DAA Theory Notes — Unit IV

Backtracking and Branch & Bound

PART A — BACKTRACKING

---

1. Backtracking — Introduction and Basic Concepts

1. Backtracking is a systematic algorithm design technique that builds a solution **step by step**, and as soon as it determines that the current partial solution cannot lead to a valid or optimal complete solution, it **abandons that path and goes back** to try another option.
2. It is essentially a **Depth First Search (DFS)** on the state space tree, with pruning to avoid exploring dead ends and save time.
3. A **Live Node** is a node that has been generated but whose children have not yet been fully generated — it is still a candidate for further exploration.
4. An **E-node (Expanded node)** is the live node whose children are currently being generated — it is the node being actively processed at any point.
5. A **Dead Node** is a node that will not be expanded further — either because it violates a constraint (infeasible), or all of its children have already been generated and explored.
6. A **Bounding Function** is a condition used to kill live nodes without generating all their children — if the function determines that no valid answer can come from a node, that entire subtree is pruned.
7. Backtracking is best suited for problems where we need to find **all solutions** or any **one valid solution**, such as N-Queens, Hamiltonian Cycle, and 0/1 Knapsack.

---

2. State Space Tree

1. A **state space tree** is a rooted tree that represents all possible states (partial and complete) of a solution — every path from root to a leaf represents one candidate solution.
2. The **root** represents an empty solution with no decisions made yet, nodes at depth 1 represent the first decision, nodes at depth 2 represent the second decision, and so on.
3. A **promising node** is one that may still lead to a valid or optimal solution and should be explored further.
4. A **non-promising node** is one that cannot lead to a valid solution — backtracking prunes this node and does not explore its subtree.
5. Backtracking performs a **DFS on the state space tree**, pruning non-promising nodes to reduce the actual number of nodes visited compared to brute force.

---

3. Recursive Backtracking Algorithm — Control Abstraction

1. The recursive backtracking algorithm is initially invoked as **Backtrack(1)**, where the first k-1 values of the solution vector x[1:n] have already been assigned.
2. **T(x[1], ..., x[k-1])** is the set of all possible values that x[k] can take given the choices already made — it defines the valid candidates at each level.
3. **Bk(x[1], ..., x[k])** is the bounding function — if it returns false, the current partial solution cannot be extended to a valid answer, so this path is pruned immediately.
4. If the bounding function returns true and the current path forms a complete answer, the solution x[1:k] is printed or stored.
5. If k < n (solution not yet complete), Backtrack(k+1) is called recursively to assign the next component of the solution vector.

```
Algorithm Backtrack(k)
{
    for each x[k] in T(x[1], x[2], ..., x[k-1])
    {
        if Bk(x[1], x[2], ..., x[k]) == true
        {
            if (x[1], ..., x[k] is a path to an answer node)
                print x[1:k];
            if (k < n)
                Backtrack(k+1);
        }
    }
}
```

---

4. Iterative Backtracking Method — Control Abstraction

1. The iterative version of backtracking achieves the same result as the recursive version but uses a **while loop** instead of recursion, avoiding function call overhead.
2. It starts with k = 1 and uses a **while(k)** loop — as long as k is valid (greater than 0), the algorithm continues searching.
3. If an untried value exists for x[k] that satisfies both T() and Bk(), it is assigned — if the current assignment forms an answer node, the solution is printed.
4. After printing (or if not yet complete), k is **incremented** to move forward and assign the next position.
5. If no valid value exists for x[k] at the current level, k is **decremented** to backtrack to the previous level and try a different value there.

```
Algorithm IBacktrack(n)
{
    k = 1;
    while (k > 0)
    {
        if (there exists untried x[k] in T(x[1],...,x[k-1])
            and Bk(x[1],...,x[k]) is true)
        {
            if (x[1], ..., x[k] is a path to an answer node)
                print x[1:k];
            k++;
        }
        else
            k--;
    }
}
```

---

5. N-Queens Problem — Theory

1. The N-Queens problem requires placing **n queens on an n×n chessboard** such that no two queens attack each other — no two queens can share the same row, column, or diagonal.
2. The solution vector **x[1:n]** stores the column position of the queen placed in each row — so x[k] = i means the queen in row k is placed in column i.
3. The **Place(k, i)** function checks if placing a queen in row k at column i is safe by verifying two conditions — no previously placed queen is in the same column (x[j] == i), and no previously placed queen is on the same diagonal (|x[j] - i| == |j - k|).
4. Queens are placed **row by row** starting from row 1 — for each row k, columns 1 to n are tried one by one, and only safe positions are explored further.
5. If no valid column exists for row k, the algorithm **backtracks to row k-1** and tries the next column for that row.
6. **Time Complexity: O(n!)** in the worst case, but the bounding function (Place check) prunes a large fraction of these paths in practice.

```
Algorithm NQueen(k, n)
{
    for i = 1 to n
    {
        if Place(k, i) then
        {
            x[k] = i;
            if (k == n) print x[1:n];
            else NQueen(k+1, n);
        }
    }
}

Algorithm Place(k, i)
{
    for j = 1 to k-1
    {
        if (x[j] == i) or (abs(x[j] - i) == abs(j - k))
            return false;
    }
    return true;
}
```

---

6. Hamiltonian Cycle Problem — Theory

1. A **Hamiltonian cycle** is a path in a graph that visits every vertex **exactly once** and returns to the starting vertex — not every graph has a Hamiltonian cycle.
2. The graph is represented using an **adjacency matrix G[1:n][1:n]** where G[i][j] = 1 means there is an edge between vertex i and vertex j, and 0 means no edge.
3. All Hamiltonian cycles start from **vertex 1** — the solution vector x[1:n] stores the sequence of vertices visited in order.
4. The **NextValue(k)** function generates the next legal vertex for position k by checking three conditions — the vertex must be connected by an edge to x[k-1], it must not already appear in x[1:k-1] (distinctness), and if k = n it must also be connected back to x[1] to close the cycle.
5. The **Hamiltonian(k)** algorithm repeatedly calls NextValue(k) to find a valid vertex for position k — if NextValue returns 0 (no valid vertex found), the algorithm backtracks to position k-1.
6. **Time Complexity: O(n!)** in the worst case since there are at most (n-1)! possible Hamiltonian cycles in a complete graph.

```
Algorithm Hamiltonian(k)
{
    repeat
    {
        NextValue(k);
        if (x[k] == 0) return;        // No valid vertex, backtrack
        if (k == n) print x[1:n];     // Complete cycle found
        else Hamiltonian(k+1);
    } until (false);
}

Algorithm NextValue(k)
{
    repeat
    {
        x[k] = (x[k] + 1) mod (n+1);    // Try next vertex
        if (x[k] == 0) return;           // No more vertices to try
        if (G[x[k-1]][x[k]] != 0) then  // Edge exists from previous
        {
            for j = 1 to k-1
                if (x[j] == x[k]) break; // Check distinctness
            if (j == k) then             // Vertex is distinct
                if ((k < n) or ((k == n) and (G[x[n]][x[1]] != 0)))
                    return;              // Valid vertex found
        }
    } until (false);
}
```

---

7. 0/1 Knapsack Problem using Backtracking — Theory

1. Given n items with profits p[i] and weights w[i], and knapsack capacity m, the goal is to **maximize total profit** by selecting a subset of items — each item is either fully included (x[i] = 1) or excluded (x[i] = 0), no fractions.
2. The state space tree is **binary** — at each level k, the left child represents including item k and the right child represents excluding item k, giving 2^n leaf nodes total.
3. The **left child** (include item k) is generated only if adding its weight does not exceed capacity — condition: cw + w[k] ⩽ m, where cw is current weight.
4. The **right child** (exclude item k) is generated only if the **bound function** value is greater than the best profit found so far (fp) — this prunes subtrees that cannot improve on the current best solution.
5. The **bound(cp, cw, k)** function computes an **optimistic upper bound** on the profit achievable from the current node — it adds remaining items greedily (by profit/weight ratio) and allows fractional filling of the last item to get the maximum possible estimate.

6. When a leaf node is reached, if cp > fp, the best solution is updated — **Time Complexity: O(2^n)** in worst case, but bounding makes it much faster in practice.

```
Algorithm Bknap(k, cp, cw)
{
    // Generate left child (include item k)
    if (cw + w[k] <= m)
    {
        y[k] = 1;
        if (k < n) Bknap(k+1, cp+p[k], cw+w[k]);
        if ((cp + p[k] > fp) and (k == n))
        {
            fp = cp + p[k];
            fw = cw + w[k];
            x[1:k] = y[1:k];
        }
    }
    // Generate right child (exclude item k)
    if (bound(cp, cw, k) >= fp)
    {
        y[k] = 0;
        if (k < n) Bknap(k+1, cp, cw);
        if ((cp > fp) and (k == n))
        {
            fp = cp;
            fw = cw;
            x[1:k] = y[1:k];
        }
    }
}

Algorithm bound(cp, cw, k)
{
    b = cp;  c = cw;
    for i = k+1 to n
    {
        c = c + w[i];
        if (c < m) b = b + p[i];
        else return b + (1 - (c - m) / w[i]) * p[i]; // fraction
    }
    return b;
}
```

8. Backtracking — Advantages and Disadvantages

Advantages:

1. Backtracking can find **all possible solutions** to a problem, not just one — useful when all configurations need to be enumerated.
2. It is **guaranteed to find a solution** if one exists because it systematically explores the entire state space tree.
3. The bounding function allows **significant pruning** of the search tree, making it much faster than brute force in practice.
4. Backtracking is **simple to implement** using recursion for a wide variety of constraint satisfaction problems.
5. It works well for problems where the solution space is large but the **number of valid solutions is small** — most branches are pruned early.

Disadvantages:

1. In the worst case, backtracking visits **all 2^n or n! nodes**, making it exponential — not suitable for very large inputs.
2. The efficiency depends heavily on the **quality of the bounding function** — a weak bound leads to little pruning and near-brute-force behavior.
3. Backtracking does **not guarantee finding the optimal solution** unless all branches are explored — it only guarantees any valid solution or all solutions.
4. For optimization problems, backtracking is **less efficient than Branch and Bound**, which is specifically designed to find the globally optimal answer.
5. The **recursive implementation** can lead to deep recursion stacks and stack overflow for very large inputs or deep state space trees.

---

9. Backtracking vs Dynamic Programming

Feature	Backtracking	Dynamic Programming
Approach	Explores state space tree using DFS with pruning	Solves overlapping subproblems bottom-up and stores results in a table
When to use	Finding all valid solutions or any one solution to constraint satisfaction problems	Finding optimal solution to optimization problems with overlapping subproblems
Memory	Low — only stores current path from root to E-node	High — stores entire table of subproblem results
Revisiting	Backtracks and re-explores different branches	Never revisits — each subproblem is computed exactly once

Feature	Backtracking	Dynamic Programming
Optimality	Not guaranteed unless all branches are explored	Always finds the globally optimal solution
Examples	N-Queens, Hamiltonian Cycle, 0/1 Knapsack (backtracking version)	0/1 Knapsack (DP version), LCS, TSP

PART B — BRANCH AND BOUND

---

10. Branch and Bound — Introduction

1. Branch and Bound is a general algorithm design technique for **optimization problems** — it finds the optimal solution (minimum or maximum) by systematically exploring the state space tree with intelligent pruning.
2. Unlike backtracking which uses **DFS**, Branch and Bound can use **BFS (FIFO)** or **best-first search (LC — Least Cost)** to traverse the state space tree.
3. The key idea is to compute a **bound** at each node — if the bound shows that no solution in that subtree can improve on the best solution found so far, the entire subtree is **pruned (killed)**.
4. For **minimization problems**, a lower bound is computed — if the lower bound at a node is already worse (higher) than the best known solution (upper), the node is killed.
5. For **maximization problems**, the objective function is multiplied by -1 to **convert it to minimization**, and the same approach is applied.

---

11. Branch and Bound — Key Terminology

1. **Live Node** — A node that has been generated but whose children have not yet been explored — it is still a candidate for expansion.
2. **E-node (Expanded node)** — The live node currently being expanded — its children are being generated at this moment.
3. **Dead Node** — A node that will not be expanded further — either it is infeasible, it has been pruned by the bounding function, or all its children have been explored.
4. **Cost function C^(x)** — An estimated cost associated with each live node — for minimization problems, it represents the minimum possible cost achievable in the subtree rooted at that node.
5. **Upper bound u(x)** — The best complete solution found so far — any node whose estimated cost C^(x) exceeds the upper bound is killed because it cannot lead to a better solution.

---

12. Control Abstraction for Least Cost (LC) Search

1. LC Search is a Branch and Bound strategy where **the live node with the minimum estimated cost C^(x) is always expanded next**, regardless of when it was generated.
2. It uses a **priority queue** (min-heap) to store live nodes ordered by their C^(x) value — the node with the lowest cost is always at the front.
3. The algorithm starts with the root t as the first E-node — if the root itself is an answer node, the solution is printed immediately.
4. For each expansion, **all children of the current E-node are generated** — each child's C^(x) is computed, and if it is not killed by the bounding condition, it is added to the live node list.
5. The process continues until an answer node is found or the live node list becomes empty — if empty with no answer found, the problem has no solution.

```
Algorithm LCSearch(t)
{
    if (*t is an answer node) { print(*t); return; }
    E = t;
    Initialize list of live nodes to empty;

    while (true)
    {
        for each child x of E
        {
            if (x is an answer node) { print path x to t; return; }
            Add(x);             // Add to live node list with C^(x)
            x->parent = E;      // Store parent pointer for path tracing
        }
        if (list is empty) { print "No answer node"; return; }
        E = Least();            // Select live node with minimum C^(x)
    }
}
```

---

13. Bounding — Job Sequencing with Deadlines Example

1. In job sequencing with deadlines, we have n jobs each with a **penalty pi, deadline di, and processing time ti** — the goal is to select a subset J of jobs that can all finish by their deadlines so that the **total penalty of excluded jobs is minimized**.
2. Jobs not selected for J incur their full penalty — the objective is to minimize the sum of penalties of missed jobs.
3. The **cost function C^(x)** at any node in the state space tree equals the **sum of penalties of all jobs not yet considered** plus the penalties of jobs already decided to be excluded.

4. The **upper bound u(x)** is initialized to infinity and is updated whenever a complete feasible solution (leaf node) is found with a lower total penalty.
5. A node is **killed (pruned)** if its C^(x) is greater than or equal to the current upper bound — that node cannot lead to a better solution than what is already known.
6. **FIFO Branch and Bound** processes live nodes in the order they were generated (queue), while **LC Branch and Bound** always processes the node with the smallest C^(x) first (priority queue).

---

14. FIFO Branch and Bound

1. **FIFO (First In First Out) Branch and Bound** uses a **queue** to store live nodes — nodes are explored in the same order they were generated (level by level, like BFS).
2. When a node is expanded, all its children are generated and added to the back of the queue — the node at the front of the queue is always the next E-node.
3. FIFO is simpler to implement than LC search because it only needs a regular queue, not a priority queue.
4. The bounding function checks at each child whether C^(child) > upper — if yes, the node is killed and not added to the queue.
5. The upper bound is updated whenever a complete solution (leaf node) with a better cost is found — all subsequent nodes with cost ⩾ upper are pruned.

---

15. LC Branch and Bound

1. **LC (Least Cost) Branch and Bound** uses a **priority queue (min-heap)** to always select and expand the live node with the **smallest estimated cost C^(x)** regardless of when it was generated.
2. This is more efficient than FIFO because it **directs the search toward the most promising parts of the tree first**, finding the optimal solution with fewer node expansions in most cases.
3. When a node is expanded in LC-BB, all its children are generated, C^(x) is computed for each, and they are inserted into the priority queue if not killed.
4. The upper bound u(x) is updated whenever a leaf node with a better solution is found — all live nodes with C^(x) ⩾ upper are immediately killed from the priority queue.
5. LC-BB tends to reach the optimal solution much faster than FIFO-BB, but requires more complex implementation due to the priority queue.

---

16. FIFO vs LC Branch and Bound

Feature	FIFO Branch and Bound	LC Branch and Bound
Data structure used	Queue — nodes processed in the order they are generated	Priority queue — nodes processed in order of minimum C^(x)
Node selection	First generated node is expanded next	Node with the smallest estimated cost is expanded next
Search order	Level by level (like BFS)	Best-first — most promising node first
Efficiency	May explore many unpromising nodes before finding the optimal	Reaches optimal solution faster by focusing on lowest-cost paths
Implementation	Simpler — standard queue	More complex — priority queue (min-heap) required
Best suited for	When all nodes have similar cost estimates	When cost estimates vary significantly and are accurate

---

17. 0/1 Knapsack using LC Branch and Bound — Theory

1. For the 0/1 Knapsack problem using LC-BB, the **maximization objective is converted to minimization** by using negative profits — we minimize the negative of the total profit.
2. The **cost function C^(x)** at any node = negative of the maximum profit achievable from that node — it is computed using a greedy bound (adding remaining items by profit/weight ratio, with fractional filling of the last item).
3. The **upper bound u(x)** = negative of the profit of all items fully included up to the current node (no fraction) — this is the actual profit achievable without any future items.
4. A node is killed if its **C^(x) > upper** (since we are minimizing, a higher cost means worse solution) — alternatively stated: if the best possible profit from a node is worse than the best known profit, kill it.
5. The upper bound **upper** is initialized to -∞ (or equivalently, we track the best profit found so far) and is updated whenever a leaf node with a better profit is found — all live nodes with C^(x) ⩾ upper are killed.
6. The algorithm always expands the live node with the **smallest C^(x)** (i.e., the highest estimated profit) until a leaf node gives the optimal complete solution.

---

18. Backtracking vs Branch and Bound

Feature	Backtracking	Branch and Bound
Traversal strategy	Depth-first search (DFS)	BFS (FIFO) or Best-first (LC) depending on variant
Primary purpose	Finding all valid solutions or any one feasible solution	Finding the globally optimal solution to an optimization problem
Pruning basis	Feasibility — prunes paths that violate constraints	Optimality — prunes paths that cannot improve on best known solution
Bounding function	Used to check feasibility of partial solutions	Used to compute estimated best cost and compare with upper bound
Typical problems	N-Queens, Hamiltonian Cycle, 0/1 Knapsack (all solutions)	TSP, 0/1 Knapsack (optimal solution), Job Sequencing
Optimality guarantee	Not guaranteed unless all branches are explored	Always finds the globally optimal solution

---

19. Summary — All Algorithm Complexities (Unit IV)

Algorithm	Time Complexity	Space Complexity
N-Queens (Backtracking)	O(n!) worst case	O(n)
Hamiltonian Cycle (Backtracking)	O(n!) worst case	O(n)
0/1 Knapsack (Backtracking)	O(2^n) worst case	O(n)
LC Search (Branch and Bound)	Depends on bounding quality	O(live nodes)
0/1 Knapsack (LC-BB)	O(2^n) worst case, much better in practice	O(2^n)
Job Sequencing (FIFO/LC-BB)	O(2^n) worst case	O(2^n)

End of Unit IV — Backtracking and Branch & Bound Theory Notes All the best!