Foundations of Algorithm Design and Machine Learning

Date: 10th Feb 2025

Topic: Approximation and Randomized Algorithms

Hard problems refer to computational problems that require significant time and resources to solve optimally, especially as the problem size grows. These problems often fall under NP-hard or NP-complete categories in computational complexity theory.

Why Are They Hard?

- Exponential Growth of Solutions: Many hard problems involve exploring an enormous number of possible solutions. For example, in the Traveling Salesman Problem (TSP) with n cities, there are (n−1)! possible tours, making brute-force search infeasible.
- No Known Polynomial-Time Solution: Unlike problems in P (Polynomial Time Complexity), which can be solved efficiently, NP-hard problems do not have a known polynomial-time algorithm unless P = NP, which is still an open question.

Since exact solutions are often impractical due to computational limits, we use two key techniques to solve these problems:

1. Combinatorial Optimization
2. Approximation Algorithms

Combinatorial Optimization

Combinatorial optimization is a mathematical technique used to find the best solution from a finite set of possibilities.

Key Techniques in Combinatorial Optimization

a. Greedy Algorithms

- Approach: Make a locally optimal choice at each step in hopes of finding a global optimum.
- Example: Minimum Spanning Tree (MST)
    - Kruskal's or Prim's algorithm efficiently finds the MST.

- Limitations: Greedy algorithms don't always work for NP-hard problems, as they may get stuck in local optima.

b. Dynamic Programming

- Approach: Break the problem into smaller overlapping subproblems, solve them once, and store results.
- Example: Knapsack Problem
    - The 0/1 Knapsack Problem can be solved using dynamic programming in O(nW) time, where W is the capacity.
- Limitations: Becomes inefficient for large inputs due to memory constraints.

c. Metaheuristic Approaches

For problems where exact methods fail, metaheuristics provide near-optimal solutions:

- Genetic Algorithms: Evolve better solutions over generations.
- Simulated Annealing: Uses probability to escape local optima.
- Tabu Search: Avoids cycling back to previously visited solutions.

Approximation Algorithms

Approximation algorithms are used to find near-optimal solutions to computationally hard problems (typically NP-hard problems) when finding an exact solution is infeasible. These problems generally fall into two categories:

1. Minimization Problems: The goal is to find the smallest possible value (e.g., minimizing cost, distance, or time).
2. Maximization Problems: The goal is to find the largest possible value (e.g., maximizing profit, efficiency, or coverage).

Approximation Factor (α)

Since approximation algorithms do not always find the optimal solution, we measure their performance using an approximation factor (α). The approximation factor tells us how close the approximate solution is to the optimal one.

1. Minimization Problems and Approximation Factor

In minimization problems, we want to find the smallest possible value. The approximation algorithm produces a solution that is at most α(alpha) times the optimal solution.

For a minimization problem:

- Let OPT be the optimal (minimum) solution.
- Let APPROX be the solution found by an approximation algorithm.
- The algorithm is an α approximation if:

APPROX ≤ α·OPT, where α≥1.

Intuition:

- If α=1, the approximation algorithm gives the exact solution.
- If α=2, the approximation solution is at most twice the optimal solution.
- The closer α(alpha) is to 1, the better the approximation.

Example: Vertex Cover (2-Approximation)

The Vertex Cover Problem (finding the smallest number of vertices to cover all edges in a graph) is NP-hard. A simple greedy algorithm provides a 2-approximation:

- Optimal Vertex Cover (OPT): Smallest possible set.
- Greedy Algorithm Solution (APPROX): At most twice the optimal size.

Thus,

$$APPROX \leq 2 \times OPT$$

2. Maximization Problems and Approximation Factor

In maximization problems, we want to find the largest possible value. The approximation algorithm produces a solution that is at least 1/α times the optimal solution.

For a maximization problem:

- Let OPT be the optimal (maximum) solution.
- Let APPROX be the solution found by an approximation algorithm.
- The algorithm is an α approximation if:

APPROX ≥ α·OPT, where α≥1

Intuition:

- If $\alpha=1$, the approximation algorithm gives the exact solution.
- If $\alpha=2$, the approximation solution is at least half of the optimal.
- The closer α is to 1, the better the approximation.

Example: MAX-SAT (3/4-Approximation)

In the MAX-SAT Problem, where we maximize the number of satisfied clauses in a Boolean formula, a simple randomized algorithm gives a 3/4-approximation:

$$APPROX \geq 3/4 \times OPT$$

This means the algorithm guarantees at least 75% of the optimal number of satisfied clauses.

Example:

Load Balancing on Identical Machines:

We have n independent jobs that need to be scheduled across m identical processors (machines). Each job Ji has a given execution time ti . The goal is to assign jobs to processors such that the total execution time (makespan) is minimized.

Input:

- n jobs: J1,J2,…,Jn
- Each job Ji has an execution time ti
- m identical processors

Approach: Greedy Scheduling Using a Min-Heap

We use the Greedy Min-Heap Algorithm to assign jobs:

1. Initialize a Min-Heap of size m, where each processor has an initial load of 0.
2. Sort jobs in decreasing order by processing time (Largest Processing Time First, LPT).
3. Assign each job to the processor with the least current load (i.e., the root of the min-heap).
4. Update the heap after assigning each job to maintain the order.

This ensures that larger jobs are allocated first, preventing long-tail effects.

Complexity Analysis

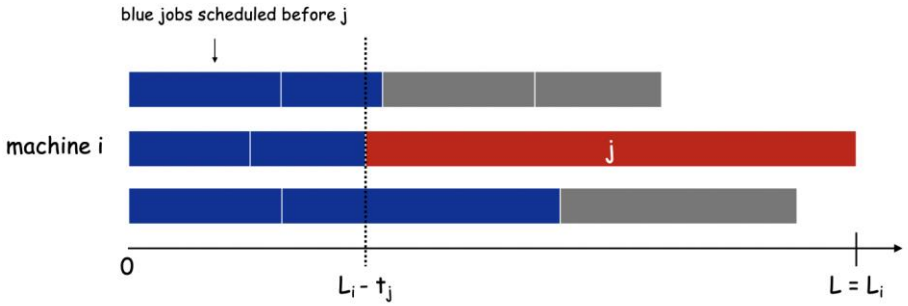
- Sorting Jobs: O(nlogn)
- Maintaining Min-Heap: O(logm) per job insertion.
- Total Complexity: O(n log n)+O(n logm)=O(n log n) since log m is much smaller than log n.

Finding the approximation factor:

Consider load Li of bottleneck machine i.

Let j be last job scheduled on machine i.

When job j assigned to machine i, i had smallest load. Its load before assignment is Li – tj, Li - tj <= Lk for all 1 <= k <= m.



The average load across all machines is at most L∗, leading to the inequality:

$$L_i - t_j \leq \frac{1}{m} \sum_k L_k = \frac{1}{m} \sum_k t_k$$

Average load across all machines provides a lower bound on the optimal makespan L∗.

$$L_i - t_j \leq L^*$$

Adding tj to account for the new job j:

$$L_i = (L_i - t_j) + t_j \leq 2L^*$$

This gives an upper bound of 2L∗ for the makespan.

RANDOMIZATION:

Randomization is a technique used in computer science and mathematics to introduce randomness into algorithms to make them more efficient, robust, or practical. It is especially useful when dealing with large datasets, complex computations, or uncertain conditions. Instead of deterministically computing results, a randomized approach makes decisions based on random choices, which often leads to faster and approximate solutions.

Example:

To figure out whether the relationship p(x).q(x) = r(x) holds?

$$p(x) = p_0 + p_1 x + p_2 x^2 + \cdots + p_{n-1} x^{n-1}$$

$$q(x) = q_0 + q_1 x + q_2 x^2 + \cdots + q_{n-1} x^{n-1}$$

$$r(x) = r_0 + r_1 x + r_2 x^2 + \cdots + r_{2n-2} x^{2n-2}$$

Steps to Verify the Expression More Efficiently:

Choosing a Test Value

- Instead of checking the equation for every possible value of xxx, a single value is picked from a predefined set, such as {1, 2, ..., 1000n-1}.
- This value is then substituted into the equation to check if the left-hand side equals the right-hand side.

Possibility of an Incorrect Conclusion

- There might be some instances where the test value satisfies the equation even if the equation is not universally true.
- The probability of selecting such a misleading value can be calculated.

The given probability of making an incorrect conclusion is:

$$P(\text{incorrect conclusion}) = \frac{2n-2}{1000n}$$

This means that the likelihood of making a mistake using this approach is relatively low (around 0.002 or 0.2%). Here, (2n - 2) represents the number of values of xxx for which the equation

falsely appears to be correct. The denominator **1000n** represents the total possible values of x that could have been chosen.

Generalization to ML algorithms:

In ML, algorithms approximate and generalize solutions. There is always a small probability that an algorithm makes a wrong prediction, similar to how a randomized algorithm may sometimes give an incorrect result. The goal in ML (just like in randomized algorithms) is to minimize errors while making computations efficient.