

# Branch and Bound

CS31005: Algorithms-II

Autumn 2020

IIT Kharagpur

# Branch and Bound

- An algorithm design technique, primarily for solving hard optimization problems
- Guarantees that the optimal solution will be found
- Does not necessarily guarantee worst case polynomial time complexity
  - But tries to ensure faster time on most instances
- Basic Idea
  - Model the entire solution space as a tree
  - Search for a solution in the tree systematically, eliminating parts of the tree from the search intelligently
- Important technique for solving many problems for which efficient algorithms (worst case polynomial time) are not known

# Optimization Problems

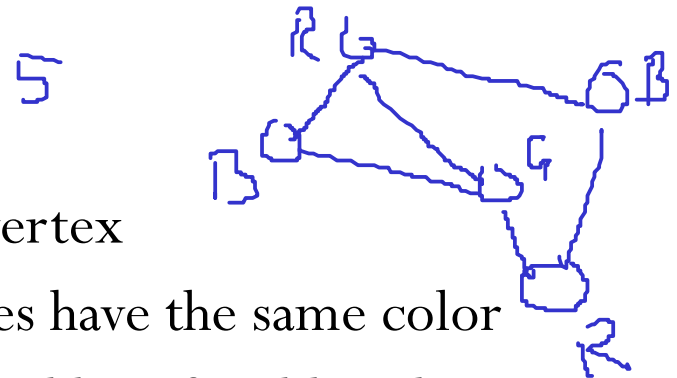
- Set of input variables  $I$
- Set of output variables  $O$ 
  - Values of the output variables define the solution space
- Set of constraints  $C$  over the variables
- Set of feasible solutions  $S$ 
  - Set of solutions that satisfy all the constraints
- Objective function  $F: S \rightarrow \mathbb{R}$  (also called *cost function*)
  - Gives a value  $F(s)$  for each solution  $s \in S$
- Optimal solution
  - A solution  $s \in S$  for which  $F(s)$  is maximum among all  $s \in S$  (for a maximization problem) or minimum (for a minimization problem)

# Example

- Many problems that you have seen so far
  - Minimum weight spanning tree, matrix chain multiplication, longest common subsequence, fractional knapsack problem, ....
  - All these (that you have seen in Algorithms-1 course) have known efficient algorithms

- Graph Coloring

- Input: an undirected graph
- Output: a color assigned to each vertex
- Constraint: no two adjacent vertices have the same color
- Objective function: no of colors used by a feasible solution
- Goal: Find a coloring that uses the minimum number of colors
- No efficient algorithm known for general graphs



- Bin Packing

- Input: a set of items  $X = \{x_1, x_2, \dots, x_n\}$  with volume  $\{v_1, v_2, v_n\}$  respectively, and set of bins  $B$ , each of volume  $V$

- Output: an assignment of the items to bins (each item can be placed in exactly one bin)

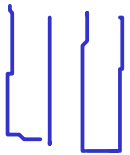
- Constraint: the total volume of items placed in any bin must be  $\leq V$

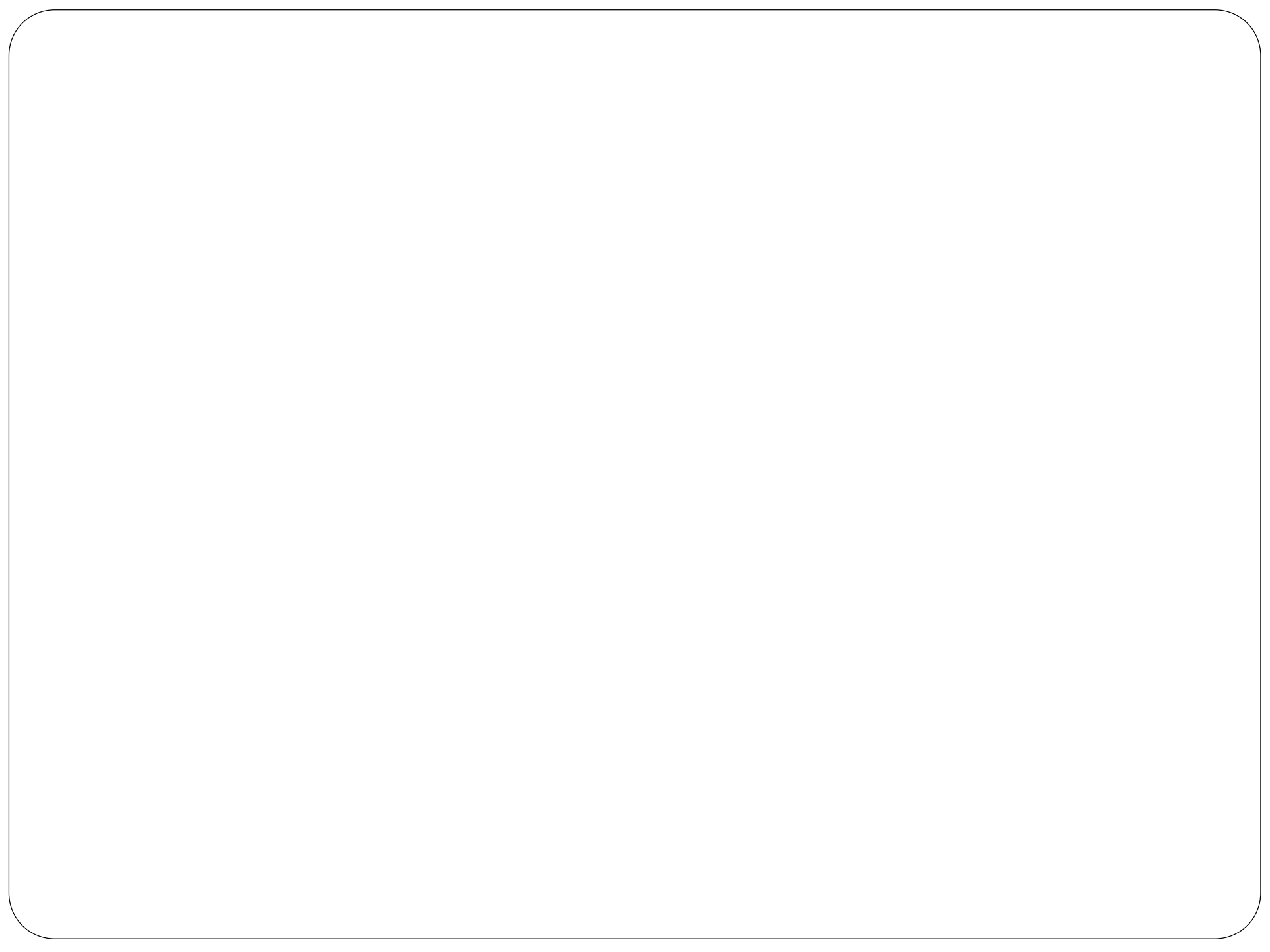
- Objective function: number of bins used

- Goal: find an assignment of items to bins that uses the minimum number of bins

- No efficient algorithm known

- Note that a maximization problem can be transformed into a minimization function (and vice-versa) by just changing the sign of the objective function





# Solution Structure

- For many optimization problems, the solution can be expressed as a  $n$ -tuple  $\langle x_1, x_2, \dots, x_n \rangle$  where each  $x_i$  is chosen from some finite set  $S_i$
- Let size of  $S_i$  be  $m_i$
- Then size of the solution space =  $m_1 m_2 m_3 \dots m_n$
- Some of these solutions are feasible, some are infeasible
- Brute force approach
  - Search the entire space for feasible solutions
  - Compute the cost function for each feasible solution found
  - Choose the one that maximizes/~~min~~imizes the objective function
- Problem: Solution space size may be too large

# Example: 0-1 Knapsack Problem

- Given a set of  $n$  items  $1, 2, \dots, n$  with weights  $w_1, w_2, \dots, w_n$  and values  $v_1, v_2, \dots, v_n$  respectively, and a knapsack with total weight capacity  $W$ , find the largest subset of items that can be packed into the knapsack such that the total value gained is maximized.
- Solution format  $\langle x_1, x_2, \dots, x_n \rangle$ 
  - $x_i = 1$  if item  $i$  is chosen in the subset, 0 otherwise
- Feasible solution:  $\sum x_i w_i \leq W$
- Objective function  $F: \sum x_i v_i$
- Optimal solution: feasible solution with maximum value of  $\sum x_i v_i$
- Solution space size  $2^n$



0 1 1 0 1 0

chosen items 2, 3, 5

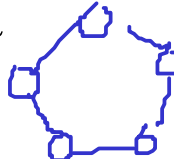
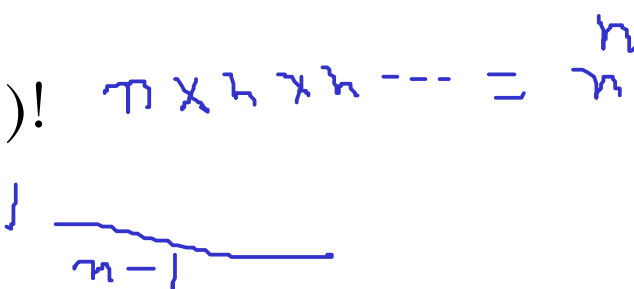
$w_1 = 5, w_2 = 10, w_3 = 2, w_4 = 5 \dots$

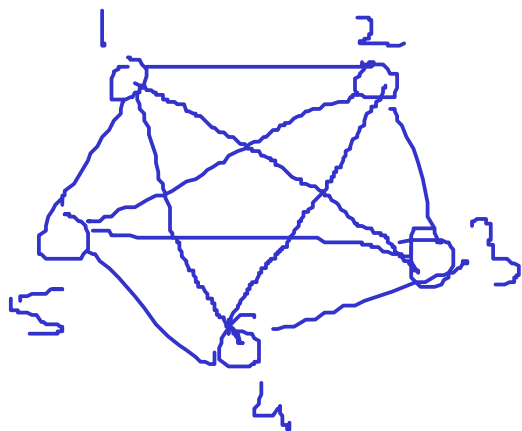
$W = 10$

~~1 1 0 0 ...~~

0 1 0 0 ... all 0's ✓

# Example: Travelling Salesman Problem

- Given a complete weighted <sup>undirected</sup> graph  $G = (V, E)$ , find a Hamiltonian Cycle with the lowest total weight 
- Suppose that the vertices are numbered  $1, 2, \dots, |V| = n$
- Solution format  $\langle x_1, x_2, \dots, x_n \rangle$ 
  - $x_i \in \{1, 2, \dots, n\}$  gives the  $i$ -th vertex visited in the cycle
- Feasible solution:  $x_i \neq x_j$  for any  $i \neq j$
- Objective function  $F: \sum_{1 \leq i < n} w(x_i, x_{i+1}) + w(x_n, x_1)$ , where  $w(i, j)$  is the weight of edge  $(i, j)$
- Optimal solution: feasible solution with minimum value of objective function
- Solution space size  $(n-1)!$   $n \times n \times n \dots = n$   




What is the no. of H-cycles?

$\langle 1, 2, 3, 4, 5 \rangle$

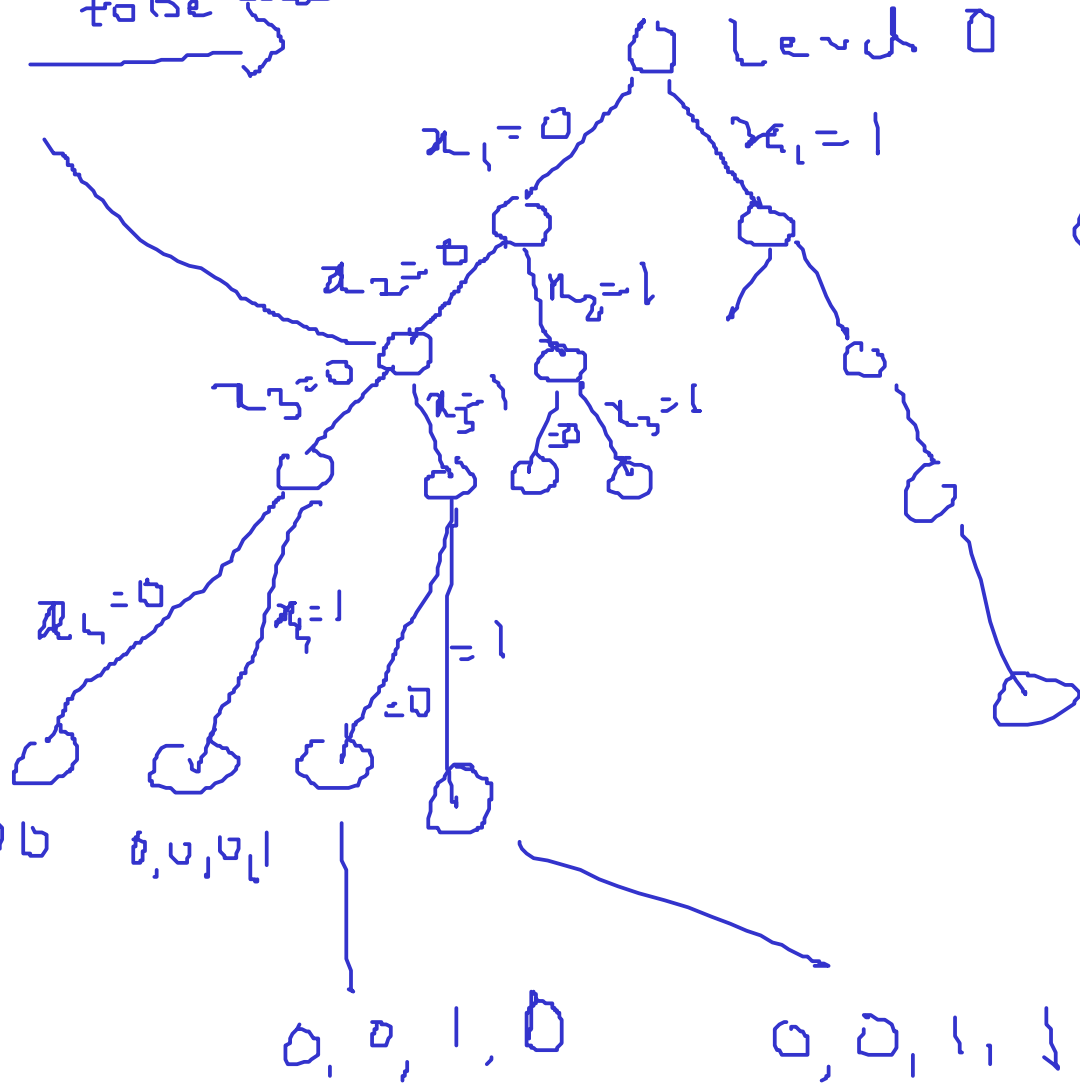
$\langle 2, 3, 4, 1, 5 \rangle$

# State Space Tree

- Represent the solution space as a tree
  - Each edge represents a choice of one  $x_i$ 
    - Level 0 to Level 1 edges show choice of  $x_1$
    - Level 1 to Level 2 edges show choice of  $x_2$
    - Level  $i - 1$  to Level  $i$  edges show choice of  $x_i$
  - Each internal node represents a partial solution
    - Partitions the solution space into disjoint subspaces
  - Leaf nodes represent the complete solution (may or may not be feasible)
- Models the complete solution being built by choosing one component at a time

$n=4$

fixed  $(0, 0, 1)$  to be chosen

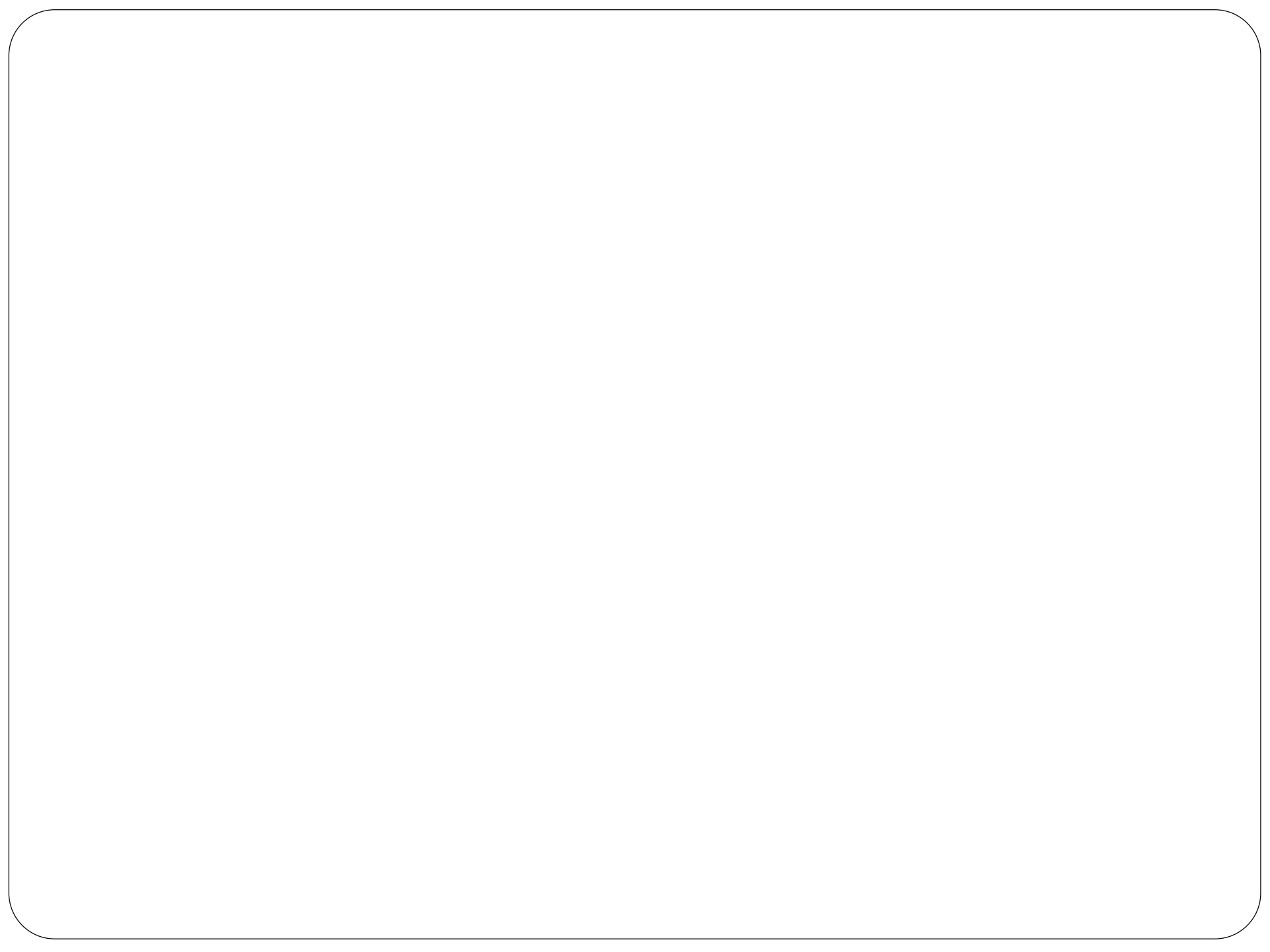


Each node = a state

$w=5$

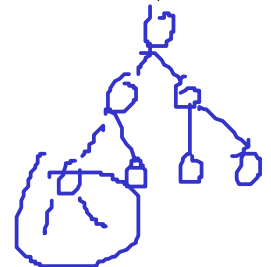
# Example: 0-1 Knapsack

- Level 0 to Level 1 edges show choice of  $w_1$
- Level 1 to Level 2 edges show choice of  $w_2$
- Level  $i - 1$  to Level  $i$  edges show choice of  $w_i$
- Level 0 (root) node partitions the solution space into those that contain  $w_1$  and those that do not contain  $w_1$
- For the subtree which has  $w_1$  chosen, Level 1 nodes partitions the subspace ( $w_1$  present) further into ( $w_1$  and  $w_2$  present) and ( $w_1$  present but  $w_2$  not present)
- Leaf nodes represent the solutions (the path from root to leaf shows what items are chosen (edges marked 1 along the path))



# Finding the Optimal Solution

- One possible approach
  - Generate the state space tree, look at all solutions (leaf nodes), find the feasible solutions, apply objective function on each feasible solution, and choose the optimal
- But generating the tree is as good as doing brute force search
  - Will take huge space (to store the tree) and huge time (to generate all nodes)
  - Need to do something better
    - To reduce space, do not generate all nodes at once
    - To reduce time, do not generate all nodes (how to decide?)
- We will first look at the problem of finding just one feasible solution to understand a basic technique



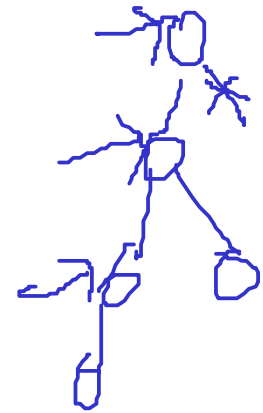


# Finding One Feasible Solution

# Basic Approach

- Expand the tree systematically in parts, stop when you find a feasible solution
  - Reduces space as whole tree is not generated at one go
    - The parts that have been looked at already are not stored
  - May reduce time for many instances as feasible solution may be found without generating the whole tree
    - But in worst case, you may still have to generate all the nodes

- How to expand the tree?
  - Start with root node
  - Generate other nodes in some order
    - DFS, BFS, .....
  - **Live node**: a node which is generated but all children of which has not been generated
  - **Dead node**: A generated node which is not to be expanded (will see why later) or whose all children have been generated
  - The different node generation orders will pick one live node to expand (generate its children) at one time
  - **E-node**: The live node being expanded currently



# Example: $n$ -Queens Problem

- Consider a  $n \times n$  board
- You have to place  $n$  queens on the  $n \times n$  squares so that no two queens can attack each other, i.e., no two queens should be
  - In the same row
  - In the same column
  - In the same diagonal

# One Solution for 8-queens

column →

	1	2	3	4	5	6	7	8
1				Q				
2						Q		
3								Q
4		Q						
5							Q	
6	Q							
7			Q					
8					Q			

row ↓

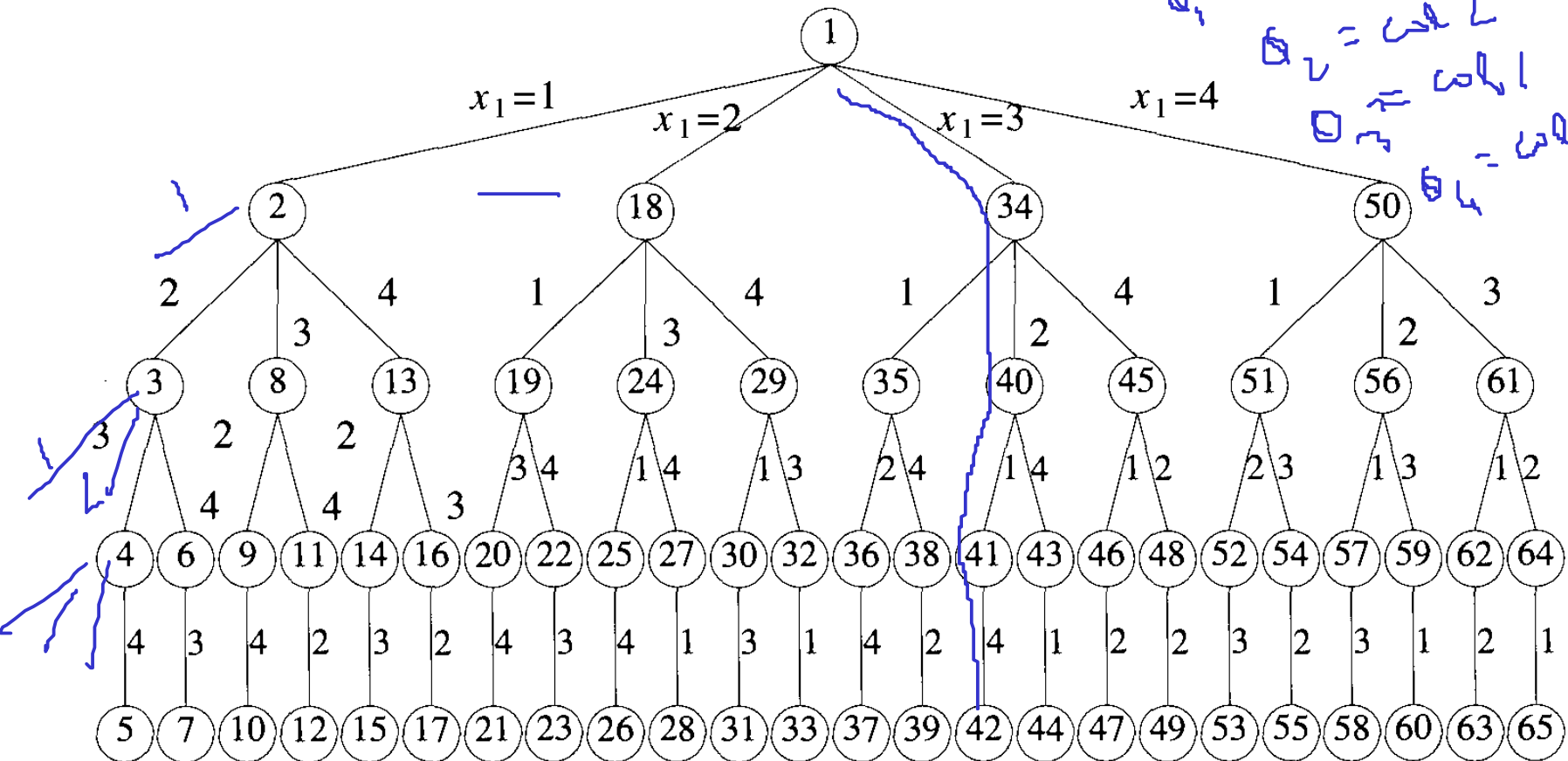
0 1 - row 1  
0 2 - row 2  
⋮

- How to find a solution?
  - Say queens are numbered 1 to  $n$
  - Rows and columns are also numbered from 1 to  $n$
  - Without loss of generality, assume that queen  $i$  is placed in row  $i$
  - So need to find the column in which each queen  $i$  needs to be placed
  - Solution format:  $(x_1, x_2, \dots, x_n)$  where  $x_i$  gives the column no. that queen  $i$  is placed in
    - So  $x_i \in \{1, 2, 3, \dots, n\}$

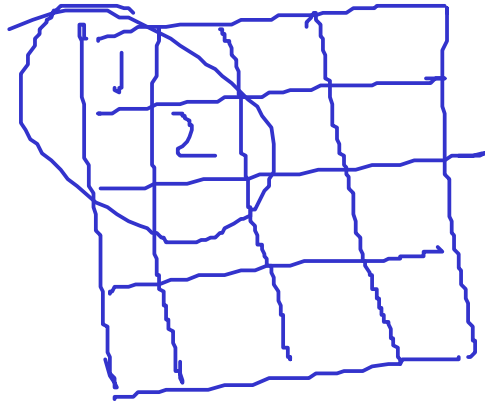
# State Space Tree for 4-Queens

Generated in DFS order (no. inside node shows order)

$n = 4$   
 $Q_1 = \text{col 3}$   
 $Q_2 = \text{col 2}$   
 $Q_3 = \text{col 1}$   
 $Q_4 = \text{col 4}$



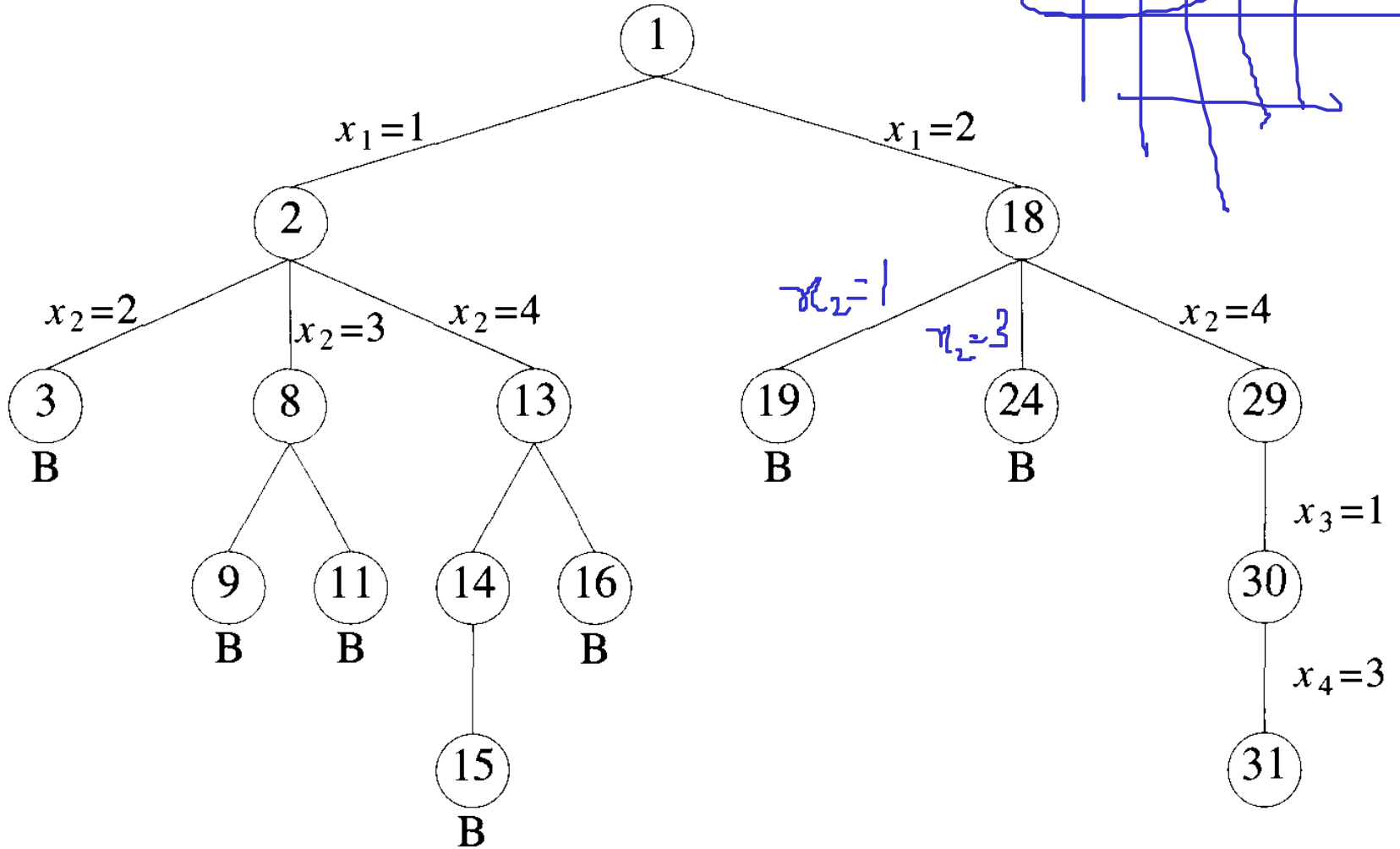
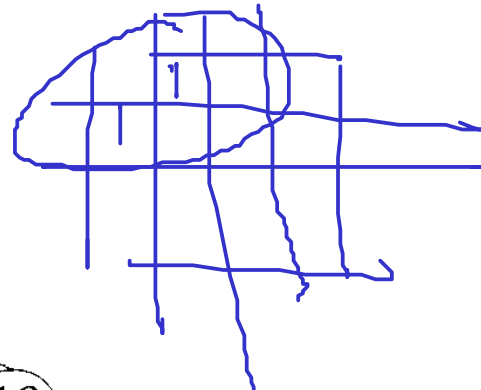
$$\lambda_1 = 1, \lambda_2 = 2$$





- But do we need to explore the entire tree?
  - Consider the node marked 3, corresponding to the choices (made so far) of  $x_1 = 1, x_2 = 2$
  - But this cannot lead to any feasible solution
    - Queen 1 and 2 are on same diagonal
    - Cannot be feasible irrespective of the consequent choice of  $x_3, x_4, \dots$
  - So while generating the tree, no need to generate the subtree rooted at node 3
  - Many other such cases in the tree...
  - Can prune (not generate) large parts of the tree, saving time

# Pruned Tree

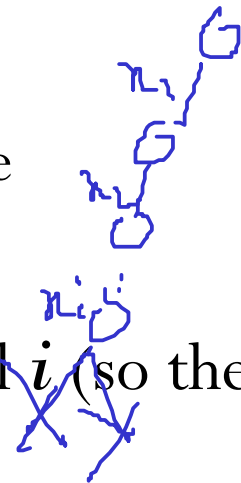


# Backtracking

- Systematic search of the state space tree with pruning
  - Pruning done using **bounding functions**
- Tree generated using DFS order
  - When a child C of the current E-node R is generated, C becomes the new E-node
  - R will become the E-node again after subtree rooted at C is fully explored
- At every step, apply the bounding function (a predicate) on a node to check if the subtree rooted at the node needs to be explored
  - Do not generate the subtree rooted at the node if the bounding function returns false
- Find all feasible solutions (or stop at the first one if only one is needed)

# Example: Subset Sum Problem

- Given a set  $S$  of  $n$  positive integers  $a_1, a_2, \dots, a_n$  and a positive integer  $M$ , find a subset of  $S$  that sums to  $M$
- Solution format  $\langle x_1, x_2, \dots, x_n \rangle$ 
  - $x_i = 1$  if  $a_i$  is chosen in the subset, 0 otherwise
- Feasible solution:  $\sum x_i a_i = M$
- Possible bounding functions at a node at Level  $i$  (so the path to the node is  $\langle x_1, x_2, x_3, \dots, x_i \rangle$ )
  - If  $\sum_{1 \leq k \leq i} x_k a_k > M$  return false
  - If  $\sum_{1 \leq k \leq i} x_k a_k + \sum_{i+1 \leq j \leq n} a_k < M$  return false



# Example: 0-1 Knapsack

- Possible bounding function at a node at Level  $i$ 
  - If  $\sum_{1 \leq k \leq i} x_k w_k > W$  return false
- Note that for 0-1 Knapsack, finding just one feasible solution does not make much sense
  - Choosing any one item only (assuming trivially that each has weight  $< W$ ) is a feasible solution, no need to do backtracking or any other thing
- We will see next that we will have to generate the tree to find the optimal solution
  - So bounding function is still very useful
  - We will augment this simple bounding function later

# Some Notes on Backtracking

- While most definitions of backtracking specify DFS order generation of state-space tree, other definitions do not restrict the order to DFS
  - We will discuss other orders later in the lecture
- Backtracking can be applied to problems other than optimization problems also
  - Just answer yes/no (decision problem)
  - Find one/all feasible solution
  - Some definitions actually restrict backtracking to non-optimization problems only, with branch and bound for optimization problems
- Forms the basis for state space search for optimization problems also
- Branch and bound uses backtracking with different tree generation order and augmented bounding function to prune the tree further for finding an optimal solution

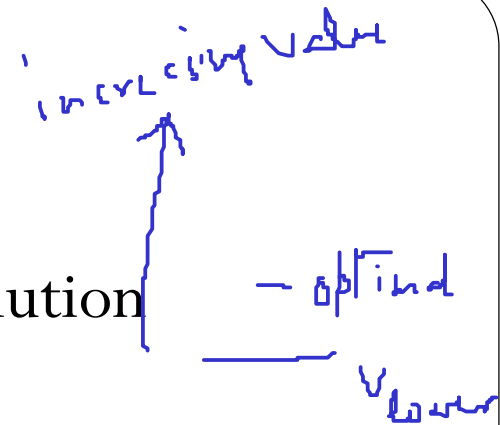
# Going Back to Finding the Optimal Solution: Branch and Bound

- Use similar methods as backtracking to generate the state space tree
  - But need not be DFS order
- Use the bounding function to prune off parts of the tree
  - Parts that do not contain any feasible solutions as before
    - Current partial solution cannot lead to a feasible solution
  - Parts that may contain feasible solutions but cannot contain an optimal solution
    - Current partial solution cannot be extended to an optimal solution
    - Use the value of the solution (objective function applied to the solution) for deciding this



# Bounding Function

- Consider a maximization problem with solution structure  $\langle x_1, x_2, x_3, \dots, x_n \rangle$
- Suppose you are at a node  $p$  at Level  $i$ 
  - So choices for  $x_1, x_2, x_3, \dots, x_i$  has been made
- Suppose you already know of a lower bound  $v_{lower}$  on the value that can be achieved by an optimal solution
- Suppose you can also find an upper bound  $v_{upper}(i+1)$  on the increment in value that can be achieved by extending the partial solution  $s_i$  with choices for  $x_{i+1}$  to  $x_n$

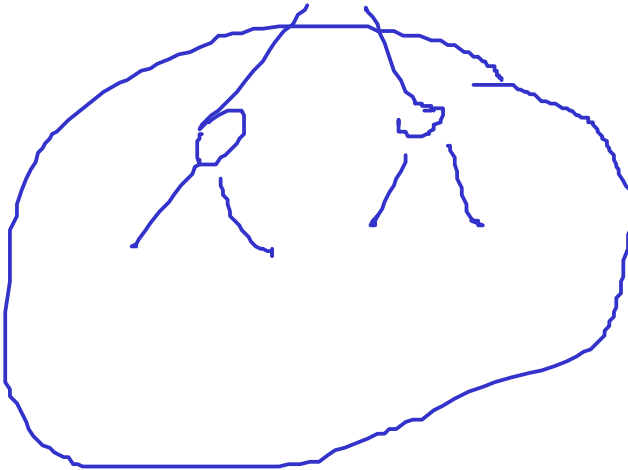
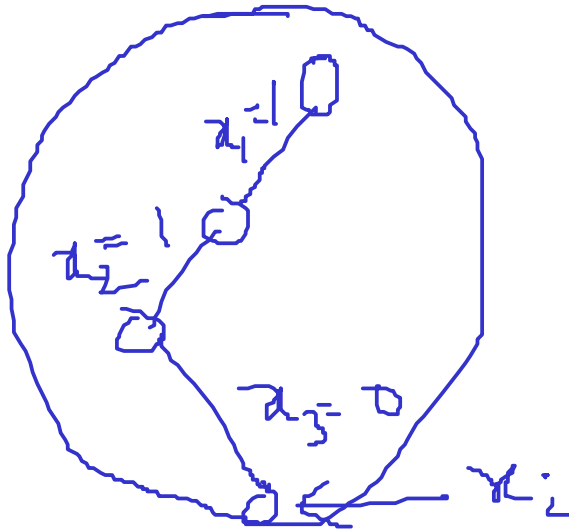


- Then, do not generate the subtree at  $p$  (kill  $p$ ) if

$$F(s_i) + v_{upper}(i+1) < v_{lower}$$

- Even if you extend the current partial solution by making the remaining choices in the best possible manner to add maximum value, it still cannot beat the current best that you know, so definitely cannot lead to an optimal solution
- Can be used to prune the state space tree further
  - In addition to removing parts with no feasible solutions
- But how do we find  $v_{lower}$  and  $v_{upper}(i+1)$  ?

$f(S_i)$   
= value  
of partial  
sol<sup>n</sup>



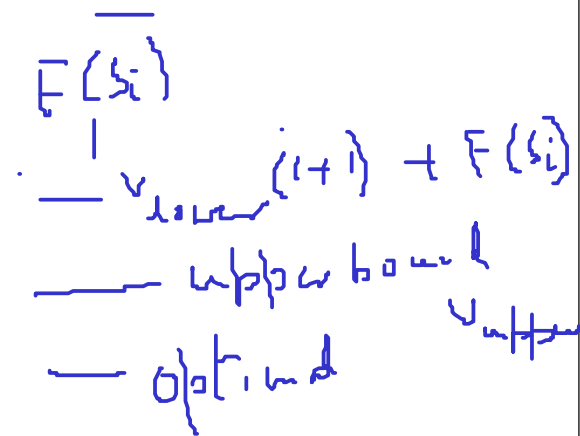
max increment  $\leq \frac{v_{upper}}{v_{lower}(i+1)}$

~~100~~ don't know  
 $> 50$  — optimal  
 — lower bound

$F(S_i)$   
 $\leq v_{upper}$   
 $\uparrow 20$   
 $60$  —  $F(S_i)$  | cannot be optimal  
 $i+1$   
 $v_{lower}$

- Note that we have defined this for maximization problems. For minimization problems, the roles of upper and lower bounds get reversed
  - $v_{upper}$  = known upper bound on optimal solution value
  - $v_{lower}(i+1)$  = lower bound on increment in value of partial solution
  - Do not generate subtree (kill node) if

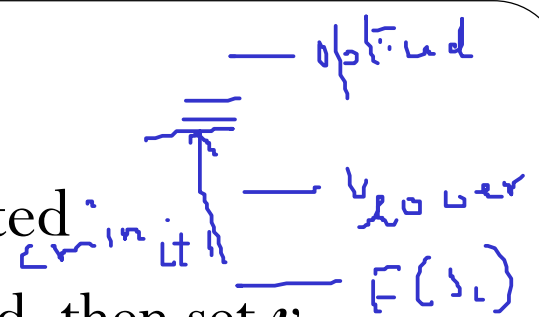
$$F(s_i) + v_{lower}(i+1) > v_{upper}$$



# Finding $v_{lower}$ = Lower bound and optimal

- Set an initial value from a feasible solution found by some method *requirement is that it should be fast*
- Solution found by a polynomial time approximation algorithm if one is known *range of optimal*
- Solution found by some fast heuristic algorithm for the problem if one is known
- Random sampling
- Any other fast method known to find a feasible solution for the problem
- If no other method can be found, set to 0 *(assuming  $F(s) \geq 0$  for all  $s$ )*

- Update dynamically as the tree is generated



- If a partial/full feasible solution  $s_i$  is found, then set  $v_{lower} = \max(v_{lower}, F(s_i) + cmin_{i+1})$  where  $cmin_{i+1}$  is the minimum increment in the value of the objective function that can be achieved by extending the solution with choices of  $x_{i+1}$  to  $x_n$

Lower bound  
on increment

- Note that the partial solution must be feasible
- Example: 0-1 Knapsack



- $F(s_i) = \sum_{1 \leq k \leq i} x_k v_k$
- $cmin_{i+1} = 0$  (can extend the solution at least by not choosing any other item, will still stay feasible)

Exhibit 14

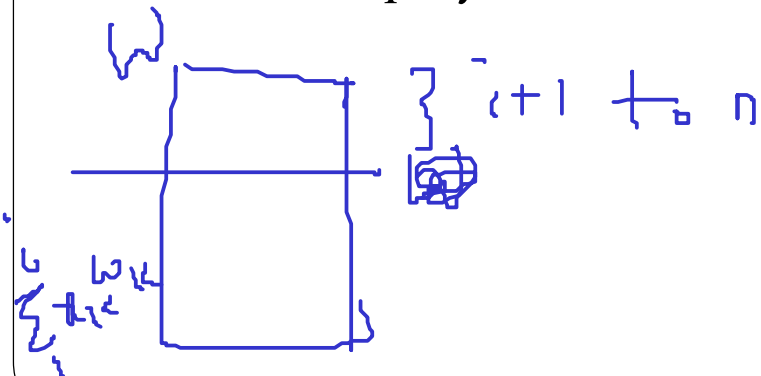
- Can find better estimates at the cost of doing more work

min value  
of an  
item for  $i+1$   
such that  
it will fit in  
the remaining  
capacity

Finding  $v_{upper}(i+1) =$  upper bound on increment in value of  $F(S_i)$

- Can be found by different means
  - Example: 0-1 Knapsack
    - Solve the fractional knapsack problem on the remaining items ( $i+1$  to  $n$ ) with the remaining Knapsack capacity  $(W - \sum_{1 \leq k \leq i} x_k w_k)$ 

*capacity used by  $S_i$*
    - You cannot get more value from the remaining items than this, as fractional knapsack can be solved optimally (in polynomial time), so this is an upper bound



*upper bound = 10*  
*we achieve*  
*max 5*

# Example: Travelling Salesman Problem

- Solution structure  $\langle x_1, x_2, \dots, x_n \rangle$ 
  - $x_i \in \{1, 2, \dots, n\}$  gives the  $i$ -th vertex visited in the cycle
- It does not matter where you start, so choose  $x_1 = 1$
- $v_{upper} = \infty$  (or set from solution of known approximation/heuristic algorithms for TSP)
- $F(s_i) = w(1, x_2) + \sum_{2 \leq k < i} w(x_k, x_{k+1})$
- $v_{lower}(i+1) = (n - i + 1)w_{min}$  where  $w_{min}$  is the minimum weight of any edge in the graph

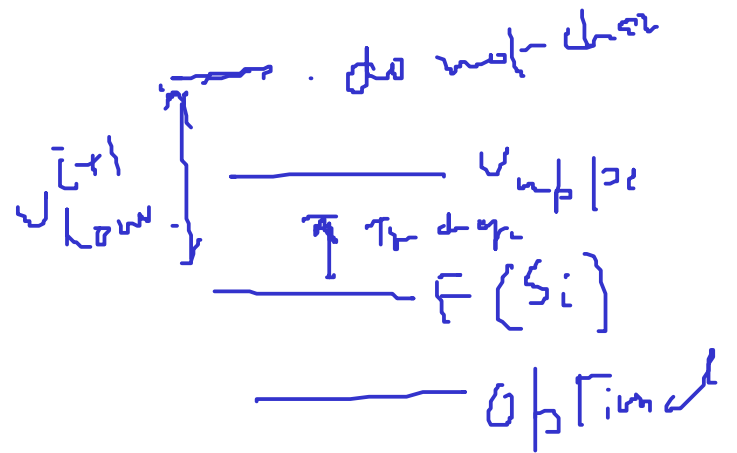
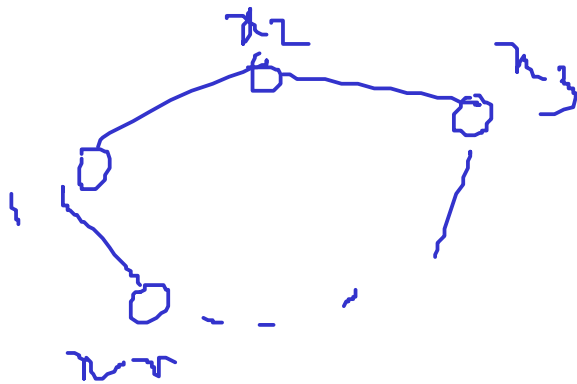
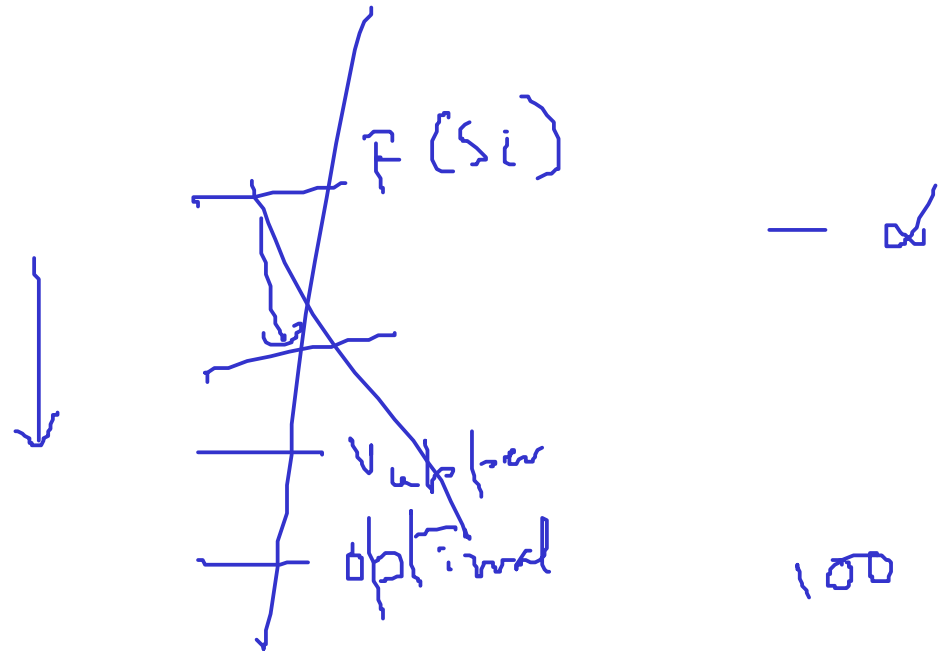
- Bounding Function:

$$x_i \in \{1, x_2, x_3, \dots, x_{i-1}\} \text{ OR}$$

$$F(s_i) + v_{lower}(i+1) > v_{upper}$$

no. of edges yet to be chosen after  $\pi_1 \dots \pi_i$  is chosen



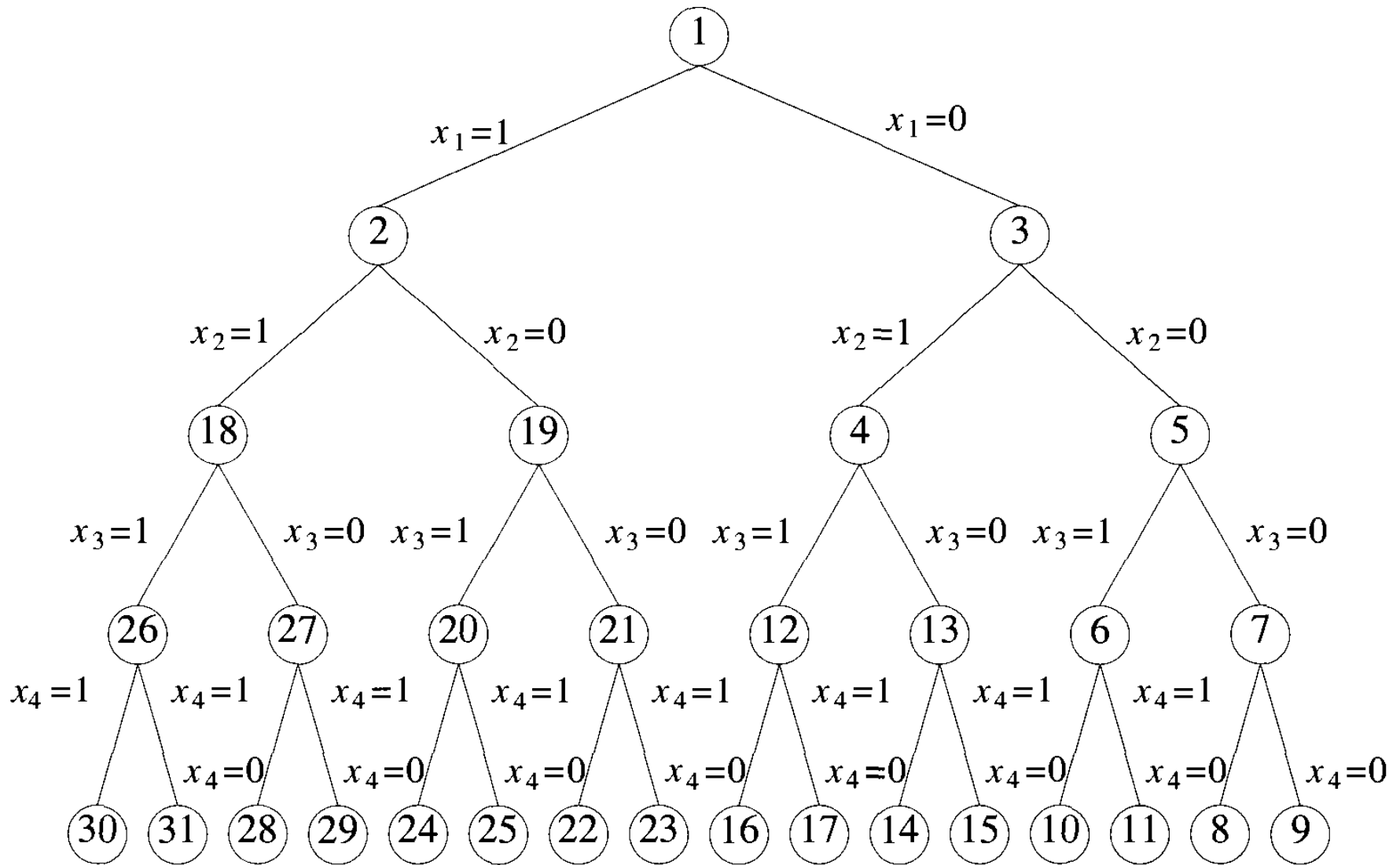


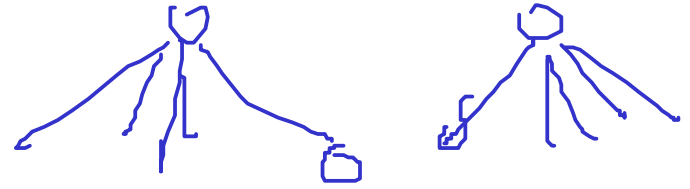
# Tree Generation Orders

- DFS order (mostly used in backtracking)
- FIFO Branch and Bound/BFS
  - Same as BFS. A E-node node remains an E-node until all its children are either generated or killed (due to pruning)
  - Implemented the same way as BFS with a queue. Node at head of queue is the next E-node
- LIFO Branch and Bound/D-Search
  - Similar to BFS in that all nodes of an E-node are generated first
  - Different from BFS in that the generated nodes are placed in a stack instead of in a queue



# Tree of 0-1 Knapsack with LIFO B&B





- Least Cost (LC) Search

- Ranks the live nodes according to some ranking function
- Chooses the one with the highest value *to expand next*
- Implemented with priority queue of live nodes
- Ranking function can be based on different things
  - Current value of the objective function on the partial solution
  - Estimate of the objective function value achievable by generating the subtree rooted at that node
  - Estimate of effort needed to find an optimal solution in the subtree
  - Different ways to estimate and use these, we will not do here

# Summary

- Branch and Bound can solve many hard optimization problems very efficiently for most instances
- Design of good bounding functions/ranking functions is very important
- However, the worst case time complexity may still not be polynomial

