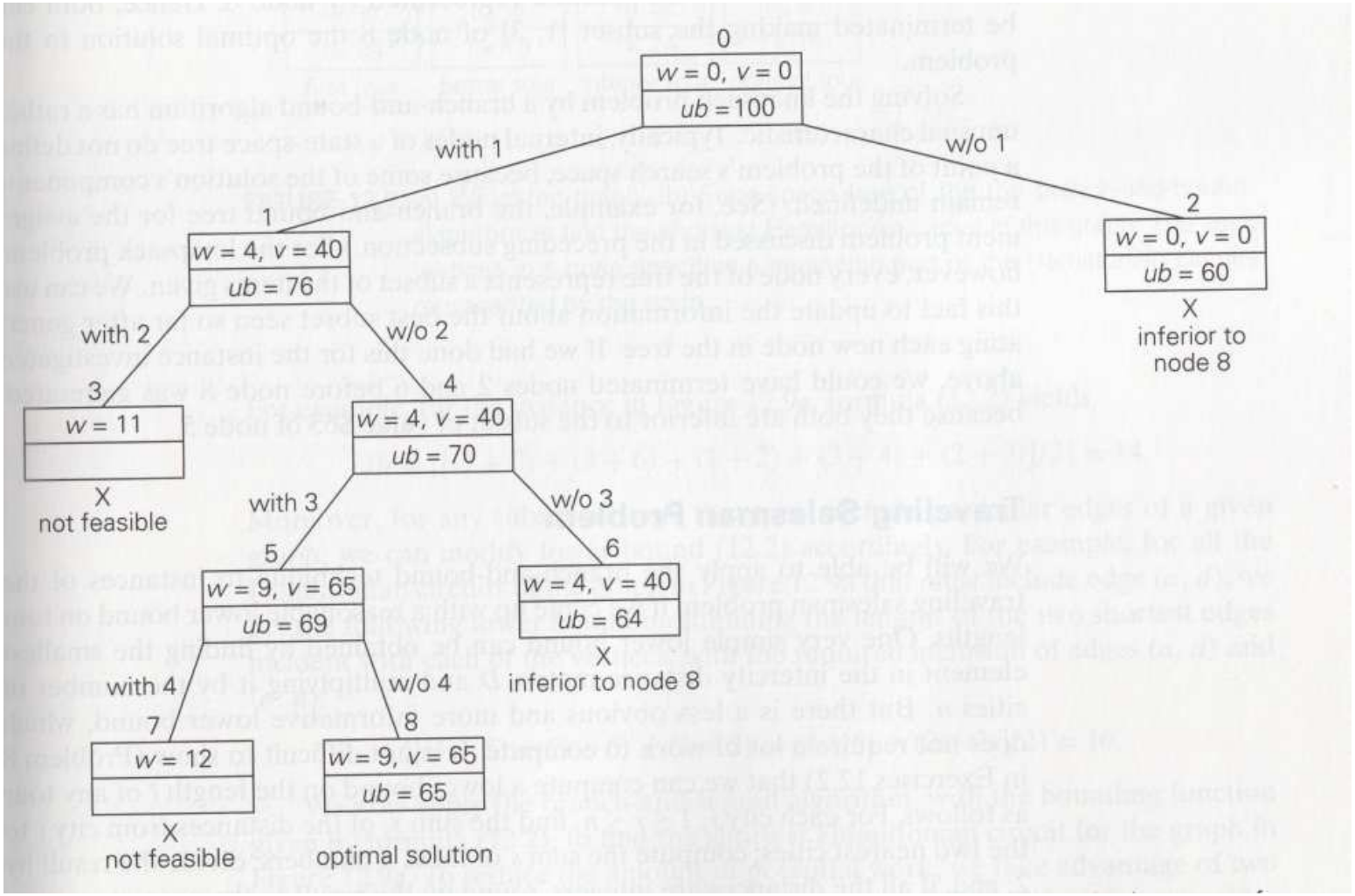
- Recall that in Knapsack problem, there are n objects $o_1, o_2, \ldots, o_n$, with weights $w_1, w_2, \ldots, w_n$ and values $v_1, v_2, \ldots, v_n$.
- There is knapsack capacity K
- Choose a subset of objects which maximises the value subject to the constraint that total weight ie $\leq K$. That, is find, $S \subseteq \{1, 2, \ldots, n\}$ such that $\Sigma_{i \in S} v_i$ is maximised subject to the constraint that $\Sigma_{i \in S} w_i \leq K$.

Ex.: Knapsack Problem

- The knapsack's capacity W is 10.

item	weight	value	$\frac{\text{value}}{\text{weight}}$
1	4	\$40	10
2	7	\$42	6
3	5	\$25	5
4	3	\$12	4

Ex.: Knapsack Problem



Branch-and-Bound

$PromisingSet = \{root\}$
currentbestvalue=∞ or $-\infty$ based on problem type
While $PromisingSet \neq \emptyset$ Do
    Pick a promising member v of the PromisingSet
    If v is a solution, then update currentbest value/solution
    Otherwise, expand v:
        For each child w of v
            If w cannot lead to a solution, then prune (i.e. do not add w to the PromisingSet)
            Else If solutions by w and its descendants is worse than currentbestvalue, then prune
            Else add w to the PromisingSet
        EndFor
EndWhile
Print the bestvalue solution

$PromisingSet = \{root\}$
Done $\leftarrow$ false
While $PromisingSet \neq \emptyset$ and not Done Do
    Pick the most promising member v of the PromisingSet
    If v is a solution, then Done $\leftarrow$ true and print the best
       solution as given by v
    Otherwise, expand v:
        For each child w of v
            If w cannot lead to a solution, then prune (i.e.
               do not add w to the PromisingSet)
            Else add w to the PromisingSet
        EndFor
EndWhile

- How to give value to a partial solution?
- Heuristics
- Should be "upper bound" for maximization problems, and lower bound for "minimization problems"

- Travelling Salesman Problem
- Sum of edges used in the partial tour constructed at a node.
- Sum of edges used in the partial tour constructed at a node plus minimum value of the edges going out from the remaining vertices
- Sum of edges used in the partial tour constructed at a node plus minimum value of the edges going out from the remaining vertices to one of the remaining vertices (plus the starting vertex)
- Take both incoming and outgoing “different edges” for the remaining vertices, except we need only entry to starting vertex and exit from the last vertex in the path constructed so far: take half of these costs as we need to avoid counting twice.