

Shri Vile Parle Kelavani Mandal’s

**Institute of Technology, Dhule**

**Deapartment of Information Technology**

Design and Analysis of Algorithms Lab

Name : Nishad Raj Ramsumer

Class : SY(IT)

PRN : 2254491246031

Roll No. : 31

**Write short note on backtracking**

Backtracking is a problem-solving algorithmic technique that involves finding a solution incrementally by trying **different options.** The backtracking algorithm explores various paths to find a sequence path that takes us to the solution. Along these paths, it establishes some small checkpoints from where the problem can backtrack if no feasible solution is found. This process continues until the best solution is found.

Backtracking involves exploring the decision space, which represents all possible choices at each step of the algorithm. Each decision leads to branching in the search tree, creating a tree-like structure where each node corresponds to a decision point. At each decision point, the algorithm considers a set of choices and picks one to proceed. This choice is evaluated against the problem constraints to ensure it leads to a valid solution.

Constraints are conditions that must be satisfied for a solution to be considered valid. Backtracking algorithms incorporate these constraints to guide the exploration process and discard invalid solutions. When a chosen path does not satisfy the constraints or reaches a dead end, the algorithm backtracks to the previous decision point and explores alternative choices. This process continues until a valid solution is found or all possible paths have been explored. Backtracking is often implemented using recursion, with each recursive call representing a decision point in the search process. Recursion simplifies the exploration of the decision space and makes the algorithm easier to implement.

During the backtracking process, the algorithm maintains the state of the solution being constructed. This includes tracking variables, data structures, and other relevant information necessary to make decisions and evaluate solutions. Pruning involves eliminating certain branches of the search tree that cannot lead to a valid solution. Pruning strategies, such as constraint propagation and heuristic pruning, help reduce the search space and improve the efficiency of the algorithm. Backtracking algorithms can be optimized using techniques such as memorization, which stores intermediate results to avoid redundant computation, and constraint relaxation, which temporarily relaxes constraints to explore the solution space more efficiently.

Backtracking algorithms need termination criteria to stop the search when a valid solution is found or when all possible solutions have been explored. Termination criteria prevent the algorithm from running indefinitely and ensure it completes in a finite amount of time.

**Algorithm:**

1. Start at the starting point.
2. For each of the four possible directions (up, down, left, right), try moving in that direction.
3. If moving in that direction leads to the ending point, return the path taken.
4. If moving in that direction does not lead to the ending point, backtrack to the previous position and try a different direction.
5. Repeat steps 2-4 until the ending point is reached or all possible paths have been explored.

Backtracking Algorithm Applications

There are various applications of backtracking algorithms. Some of them are following:

* Maize solve the problem
* N- Queens problem
* Finding all Hamiltonian Paths present in the graph
* Subset Sum problem
* The Knight's Tour problem
* Word break problem

**Complexity:** The time complexity of a backtracking algorithm depends on the size of the state space and the efficiency of the pruning techniques used. In the worst case, backtracking can have exponential time complexity, but efficient pruning can often reduce this to a more manageable level.

**Conclusion:** Implementing a backtracking algorithm typically involves writing a recursive function that explores the decision tree and keeps track of the current state of the solution. You also need to define the problem constraints and the conditions for terminating the search.

Backtracking is a versatile and powerful technique for solving a wide variety of combinatorial problems. Its effectiveness lies in its ability to systematically explore the solution space while avoiding unnecessary computation through pruning.

**Draw and explain P, NP, NP-complete and NP-Hard problem. Write a short note on branch and bound technique**

In complexity theory, a Complexity Class is a set of problems with related complexity. It is the branch of the theory of computation that deals with the resources required to solve a problem.

The common resources are time and space, meaning how much time the algorithm takes to solve a problem and the corresponding memory usage.

The time complexity of an algorithm is used to describe the number of steps required to solve a problem, but it can also be used to describe how long it takes to verify the answer.

The space complexity of an algorithm describes how much memory is required for the algorithm to operate.

**P Class:**

The P in the P class stands for Polynomial Time. It is the collection of decision problems that can be solved by a deterministic machine in polynomial time.

Features:

The solution to P problems is easy to find.

P is often a class of computational problems that are solvable and tractable. Tractable means that the problems can be solved in theory as well as in practice. But the problems that can be solved in theory but not in practice are known as intractable.

The class P consists of those problems that are solvable in polynomial time, i.e. these problems can be solved in time O(nk) in worst-case, where k is constant.

These problems are called tractable, while others are called intractable or superpolynomial.

Formally, an algorithm is polynomial time algorithm, if there exists a polynomial p(n) such that the algorithm can solve any instance of size n in a time O(p(n)).

Problem requiring Ω(n50) time to solve are essentially intractable for large n. Most known polynomial time algorithm run in time O(nk) for fairly low value of k.

The advantages in considering the class of polynomial-time algorithms is that all reasonable deterministic single processor model of computation can be simulated on each other with at most a polynomial slow-d

**NP Class:**

The NP in NP class stands for Non-deterministic Polynomial Time. It is the collection of decision problems that can be solved by a non-deterministic machine in polynomial time.

Features:

The solutions of the NP class are hard to find since they are being solved by a non-deterministic machine but the solutions are easy to verify.

Problems of NP can be verified by a Turing machine in polynomial time.

Example:

Let us consider an example to better understand the NP class. Suppose there is a company having a total of 1000 employees having unique employee IDs. Assume that there are 200 rooms available for them. A selection of 200 employees must be paired together, but the CEO of the company has the data of some employees who can’t work in the same room due to personal reasons.

This is an example of an NP problem. Since it is easy to check if the given choice of 200 employees proposed by a coworker is satisfactory or not i.e. no pair taken from the coworker list appears on the list given by the CEO. But generating such a list from scratch seems to be so hard as to be completely impractical.

It indicates that if someone can provide us with the solution to the problem, we can find the correct and incorrect pair in polynomial time. Thus for the NP class problem, the answer is possible, which can be calculated in polynomial time.

**Co-NP Class:**

Co-NP stands for the complement of NP Class. It means if the answer to a problem in Co-NP is No, then there is proof that can be checked in polynomial time.

Features:

If a problem X is in NP, then its complement X’ is also in CoNP.

For an NP and CoNP problem, there is no need to verify all the answers at once in polynomial time, there is a need to verify only one particular answer “yes” or “no” in polynomial time for a problem to be in NP or CoNP.

Some example problems for CoNP are:

To check prime number.

Integer Factorization.

**Features of NP-Hard:**

All NP-hard problems are not in NP.

It takes a long time to check them. This means if a solution for an NP-hard problem is given, it takes a long time to check whether it is right.

A problem ‘P’ is NP-hard if there is a polynomial-time reduction from ‘Q’ to ‘P’ for every problem ‘Q’ in NP.

Real-World Applications:

Approximate Computing: In approximate computing, the goal is to trade off accuracy for computational efficiency. Many optimization problems in this domain, like optimizing approximation algorithms, can be NP-hard. This implies that finding optimal solutions might be impractical, and researchers need to devise heuristic or approximation methods to get close-to-optimal results.

Cryptography: While most cryptographic algorithms involve polynomial-time operations, certain cryptographic problems, such as solving certain lattice-based problems in post-quantum cryptography, are believed to be NP-hard. This is crucial for ensuring the security of cryptographic systems.

Data Mining: NP-hard problems often arise in data mining tasks like clustering, feature selection, and association rule mining. As data mining involves finding patterns and relationships in large datasets, it’s common to encounter optimization problems that are difficult to solve exactly.

**Branch and Bound:**

Branch and Bound is another algorithmic technique used for optimization problems, particularly in combinatorial optimization. It's a systematic way of exploring the solution space by dividing it into smaller subspaces (branches) and using lower bounds to prune unpromising branches. Here's detailed information about Branch and Bound:

Branch and Bound is used to solve optimization problems, where the goal is to find the best solution among a set of feasible solutions. It systematically explores the solution space by breaking it down into smaller subspaces (branches) and pruning unpromising branches using lower bounds.

Branch and Bound explores the solution space using a tree-like structure. Each node in the tree represents a partial solution or a decision point, and branches from each node represent different choices or options.

At each decision point, the algorithm branches into multiple subproblems by making a choice. These subproblems represent different potential paths in the search space.

Branch and Bound uses bounding techniques to estimate the quality of partial solutions and prune branches that cannot lead to a better solution than the current best solution found so far. This is typically done using lower bounds, which provide a lower limit on the quality of a solution.

Pruning involves eliminating branches of the search tree that are guaranteed not to contain the optimal solution. This reduces the size of the search space and improves the efficiency of the algorithm.

Branch and Bound terminates when either the entire solution space has been explored, or when all remaining branches have been pruned due to lower bounds. Termination criteria ensure that the algorithm terminates in a finite amount of time.

**Complexity Analysis:**

The time and space complexity of Branch and Bound algorithms depend on factors such as the size of the solution space, the efficiency of bounding techniques, and the overhead of backtracking. In general, Branch and Bound algorithms have exponential time complexity, but efficient bounding and pruning strategies can reduce the search space and improve performance.

**Conclusion:**

Branch and Bound is a powerful algorithmic technique for solving optimization problems. By systematically exploring the solution space and using bounding and pruning techniques, Branch and Bound can efficiently find optimal or near-optimal solutions to a wide range of combinatorial optimization problems.