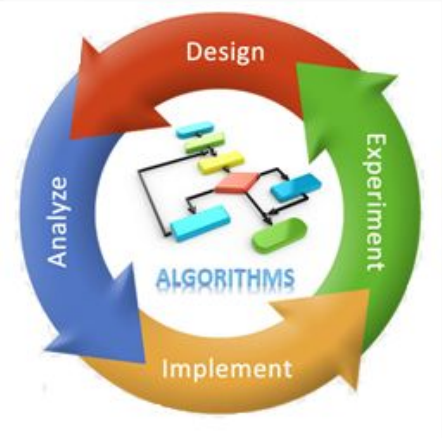
Analysis and Design of Algorithms
(ADA)
GTU # 3150703

Unit-7: Backtracking and **Branch & Bound**



Dr. Gopi Sanghani
Computer Engineering Department
Darshan Institute of Engineering & Technology, Rajkot
gopi.sanghani@darshan.ac.in
9825621471

Outline

- Backtracking
- The N - queens problem
- Branch & Bound
- Knapsack problem
- Travelling Salesman problem
- Minimax principle

Backtracking

Introduction

- Backtracking can be defined as a general algorithmic technique that considers **searching every possible combination** in order to solve an optimization problem.
- It is a recursive technique.
- It generates a state space tree for all possible solutions.
- It traverse the state space tree in the depth first order.
- So, in a backtracking we attempt solving a sub-problem, and if we don't reach the desired solution, then undo whatever we did for solving that sub-problem, and try solving another sub-problem.
- All the solutions require a set of constraints divided into two categories: explicit and implicit constraints.

The N - Queen Problem

- The *N* - queen is the problem of placing N chess queens on an $N \times N$ chessboard so that, no two queens attack each other.
- Two queens of same row, same column or the same diagonal can attack each other.
- K-Promising solution: A solution is called k-promising if it arranges the k - queens in such a way that, they can not threat each other.

Q	

1 - Promising Solution

Q	
Q	

0 - Promising Solution

Q	
	Q

0 - Promising Solution

0 - Promising Solution

The 4 - Queen Problem

1	2	3	4
Q			

1 - Promising

1	2	3	4
Q			
X	X		
X	X	X	X

2 - Promising

1	2	3	4
		Q	
X	X	X	
X			

4 – Promising **<3, 1, 4, 2>**

	1	2	3	4
1		Q		
2				Q
3	Q			
4			Q	

4 – Promising **<2, 4, 1, 3>**

- Above 4-promising solution can be written as **<3, 1, 4, 2>**
- Another possible solution is **<2, 4, 1, 3>**

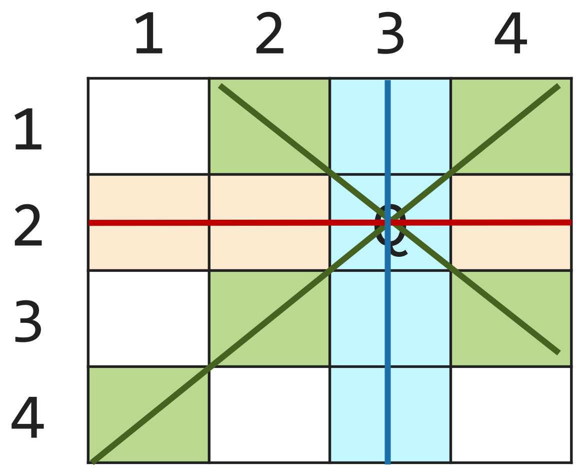
Dedicated Faculty, Committed Education
Darshan
Institute of Engineering & Technology

N - Queen Problem

Number of Queens	Possible Solutions
1	1
2	0
3	0
4	2
5	10
6	4
7	40
8	92
9	352
10	724

N - Queen Problem Solution

- Here, suppose queen position is $(2, 3)$.
- To identify the positions that can not be chosen, so that the queen does not attack.
- The diagonal positions which are under attack are denoted as,



Same Row	Same Col	Same Diagonal	

$Row - Col$

$Row + Col$

Sol = [2,4,1,3]   K = 4

col = {2,4,1}   diag45 = {2,3,-2}

diag135 = {2,5,4}

	1	2	3	4
1		Q		
2				
3				
4				

4- Promising

procedure queens (k, col, diag45, diag135)

{sol[1..k] is k-promising,

col = {sol[i] | 1≤i≤k},

diag45 = {sol[i]–i+1 | 1≤i≤k}, and

diag135 = {sol[i]+i–1 | 1≤i≤k}}

if k = 4 **then**

write sol

else

for j ← 1 **to** 4 **do**   j = 1   j - k = -1   j + k = 3

if j ∉ col **and** j – k ∉ diag45 **and** j + k ∉ diag135

then sol[k+1]□ j

queens(k + 1, col U {j}, diag45 U {j - k}, diag135 U {j + k})

N – Queen Algorithm

sol[1…8] is global array, for all solutions to the eight queens problem call queens (0, ∅, ∅, ∅)

procedure queens (k, col, diag45, diag135)

{sol[1..k] is k-promising,

col = {sol[i] | 1≤i≤k},

diag45 = {sol[i]–i+1 | 1≤i≤k}, and

diag135 = {sol[i]+i–1 | 1≤i≤k}}

if k = 8 **then** {an 8-promising vector is a solution}

write sol

else {explore (k+1)-promising extensions of sol }

for j ← 1 **to** 8 **do**

if j ∉ col **and** j – k ∉ diag45 **and** j + k ∉ diag135 ∉ sol[k+1] ← j

then sol[k+1]□ j

{sol[1..k+1] is (k+1)-promising}

queens(k + 1, col U {j}, diag45 U {j - k}, diag135 U {j + k})

Branch & Bound

Introduction

- The branch & bound approach is based on the principle that the total set of feasible solutions can be partitioned into smaller subsets of solutions.
- These smaller subsets can then be evaluated systematically until the best solution is found.
- Branch & bound is an algorithm design approach which is generally used for solving combinatorial optimization problems.
- These problems are typically exponential in terms of time complexity and may require exploring all possible permutations in worst case.
- The Branch & Bound Algorithm technique solves these problems relatively quickly.

0/1 Knapsack Problem – Introduction

- Let us consider the 0/1 Knapsack problem to understand Branch & Bound.
- The Backtracking Solution can be optimized if we know a bound on best possible solution subtree rooted with every node.
- If the best in subtree is worse than current best, we can simply ignore this node and its subtrees.
- So, we compute bound (the best solution) for every node and compare the bound with current best solution before exploring the node.
- We are given a certain number of **objects** and a **knapsack.**
- Instead of supposing that we have n objects available, we shall suppose that we have **n types of object**, and that an adequate number of objects of each type are available.
- Our aim is to fill the knapsack in a way that **maximizes the value** of the included objects.
- We may take an object or leave behind, but we **may not take fraction** of an object.

0/1 Knapsack using Branch & Bound

- Initially solution is empty.
- Left of the semicolon are weights of selected objects.
- Right of the semicolon is the current total value of load.

Weight of

Corresponding

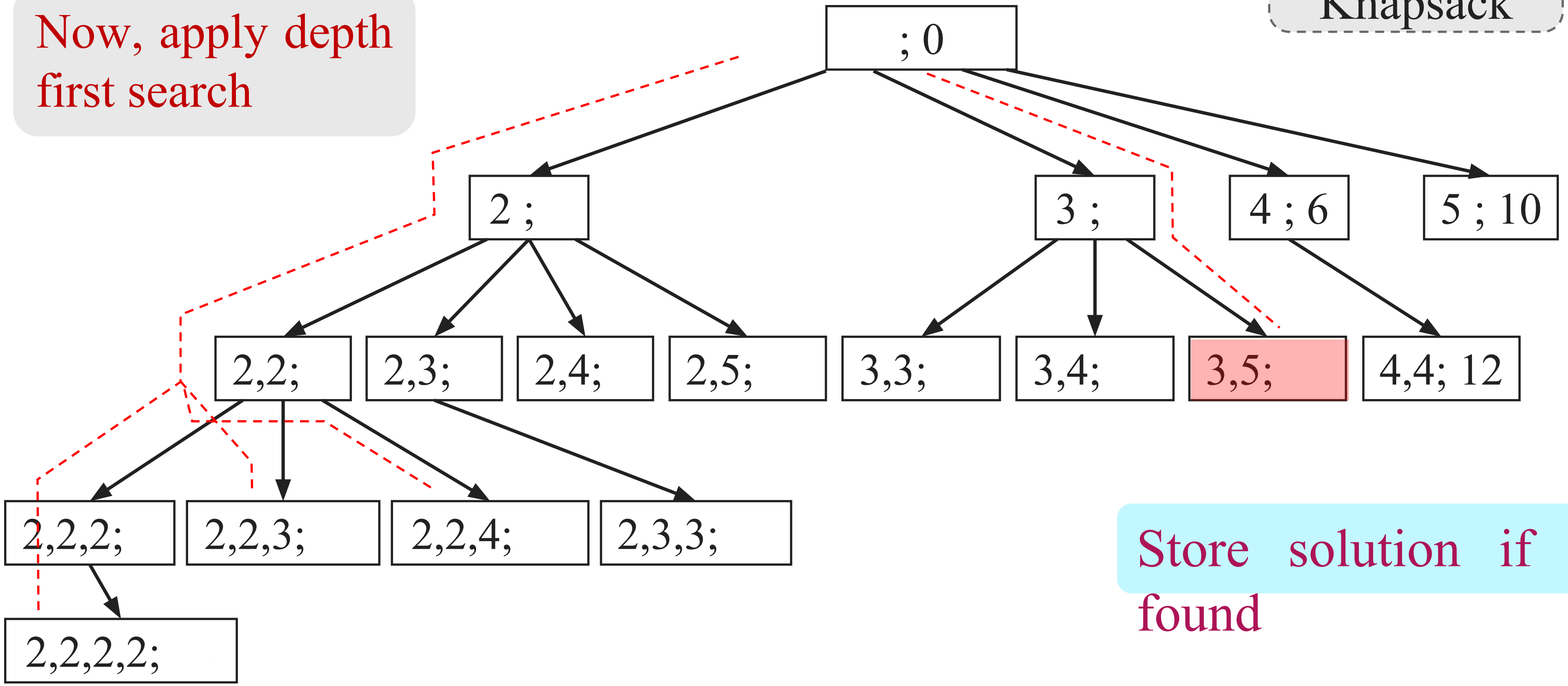
Capacity of Knapsack

w = (2,3,4,5)

v = (3,5,6,10)

W = 8

Now, apply depth first search



Store solution if optimal solution is found

0/1 Knapsack Problem – Algorithm

```
function backpack(i, r)
      {Calculates the value of the best load that can be constructed
      using items of type i to n and whose total weight does not
      exceed r}
    b ← 0
    {Try each allowed kind of item in turn}
    for k ← i to n do
       if w[k] ≤ r then
          b ← max(b, v[k] + backpack (k, r – w[k]))
    return b
```

Travelling Salesman Problem (TSP) – Introduction

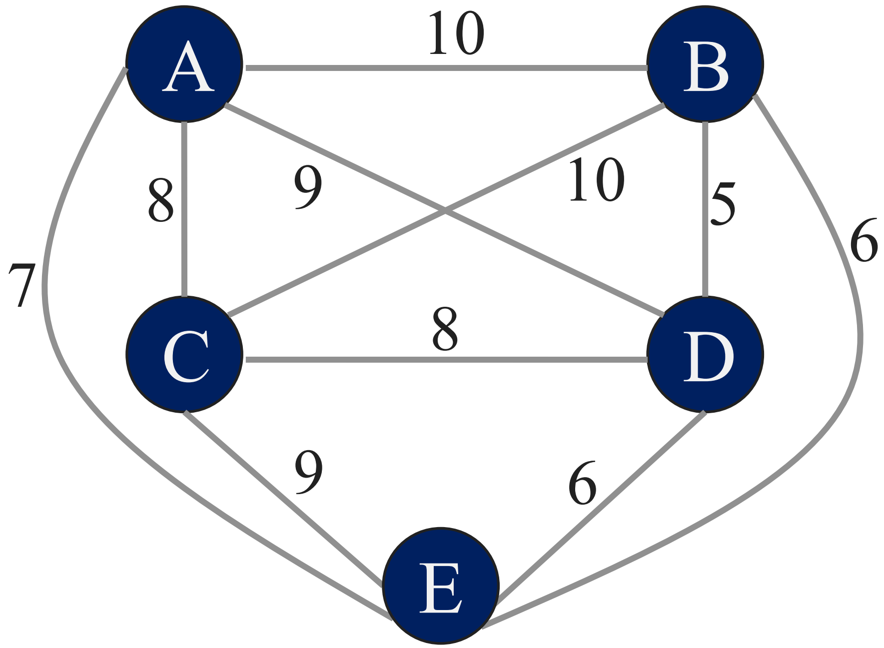
- A traveler needs to visit all the cities from a list, where distances between all the cities are known and each city should be visited just once.
- So, the problem is to find the shortest possible route that visits each city exactly once and returns to the starting point.
- Solution:
  1. Consider city 1 as the starting and ending point.
  2. Generate all (n-1)! Permutations of cities.
  3. Calculate cost of every permutation and keep track of minimum cost permutation.
  4. Return the permutation with minimum cost.
- Time Complexity is $\Theta(n!)$

Travelling Salesman Problem (TSP) – Introduction

- The number of tours grows exponentially as we add cities to the map,

#cities	#tours
5	12
6	60
7	360
8	2,520
9	20,160
10	181,440

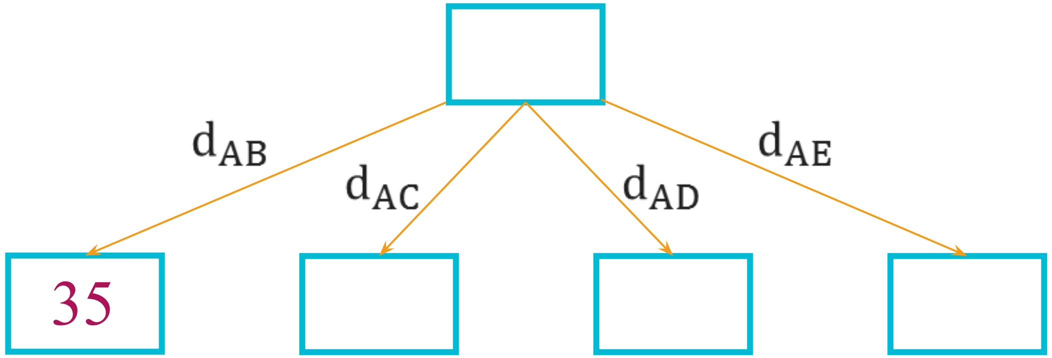
TSP using Branch & Bound



	A	B	C	D	E
A	--	10	8	9	7
B	10	--	10	5	6
C	8	10	--	8	9
D	9	5	8	--	6
E	7	6	9	6	--

- Here, total minimum distance = sum of row/column minimum 31
- The upper bound = A →B→C→D→E→A =
- Solution : [31…41]

TSP using Branch & Bound



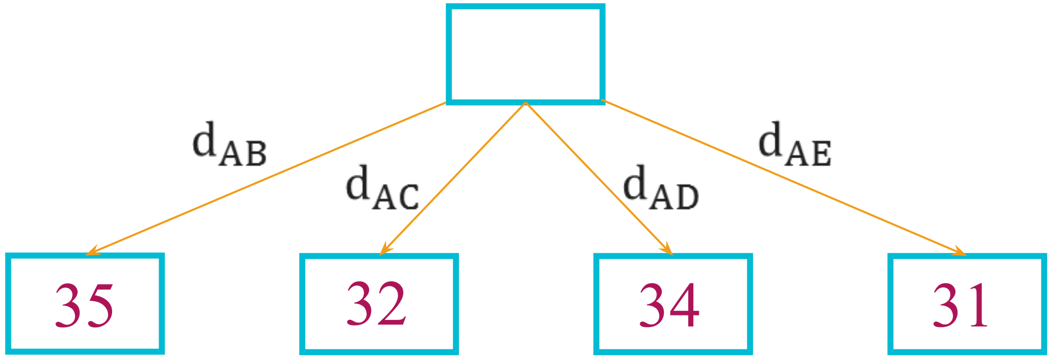
	A	B	C	D	E
A	--	10	8	9	7
B	10	--	10	5	6
C	8	10	--	8	9
D	9	5	8	--	6
E	7	6	9	6	--

	A	C	D	E
B	--	10	5	6
C	8	--	8	9
D	9	8	--	6
E	7	9	6	--

$$d_{AB} = 10 + 5 + 8 + 6 + 6 = 35$$

Distance from A to B

TSP using Branch & Bound



	A	B	C	D	E
A	--	10	8	9	7
B	10	--	10	5	6
C	8	10	--	8	9
D	9	5	8	--	6
E	7	6	9	6	--

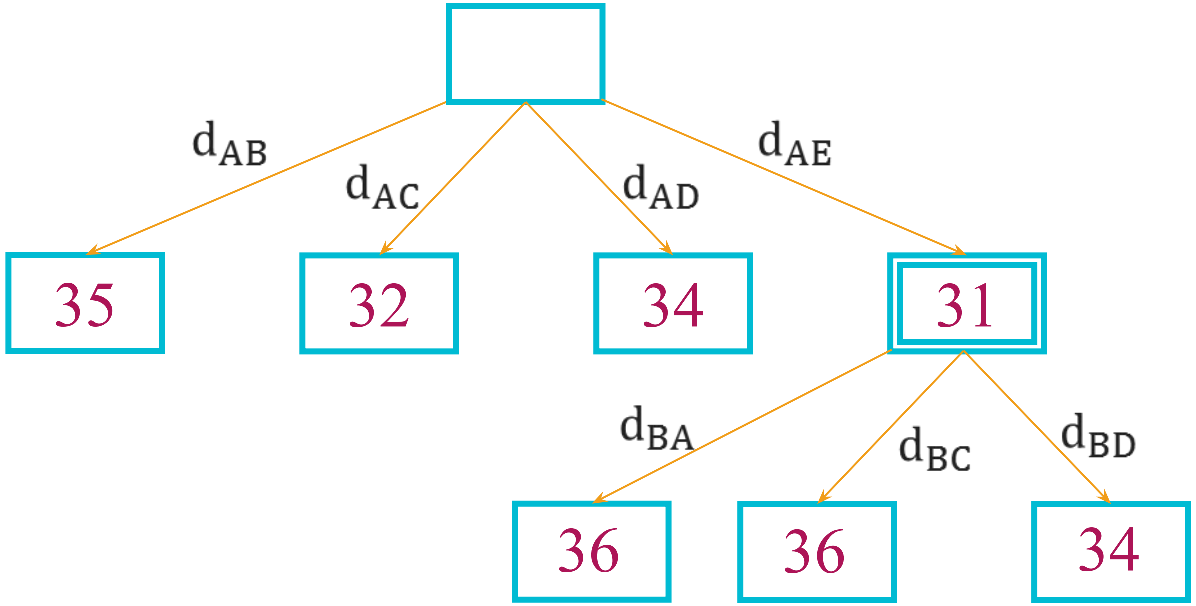
	A	C	D	E
B	10	--	5	6
C	--	10	8	9
D	9	5	--	6
E	7	6	6	--

$$d_{AC} = 8 + 5 + 8 + 5 + 6 = 32$$

Distance from A to C

Dedicated Faculty, Committed Education
Darshan
Institute of Engineering & Technology

TSP using Branch & Bound



	A	B	C	D	E
A	--	10	8	9	[illegible]
B	10	--	10	5	6
C	8	10	--	8	9
D	9	5	8	--	6
E	7	6	9	6	--

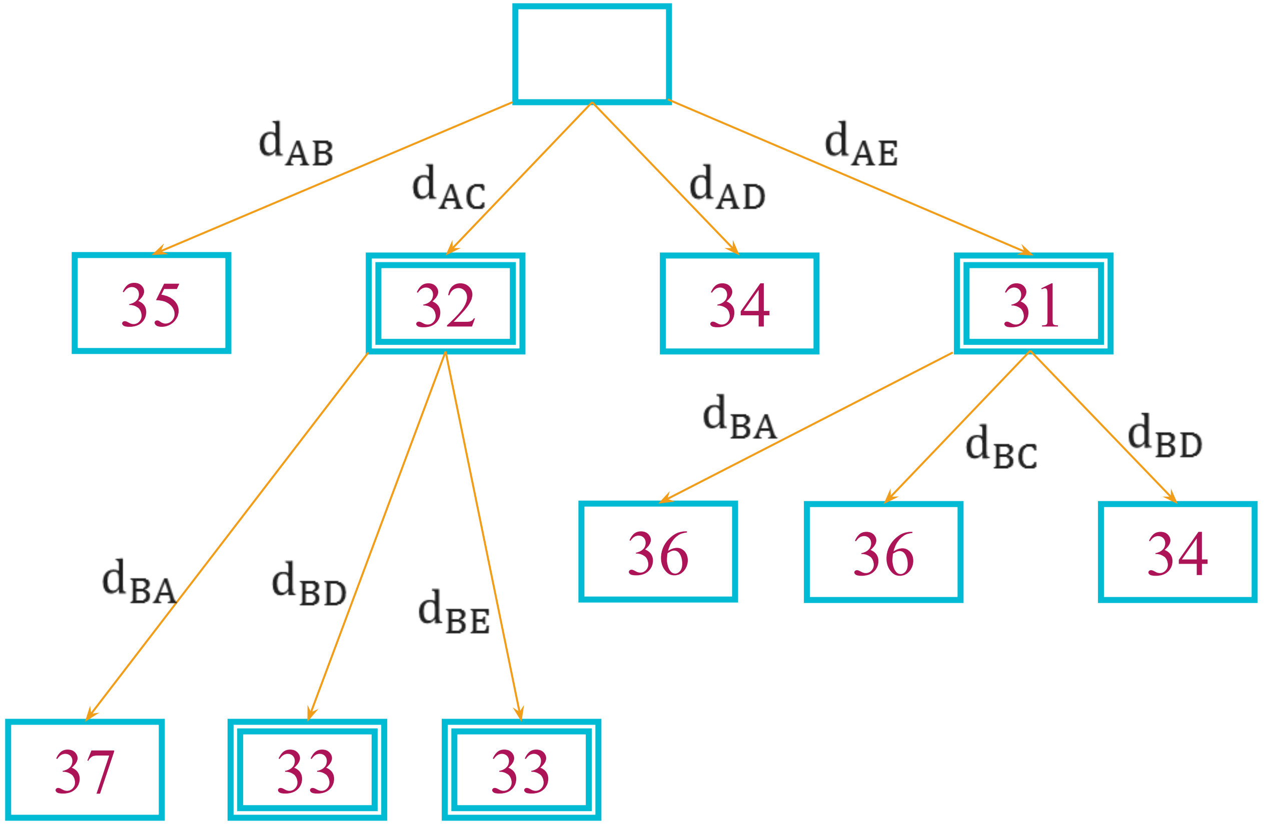
	A	B	D
C	8	--	8
D	9	5	--
E	7	6	6

For d_{AE} and $d_{BC} = 7 + 10 + 8 + 5 + 6 = 36$

Dist from E

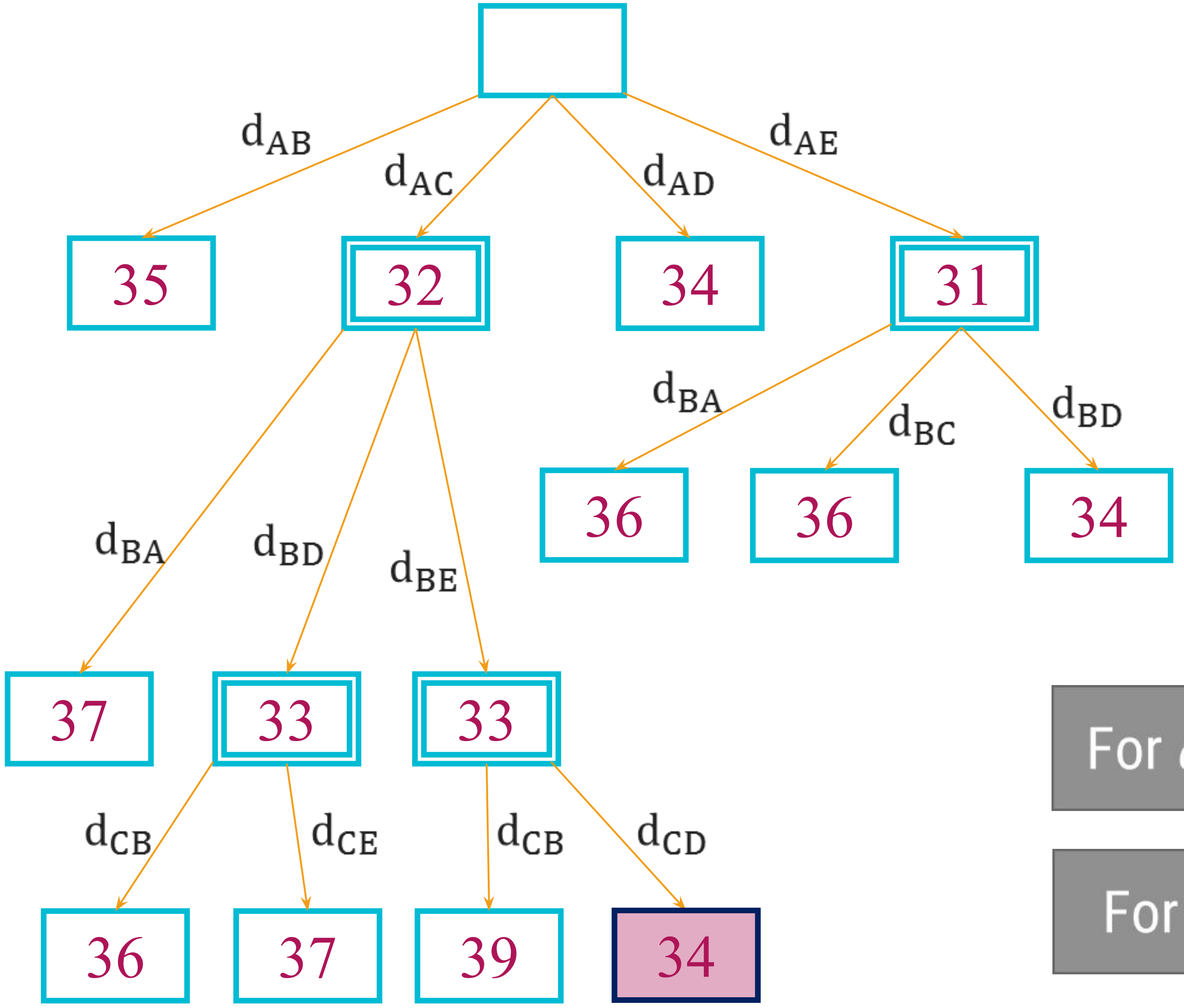
Distance from B to C

TSP using Branch & Bound



	A	B	C	D	E
A	--	10	8	9	7
B	10	--	10	5	6
C	8	10	--	8	9
D	9	5	8	--	6
E	7	6	9	6	--

TSP using Branch & Bound



	A	B	C	D	E
A	--	10	8	9	7
B	10	--	10	5	6
C	8	10	--	8	9
D	9	5	8	--	6
E	7	6	9	6	--

For $d_{AC} + d_{BE} + d_{CB} = 8 + 6 + 10 + 9 + 6 = 39$

For $d_{AC} + d_{BE} + d_{CD} = 8 + 6 + 8 + 5 + 7 = 34$

The optimal route is A – C – D – B – E – A with total cost = 34

Difference between Branch & Bound and Backtracking

Branch & Bound	Backtracking
Branch-and-Bound is used to solve optimization problems.	Backtracking is a general algorithm for finding all or some solutions to the computational problems.
A branch-and-bound algorithm consists of a systematic enumeration of candidate solutions. The set of candidate solutions is thought of as forming a rooted tree, the algorithm explores branches of this tree, which represent the subsets of the solution set.	It incrementally builds candidates to the solutions, and backtracks as soon as it determines that the candidate cannot possibly be completed to a valid solution.
Branch-and-Bound traverse the tree in any manner, DFS or BFS.	It traverses the state space tree by DFS(Depth First Search) manner.

Dedicated Faculty, Committed Education
Darshan Institute of Engineering & Technology

Difference between Branch & Bound and Backtracking

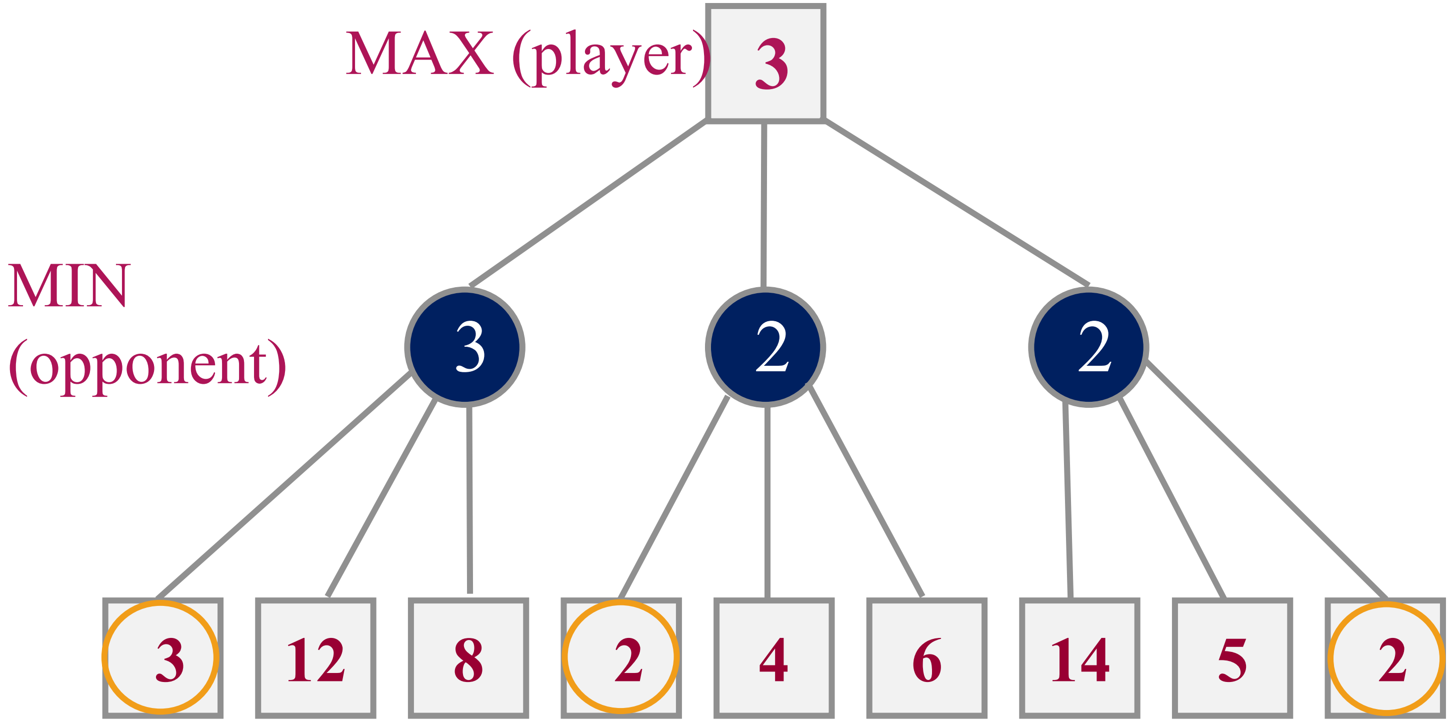
Branch & Bound	Backtracking
Before enumerating the candidate solutions of a branch, the branch is checked against upper and lower estimated bounds on the optimal solution and is discarded if it cannot produce a better solution than the best one found so far by the algorithm.	It is an algorithmic-technique for solving problems using recursive approach by trying to build a solution incrementally, one piece at a time, removing those solutions that fail to satisfy the constraints of the problem at any point of time.
Branch-and-Bound involves a bounding function.	Backtracking involves feasibility function.
Branch-and-Bound is less efficient.	Backtracking is more efficient.

Dedicated Faculty, Committed Education
Darshan
Institute of Engineering & Technology

MiniMax principle

Minimax – Example

- Given a given game tree, the optimal strategy can be determined from the minimax value of each node.
- MAX prefers to move to a state of maximum value, whereas MIN prefers a state of minimum value.



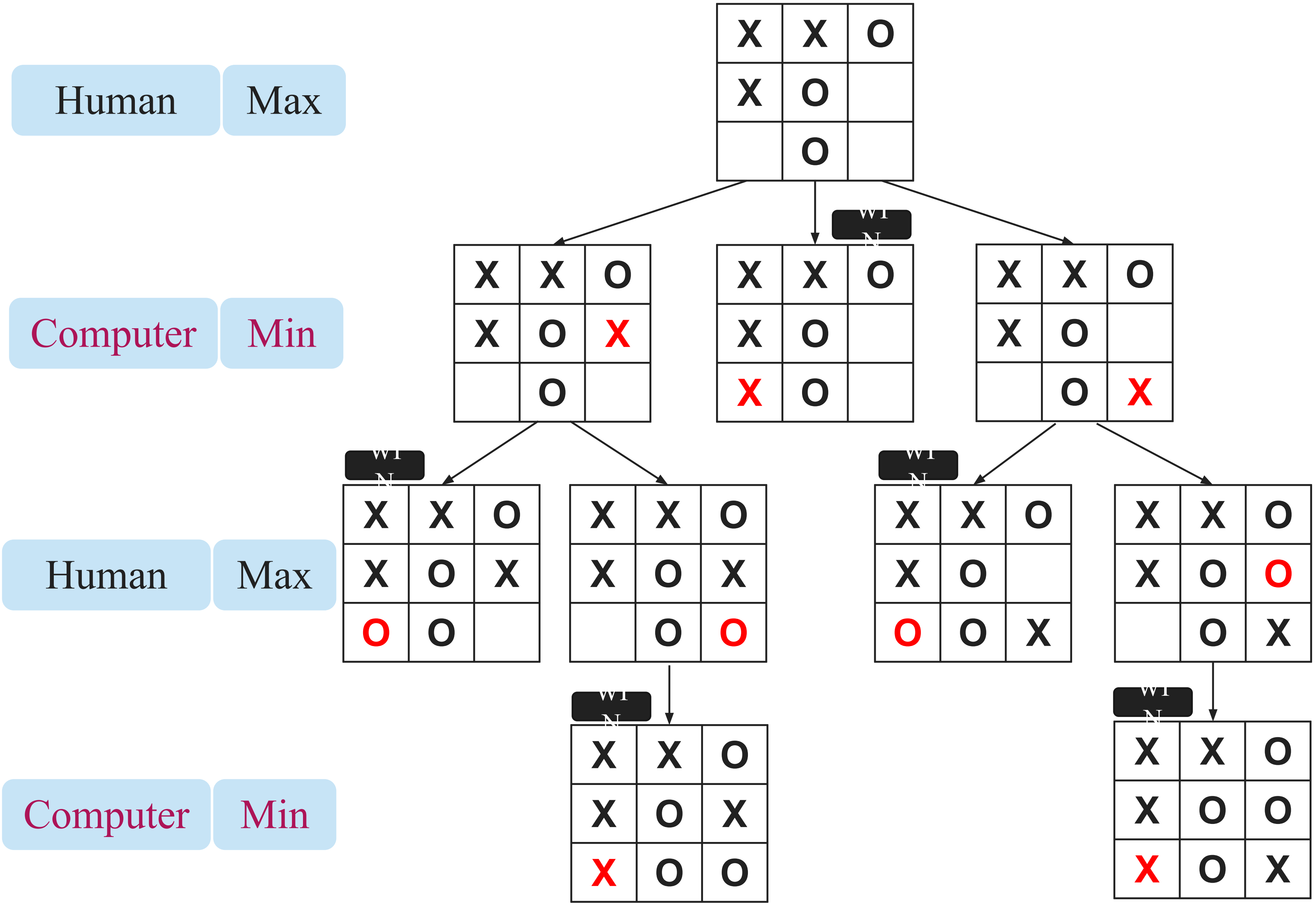
Dedicated Faculty, Committed Education
Darshan
Institute of Engineering & Technology

Minimax - Introduction

- The minimax value of a node is the utility (for MAX) of being in the corresponding state, assuming that both players play optimally from there to the end of the game.
- The key to the Minimax algorithm is a back and forth between the two players, where the player whose "turn it is" desires to select the move with the maximum score.
- In turn, the scores for each of the available moves are determined by the opposing player deciding which of its available moves has the minimum score.
- It uses a simple recursive computation of the minimax values of each successor state.
- The recursion proceeds all the way down to the leaves of the tree, and then the minimax values are backed up through the tree as the recursion unwinds.

Minimax Algorithm in Game Theory – Tic Tac Toe

- **Tic-tac-toe** is a paper-and-pencil game for two players, X and O, who take turns marking the spaces in a 3×3 grid.
- The player who succeeds in placing three of their marks in a horizontal, vertical, or diagonal row wins the game.
- Minimax is a recursive algorithm which is used to choose an optimal move for a player assuming that the opponent is also playing optimally.
- As its name suggests, its goal is to minimize the maximum loss (minimize the worst case scenario).
- To check whether or not the current move is better than the best move we take the help of **minimax()** function which will consider all the possible ways the game can go and returns the best value for that move, assuming the opponent also plays optimally.


Human
Max
Computer
Min
Human
Max
Computer
Min

Thank You!