

Module 5 Branch and Bound, Np-Hard and Np-Complete Problems

5.1. Least Cost (LC) search

Least cost (LC) search is an algorithm used in various optimization problems to find the least costly path or solution. It is commonly applied in AI, operations research, and computer science.

Key Concepts

- **Node:** Represents a state or position in the search space.
- **Edge:** Represents a transition between states with an associated cost.
- **Cost:** The value associated with moving from one node to another.
- **Heuristic:** An estimate of the cost to reach the goal from a given node, used in A* search.

Steps in a Least Cost Search Algorithm

1. **Initialize:** Start with the initial node, setting its cost to 0.
2. **Expand:** Explore the node with the lowest cost.
3. **Generate:** Create successors of the current node.
4. **Update Costs:** Calculate the costs for each successor.
5. **Select Next Node:** Choose the next node with the lowest cost.
6. **Repeat:** Continue expanding nodes until the goal is reached.

Example: Uniform Cost Search

1. Initialize the priority queue with the start node and cost 0.
2. While the priority queue is not empty:
 - Remove the node with the lowest cost.
 - If it is the goal, return the path and cost.
 - For each neighbor, calculate the path cost.
 - If the neighbor has not been visited or the new path cost is lower, update the cost and add the neighbor to the queue.

Example: A* Search

1. Initialize the open list with the start node, setting $f(n)$.
2. While the open list is not empty:

- Remove the node with the lowest $f(n)$.
- If it is the goal, return the path and cost.
- For each neighbor, calculate $g(n)$ and $f(n)$.
- If the neighbor has not been visited or the new $f(n)$ is lower, update and add to the open list.

Least cost search algorithms are fundamental in finding optimal solutions in various applications, from route planning to game AI.

5.2. Least Cost with Branch and Bound (LCBB):

One method for solving optimization problems inside the Branch and Bound methodology is the LC (Least Cost) Branch and Bound algorithm. It prioritizes nodes with the lowest cost and uses a heuristic cost function to choose the next node to investigate. Let's use an example to further understand the LC Branch and Bound algorithm.

Look at a sample issue: the traveling salesperson problem (TSP). The task is to determine the shortest trip that will make precisely one stop in each city before returning to the starting place. Assume we know the distances between every pair of cities in our collection.

The following steps are applied using the LC BB algorithm to solve the TSP:

Initialize the search: Start with an empty tour and calculate the initial lower bound based on the current partial tour.

Generate child nodes: Create child nodes by considering every city that could be the next place to visit. Determine the lower bound for every child node by considering the cities that still need to be visited and the current partial tour.

Select the next node: Choose the node with the least cost as the next node to explore. This means selecting the child node with the lowest lower bound.

Update the current tour: Update the current tour by including the city associated with the selected node.

Prune unnecessary branches: Eliminate certain branches based on bounds on the optimal solution. We can prune a branch if its lower bound exceeds the upper bound of the current best solution since it cannot go to the best solution.

Steps 2 to 5 should be repeated until all nodes have been investigated or the termination condition has been satisfied.

Output the optimal solution: Once the algorithm terminates, output the tour with the shortest distance.

By using the LC Branch and Bound algorithm, we can efficiently explore the search space of possible tours and find the optimal solution for the TSP.

It is important to note that the LC Branch and Bound algorithm can be applied to various optimization problems, not just the TSP. The key idea is to use a heuristic cost function to direct the search space exploration and rank the nodes most likely to get better answers.

5.3. FIFO branch and bound:

One method for solving optimization problems inside the Branch and Bound methodology is the FIFO (First-In-First-Out) Branch and Bound algorithm. Using a breadth-first approach, it prioritizes nodes according to the order in which they arrive. Using an example, let us learn about the FIFO Branch and Bound algorithm.

Look at the Knapsack Problem (0/1), where a subset of items must be chosen to maximize the overall value while adhering to the knapsack's capacity restriction. Everything has a value and a weight attached to it.

The FIFO Branch and Bound algorithm can be used to solve the 0/1 Knapsack Problem in the following steps:

Start the search now: Using the partial solution that is now available, begin with an empty knapsack and compute the initial upper bound.

Generate child nodes: Generate child nodes by considering all possible items to include in the next knapsack. Calculate the upper bound for each child node based on the current partial solution and the remaining items.

Select the next node: Choose the node that arrived first in the search as the next node to explore. This means selecting the child node that was generated earliest.

Update the current solution: Update the current solution by including the item associated with the selected node in the knapsack.

Prune unnecessary branches: Eliminate certain branches based on bounds on the optimal solution. We can prune a branch if a node's upper bound is less than the best solution that exists right now because it cannot lead to a better solution.

Steps 2 through 5 should be repeated until all nodes have been investigated or the termination condition has been satisfied.

Output the optimal solution: Once the algorithm terminates, output the knapsack with the maximum value.

The FIFO Branch and Bound technique allows us to quickly search through the space of potential solutions and choose the best one for the 0/1 Knapsack Problem.

It's crucial to remember that the FIFO Branch and Bound technique is not limited to solving the 0/1 Knapsack Problem; it may also be used to solve other optimization issues. The main concept is to analyze nodes according to their arrival sequence and broaden your search space initially.

5.4. 0/1 Knapsack Problem

Problem Statement

You are given a set of items, each with a weight and a value. You need to maximize the total value in your knapsack without exceeding its weight capacity. You cannot take fractions of items; you either take an item or you do not.

Algorithm for finding upper bound:

```

1  Algorithm UBound(cp, cw, k, m)
2  // cp, cw, k, and m have the same meanings as in
3  // Algorithm 7.11. w[i] and p[i] are respectively
4  // the weight and profit of the ith object.
5  {
6      b := cp; c := cw;
7      for i := k + 1 to n do
8          {
9              if (c + w[i] ≤ m) then
10                 {
11                     c := c + w[i]; b := b + p[i];
12                 }
13             }
14      return b;
15  }
```

Why the Greedy Method Does not Work for 0/1 Knapsack Problem

In the 0/1 Knapsack problem, the greedy method does not always yield an optimal solution because choosing items based solely on the value-to-weight ratio may lead to suboptimal selections. For example, it might prioritize a high-ratio item that leaves no room for a combination of smaller items that together have a higher total value.

An example where Greedy Fails:

- **Values:** [60, 100, 120]
- **Weights:** [10, 20, 30]
- **Capacity:** 50
- **Greedy Selection:** First, use the highest value-to-weight ratio to choose the item. This might lead to selecting the first two items (60 and 100) with a total weight of 30 and 20, respectively, totaling 160 in value. However, the optimal solution might be to select the last item (120) and one of the first items, totaling a higher value.

5.5. Traveling salesperson problem:

The Traveling Salesperson Problem (TSP) is a well-known optimization problem in computer science and operations research. It finds the quickest path that passes through several cities and returns to the origin. The TSP issue is challenging because it is NP-hard, which means that solving it grows more computationally expensive as the number of cities rises.

The Branch and Bound algorithm methodically searches for possible solutions to solve optimization problems. It divides the problem into smaller subproblems and filters out certain regions of the search space using boundaries or rules.

To solve the TSP using Branch and Bound, these are the actions we can take:

Initialization: Choose a single city to serve as the starting point.

Generate child nodes: Create child nodes by considering every city that could be the next place to visit. Taking into account the remaining cities and the current partial solution, determine the lower bound for every child node.

Select the next node: Choose The next node to investigate is the one with the lowest lower bound. This entails choosing the child node with the best chance of leading to a shorter path. Update the existing resolution. Add the city connected to the chosen node on the route to the existing solution.

Prune unnecessary branches: Eliminate certain branches based on bounds on the optimal solution. We can prune a branch if a node's lower bound is higher than the best solution that exists right now since it cannot lead to a better solution.

Steps 2 through 5 should be repeated until all nodes have been investigated or the termination condition has been satisfied.

Output the optimal solution: Once the algorithm terminates, output the route with the shortest distance.

We can quickly identify the best route that visits every city and returns to the origin city while minimizing the overall distance traveled by utilizing the Branch and Bound method to explore the search space.

5.6. Nondeterministic Algorithms

Understanding the complexity of computer problems requires the use of nondeterministic techniques. They are especially strongly connected to the ideas of NP-hard and NP-complete issues. Let us take a closer look at these ideas.

NP-Hard Problems

NP-hard problems are a class of problems for which no known polynomial-time algorithms exist. They are at least as hard as the hardest problems in NP (nondeterministic polynomial time). In other words, if a polynomial-time algorithm could be found for an NP-hard problem, then every problem in NP could be solved in polynomial time (which implies $P=NP$, one of the biggest open questions in computer science).

Characteristics of NP-Hard Problems

- **No Polynomial-Time Solution:** No known algorithm can solve NP-hard problems in polynomial time.
- **Verification in Polynomial Time:** A proposed solution can be verified in polynomial time for some NP-hard problems (those that are also in NP).
- **Reduction:** Any problem in NP can be reduced to an NP-hard problem in polynomial time. This implies that solving an NP-hard problem efficiently would enable the efficient solution of all NP problems.

Strategies for Tackling NP-Hard Problems

1. Exact Algorithms:

Brute Force: Try all possible solutions and select the best one. This is usually impractical for large instances due to exponential time complexity.

Dynamic Programming: Break down the problem into simpler subproblems and store the results to avoid redundant computations.

Branch and Bound: Systematically explore the solution space and prune branches that cannot yield better solutions than the current best.

2. Approximation Algorithms:

Provide solutions that are close to the optimal solution within a guaranteed bound.

Useful when an exact solution is impractical due to time constraints.

3. Metaheuristics:

Simulated Annealing: Mimic the annealing process in metallurgy to find a good approximation of the global optimum.

Tabu Search: Use memory structures to avoid cycles and improve local search techniques.

Ant Colony Optimization: Mimic the behavior of ants searching for food to find good solutions.

NP-hard problems are pervasive in various fields, and finding efficient solutions to these problems is crucial for many real-world applications. While exact solutions are often impractical due to their exponential complexity, approximation and heuristic methods provide valuable tools for tackling these challenging problems.

NP-Complete

If a problem is both NP-hard and belongs to the class NP, it is called NP-complete. The hardest issues in NP are thought to be NP-complete problems. All NP-complete problems should be able to be solved in polynomial time if there is a polynomial-time algorithm for it. The fact that these issues are complete in the sense that they represent the toughest problems in NP is reflected in their name, "NP-complete". If we can swiftly solve any NP-complete problem, we can do the same for every other NP problem that can be readily verified.

NP-complete problems are a subset of NP (nondeterministic polynomial time) problems that are both in NP and NP-hard. This means they are as difficult as any problem in NP and can be verified in polynomial time. If any NP-complete problem can be solved in polynomial time, then every problem in NP can be solved in polynomial time, which would imply that $P = NP$.

Characteristics of NP-Complete Problems

1. **Verification in Polynomial Time:** A proposed solution to an NP-complete problem can be verified in polynomial time.
2. **NP-Hardness:** Any problem in NP can be reduced to an NP-complete problem in polynomial time. This means solving an NP-complete problem efficiently would enable solving all NP problems efficiently.
3. **Equivalence:** All NP-complete problems are equivalent in terms of their computational complexity. If a polynomial-time algorithm exists for one NP-complete problem, then polynomial-time algorithms exist for all NP-complete problems.

NP-complete problems represent some of computer science's most challenging and significant problems. They encapsulate the complexity of NP problems and have widespread implications in theoretical and practical domains. Finding efficient solutions to these problems, or proving that such solutions do not exist, remains a central quest in computational complexity.

Relationship between NP-Hard and NP-Complete Problems

1. Inclusion:

All NP-complete problems are NP-Hard, but not all NP-Hard problems are NP-Complete.

NP-complete problems are a subset of NP-Hard problems that are also in NP.

2. Implications:

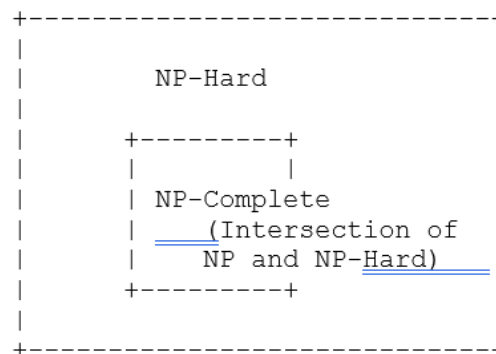
If an NP-complete problem can be solved in polynomial time, then every problem in NP can also be solved in polynomial time. This would mean $P = NP$.

Suppose an NP-Hard problem that is not in NP (e.g., it cannot be verified in polynomial time) can be solved in polynomial time. In that case, it does not directly affect the P vs NP question but would imply significant breakthroughs in understanding computational complexity.

3. Visualization

To visualize the relationship, consider a Venn diagram:

- **NP:** Contains problems that can be verified in polynomial time.
 - **NP-Complete:** Problems that are both in NP and NP-Hard (intersection of NP and NP-Hard).
- **NP-Hard:** Contains problems as hard as the hardest problems in NP, but not necessarily in NP (includes NP-complete problems).



5.7. Cook's theorem:

Cook's theorem is a fundamental result of computational complexity theory, and it was named after Stephen Cook, who proved it in 1971. The theorem states that the Boolean satisfiability problem (SAT) is NP-complete. This was the first problem to be proven as NP-complete, which has significant implications for the field of computer science.

Explanation:

- **Boolean Satisfiability Problem (SAT):** This is the problem of determining if an assignment of truth values exists to variables such that a given Boolean formula is satisfied (i.e., evaluates to true). For example, one possible satisfying assignment for the formula $(A \vee \underline{B}) \wedge (B \vee \underline{C})$ is $A = \text{true}, B = \text{false}, C = \text{false}$.
- **NP (Nondeterministic Polynomial time):** This is a class of problems for which a deterministic Turing machine can verify a proposed solution as correct or incorrect in polynomial time. Informally, these are problems for which a solution can be guessed and then checked quickly.

- **NP-complete:** A problem is NP-complete if it is both in NP and as hard as any problem in NP, meaning that every problem in NP can be reduced to it in polynomial time. This implies that if there is a polynomial-time algorithm for any NP-complete problem, there would be polynomial-time algorithms for all problems in NP.

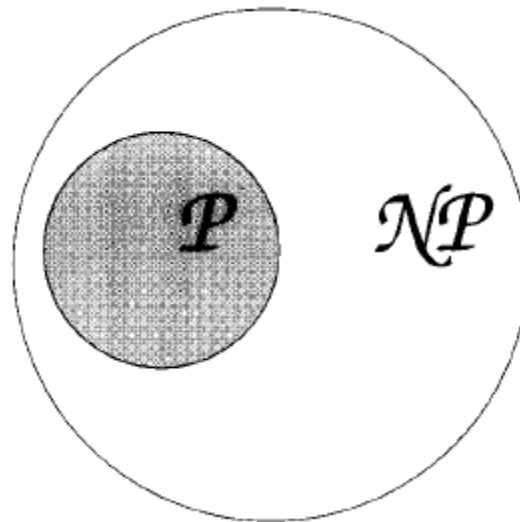


Figure: Relationship between P and NP

Implications of Cook's Theorem:

- Cook's theorem implies that SAT is a universal problem for NP, meaning that all problems in NP can be transformed into SAT.
- If SAT (or any NP-complete problem) can be solved in polynomial time, then all problems in NP can be solved in polynomial time, implying $\mathcal{P} = \mathcal{NP}$. Conversely, if no NP-complete problem can be solved in polynomial time, then $\mathcal{P} \neq \mathcal{NP}$.
- The theorem provided a framework for identifying other NP-complete problems, leading to the development of many important concepts and techniques in theoretical computer science.

Cook's theorem laid the foundation for much of the research in computational complexity theory, including the famous P vs NP problem, which asks whether every problem whose solution can be quickly verified (NP) can also be quickly solved (P). This remains one of the most important open questions in computer science.

5.8. NP-hard scheduling problems – Scheduling identical processors:

It is well known that scheduling issues with identical processors are NP-hard. This indicates that certain problems are computationally difficult to solve optimally, and no effective algorithm can do it in polynomial time.

The minimize the makespan problem illustrates an NP-hard scheduling problem on identical processors. When a series of jobs are scheduled on identical processors, the total time needed to accomplish them is referred to as the makespan. The aim is to find a schedule that reduces the makespan.

Let's look at an example where we have a collection of jobs that need to be scheduled on the same processors in order to demonstrate this. Every activity has a processing time associated with it; the goal is to reduce the total amount of time needed to finish all activities.

It is well known that the makespan minimization problem on identical processors is NP-hard. This indicates that no effective algorithm currently can handle this issue in polynomial time. Instead, we usually use heuristics or approximation methods to identify good solutions that approximate the best.

For instance, the Shortest Processing Time First (SPT) algorithm is a widely used heuristic for task scheduling on identical processors. The SPT algorithm sorts the tasks by processing times and allocates them to the processor with the earliest end time to date. Although the SPT algorithm offers a reasonable approximation for reducing the average completion time, it is still NP-hard to discover the schedule with the least finish time or optimal mean finish time.

In conclusion, scheduling issues involving identical processors, such as decreasing the makespan, are known to be NP-hard. This indicates that it is computationally difficult to identify the best solution to these issues. No known efficient method can handle these problems optimally in polynomial time, although approximation algorithms and heuristics can be applied to find good answers.