Indian Institute of Technology Madras

Department of Data Science and Artificial Intelligence

Dual Degree Project
Project Title: Optimization at Scale

Author:

Sreerag S

CE20B111

IDDD Data Science

Guides:

Dr. Sivaram Ambikasaran, Dr. Sri Vallabha Deevi

Department of Data Science and AI

Abstract

Combinatorial problems like Job Scheduling, Knapsack, Assignment, Vehicle Routing, among others, are critical in fields such as operations research and computer science. These problems have practical implications in logistics, production, and resource planning. Heuristic techniques and exact solvers based on linear, mixed-integer, and constraint programming are traditional approaches for solving these problems. While effective, these approaches frequently encounter scalability issues as problem sizes grow. This study focuses on the performance and scalability of three popular optimization solvers: PuLP, Google OR-Tools, and Gurobi, over a variety of standard optimization problems. We develop functions to generate synthetic data for each optimization problem to facilitate solver comparisons and evaluate solution quality and computational efficiency as problem complexity grows. Furthermore, we explore the RL4CO framework, a reinforcement learning-based approach, for tackling large-scale instances of the Capacitated Vehicle Routing Problem (CVRP). We aim to demonstrate how RL4CO technique can yield near optimal solutions in considerably fewer time as compared to conventional solvers, highlighting its potential as a fast and scalable solution for complex combinatorial optimization tasks. This study emphasizes machine learning's growing role in advancing large-scale optimization.

Keywords: Mixed-integer Programming, Constraint Programming, Reinforcement Learning, Heuristic Methods

GitHub Link: https://github.com/sreerag1234/Dual-Degree-Project

Thesis Certificate

This is to certify that the report titled **"Optimization at Scale"**, submitted by **Sreerag S (CE20B111)**, in partial fulfillment of the requirements for the award of the degree of **Master of Technology in Data Science**, Indian Institute of Technology Madras, is a record of the work done by him during the academic year **2024–2025** in the **Department of Data Science and AI**, IIT Madras, India, under our supervision.

Prof. Sivaram Ambikasaran
Associate Professor
Department of Mathematics
Department of Data Science & AI
Wadhwani School of Data Science & AI
Indian Institute of Technology Madras

Dr. Sri Vallabha Deevi
Director, Data Science, Tiger Analytics, Chennai, India
Adjunct Faculty, Department of Data Science & AI, IIT Madras

10 May 2025

Acknowledgements

I would like to express my sincere gratitude to Prof. Sivaram Ambikasaran and Dr. Sri Vallabha Deevi for their constant support and guidance throughout the course of this project. Their insightful inputs, mentorship, and encouragement helped shape the path and in effective completion of this project.

I am particularly grateful to the faculty at IIT Madras, whose teaching and guidance provided me with the fundamental knowledge and motivation required to undertake this research. I extend my sincere appreciation to my friends for their constant support, motivation, and the many thought-provoking discussions that helped me overcome challenges along the way. I am extremely grateful to my family for their unconditional love and support, which have been a continual source of motivation and strength.

Finally, I thank the Department of Data Science & AI at IIT Madras and Tiger Analytics for providing me with the opportunity to learn more about this exciting and significant area of research through a rewarding and intellectually engaging project.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Combinatorial optimization problems are central to decision-making in many real-world applications, ranging from logistics and supply chain management to manufacturing, scheduling, and telecommunications. These problems deal with finding an optimal solution from a finite set of possibilities, where the number of possible configurations grow non linearly with problem size. Classic examples include the Job Scheduling, Traveling Salesman Problem (TSP), Assignment Problem and the Capacitated Vehicle Routing Problem (CVRP). Solving these problems efficiently and accurately is essential for enhancing operational performance, cutting down costs, and supporting intelligent automation in various industries.

Traditionally, combinatorial optimization problems are solved using exact solvers and heuristic approaches. To ensure optimal solutions, exact solvers use mathematical programming techniques such as linear programming (LP), mixed-integer programming (MIP), and constraint programming (CP). Even though these solvers are powerful and widely accepted, as problems size increases, their performance often deteriorates. In practical situations where decisions must be made quickly and on large-scale data, this limitation poses a significant challenge.

Recently, machine learning techniques have emerged as a promising alternative for solving combinatorial optimization problems. Among these, RL4CO stands out, offering a data-driven approach to tackle these challenges.

In this project, we begin by formulating and solving standard problems such as the Knapsack, TSP, Assignment, Job Scheduling etc. using a variety of solvers: PuLP, Gurobi, Google OR-Tools (CP-SAT and SCIP). To compare solver performance, we generate synthetic problem instances of varying sizes and complexities for each problem type. We benchmark their performance in terms of scalability, solution quality, and computation time across increasing problem sizes.

Finally, we explore the RL4CO approach, a reinforcement learning-based approach, focusing on the Capacitated Vehicle Routing Problem (CVRP) as a case study. We use benchmark instances from the VRPLib [1] dataset to run various experiments and conduct solver comparisons between RL4CO and traditional optimization solvers. Using pretrained or custom-trained RL models, we demonstrate how reinforcement learning can produce feasible and near-optimal solutions faster than traditional methods for large instances.

Chapter 2

A Landscape of Optimization Solvers

Over the decades, the field of combinatorial optimization has undergone a profound transformation in the tools and algorithms used to solve complex problems. From early exact algorithms to the most recent learning-based solvers, the field has advanced steadily, becoming more powerful, scalable, and adaptable.

The Era of Exact Solvers
Linear programming (LP) and integer programming (IP), in particular, were the foundation of the first combinatorial optimization solvers. Solvers like **CPLEX** [2] (developed in the late 1980s) and **Gurobi** [3] (2008) set the benchmark for commercial-grade performance, which performed exceptionally well in tasks like scheduling, assignment, routing and facility location. These solvers use techniques like *branch-and-cut*, *branch-and-bound*, and *cutting planes* to systemically explore the solution space.

Open-source alternatives like **GLPK** [4] and **CBC** [5] offered options that were easily accessible for academic use, while modeling libraries such as **PuLP** [6] and **Pyomo** [7] simplified formulation and integration in Python.

The Rise of Constraint Programming
While LP and IP solvers focus on optimization, Constraint Programming (CP) emerged as a powerful tool for scheduling and feasibility problems. With the help of constraint propagation and backtracking, CP can efficiently navigate massive search trees and performs best in situations where logical constraints dominate, such as exam scheduling or register allocation. Solvers such as **Choco** [8], **IBM CP Optimizer** [9], and **Google OR-Tools(CP-SAT)** [10] brought CP into common practice, often outperforming traditional IP models in structured scheduling scenarios.

Metaheuristics and Practical Scalability
As problem sizes grew, so did the need for fast, approximate solutions. Metaheuristic algorithms such as *Simulated Annealing*, *Genetic Algorithms*, *Tabu Search*, and *Ant Colony Optimization* offered flexible frameworks that could handle complex and highly non-linear problems. Although these techniques cannot ensure optimality, they frequently produce high-quality results in a reasonable amount of time, making them suitable for practical uses like 2D Bin Packing, TSP and CVRP.

Hybrid and Integrated Approaches
In practice, solvers are increasingly hybridized, combining the strengths of multiple methods. For example, a Genetic Algorithm may generate a good starting solution, which is then im-

proved by an IP solver. CP can be embedded within MIP frameworks, or heuristics can guide the branching strategy in exact solvers. One great example is Google OR-Tools, which provides CP, MIP, and routing solvers on a unified platform, facilitating the smooth integration of strategies.

Learning-Based Optimization
In recent years, the integration of Machine Learning with combinatorial optimization has opened up new frontiers. Techniques such as *Reinforcement Learning*, *Graph Neural Networks*, and *Neural Combinatorial Optimization* allow solvers to learn heuristics, predict promising regions of the solution space, or even generate full solutions. In problems like TSP and CVRP, frameworks like **RL4CO** [11] and techniques like *Pointer Networks* or *Neuro-Symbolic Solvers* have demonstrated promising results. These models can also be used to warm-start traditional solvers, speeding up convergence.

Modern Solver Ecosystem: Today, practitioners have access to a rich ecosystem of solvers, each with its own strengths:

- **Exact solvers** for provable optimality
- **CP solvers** for structured scheduling and feasibility
- **Heuristics/metaheuristics** for flexibility and speed
- **Learning-based models** for adaptability and predictive power

Solver choice now depends not just on problem structure, but also on scale, time constraints, and desired solution quality. This diversity allows for more intelligent solver design, often combining multiple paradigms.

Chapter 3

Resource Allocation and Scheduling

Consider a delivery company operating across multiple cities. It must assign drivers to delivery zones, decide where to open regional hubs, and schedule delivery slots to meet deadlines, all while minimizing costs, respecting capacity limits, and ensuring smooth operations. Such complex decision-making lies at the heart of resource allocation and scheduling problems.

These problems revolve around efficiently assigning tasks, jobs, or resources, such as workers, machines, or facilities, while respecting various constraints. The objective is often to minimize cost, maximize resource utilization, or ensure timely completion of tasks. They are foundational in fields such as workforce planning, logistics, manufacturing, and cloud computing.

Typical characteristics include:

- Limited availability of resources (e.g., workers, machines, locations)
- Constraints related to capacity, time, or precedence
- Objective functions such as minimizing total cost, makespan, or delay

Three classic problems in this domain are:

- **Assignment Problem:** This involves assigning agents (e.g. workers, machines) to tasks in a one-to-one manner to minimize total cost or completion time. For example, in a customer service center, determining which support agent should handle which ticket to ensure minimal waiting time and balanced workload falls under this category.
- **Facility Location Problem:** This focuses on determining optimal locations for facilities (e.g. warehouses, data centers) and assigning customers or demand points to them. For example, a retailer must decide which distribution centers to activate and which regions to serve to minimize logistics costs.
- **Job Scheduling Problem:** This deals with allocating jobs or operations to time slots or machines while minimizing metrics like total completion time (makespan) or lateness. A common example is in a factory where different jobs must be scheduled on machines, each with specific processing times and dependencies.

Despite the fact that all three involve resource decisions, they differ in structure. The assignment problem is about *matching* - finding the best one-to-one pairing between agents and tasks to minimize cost or time. The facility location problem focuses on *placement and allocation* - choosing optimal facility locations and assigning customers or demand points to minimize operational costs. In contrast, the job scheduling problem consists of *timing and sequencing* - organizing tasks or jobs over time and across machines to meet deadlines or reduce overall completion time.

3.1 Assignment Problem

The Assignment Problem involves allocating n workers to m tasks in a way that minimizes the total cost of assignments. There are more workers than tasks i.e. $n>m$. Each worker is assigned to at most one task, and each task is assigned to exactly one worker.

Given:
Let $i \in \{1, 2, \ldots, n\}$ represent the workers, and $j \in \{1, 2, \ldots, m\}$ represent the tasks. We are given a cost matrix $C = (c_{ij})$, where each entry c_{ij} denotes the cost of assigning worker i to task j.

Decision Variables:
To represent this assignment, we define binary decision variables x_{ij}, where $x_{ij} = 1$ if worker i is assigned to task j, and $x_{ij} = 0$ otherwise.

Mathematical Formulation:
The objective function of the assignment problem is to minimize the total cost of assigning workers to tasks. Mathematically, it is expressed as the sum of $c_{ij}x_{ij}$ over all workers i and tasks j, where c_{ij} denotes the cost of assigning worker i to task j, and x_{ij} is a binary variable indicating whether this assignment is made. The constraints ensures that each worker is assigned to at most one task, thereby preventing over-allocation (eq. 3.2) and that each task is assigned to exactly one worker, so that all tasks are completed (eq. 3.3). The mathematical representation of the objective function and the constraints are given below.

$$\text{Minimize} \quad \sum_{i=1}^{n}\sum_{j=1}^{m} c_{ij}x_{ij} \tag{3.1}$$

$$\text{Subject to} \quad \sum_{j=1}^{m} x_{ij} \leq 1 \quad \forall i \tag{3.2}$$

$$\sum_{i=1}^{n} x_{ij} = 1 \quad \forall j \tag{3.3}$$

$$x_{ij} \in \{0, 1\} \quad \forall i, j \tag{3.4}$$

Example:
Consider the following assignment problem involving 5 workers and 4 tasks. The cost matrix below represents the cost incurred if a particular worker is assigned to a specific task. Each row corresponds to a worker and each column corresponds to a task.

$$\text{Cost matrix(C)} = \begin{bmatrix} 90 & 80 & 75 & 70 \\ 35 & 85 & 55 & 65 \\ 125 & 95 & 90 & 95 \\ 45 & 110 & 95 & 115 \\ 50 & 100 & 90 & 100 \end{bmatrix}$$

Solution: On solving the problem using the PuLP solver, the following result was obtained.

- Total Cost: 265.0

Worker	Task Assigned	Cost
0	3	70
1	2	55
2	1	95
3	0	45

Table 3.1: Task assignments and associated costs

Solver Comparison:
To evaluate the performance of the solvers on the assignment problem, we performed a scaling study using synthetically generated data. A custom function was implemented to generate cost matrices based on the number of workers and tasks. The function used Python's random number generator to populate the cost matrix with integer values, simulating a wide range of realistic cost scenarios.The problem was solved at various scales to observe performance. The scaling study was carried out by increasing the number of workers and at each scale the data was same across all the solvers. The objective was to analyze solver behavior and runtime trends as the problem size grew. We compared the performance of four solvers: PuLP, Gurobi, Google OR-Tools, and CP-SAT Solver. Each solver's runtime was recorded across problem instances of increasing size, and the result is visualized in Figure 3.1.

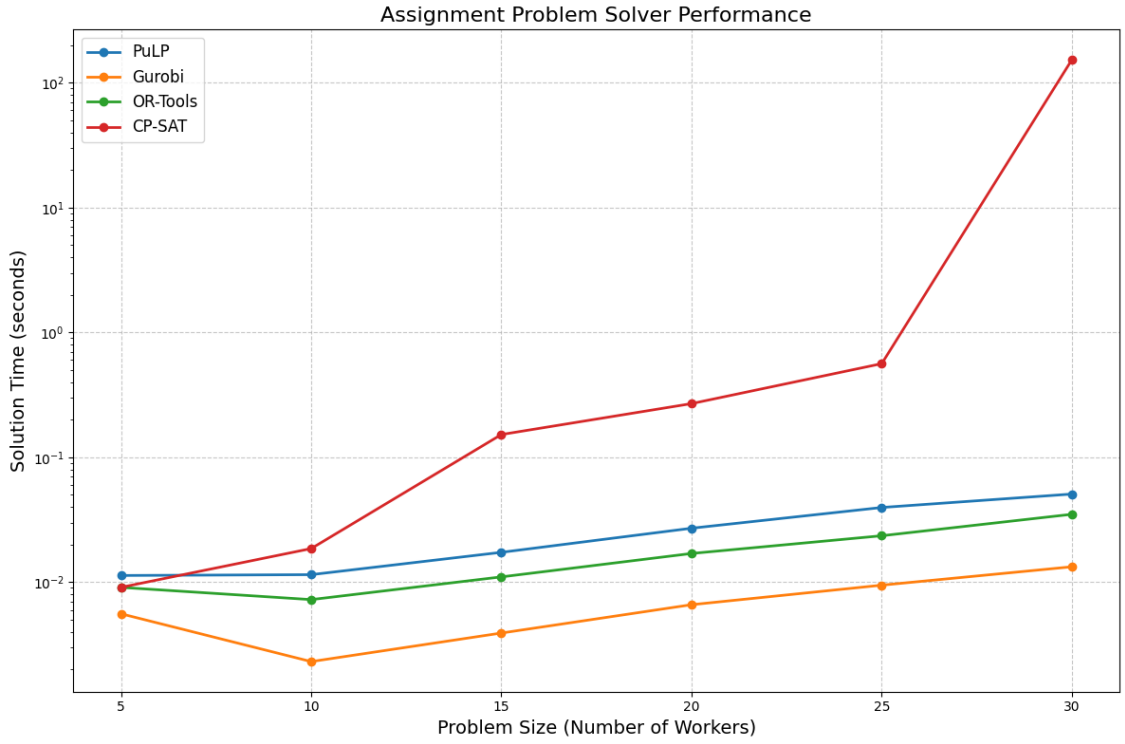

Figure 3.1: Solver Comparison for Assignment Problem

Applications:

- **Project Task Allocation:** In a consulting firm managing multiple client deliverables, employees with different expertise levels must be assigned to tasks such that deadlines are met and high-priority projects receive the required skill sets without overloading any individual.

- **Airline Crew Scheduling:** For a busy domestic airline network, crew members need to be scheduled across multiple connecting flights while ensuring compliance with mandatory rest periods, individual certifications, and seamless coordination between pilot and cabin staff.
- **School Course Scheduling:** In an academic institution, assigning teachers to available class slots must consider subject expertise, availability, and classroom constraints, ensuring an optimal teaching schedule that avoids conflicts and balances workload.

Variants:

3.1.1 Assignment with Teams of Workers

Here the workers are divided into teams, and there is a constraint that no team can be assigned more than a given number of tasks. The goal is to assign workers to tasks while minimizing the total cost.

Given:
Let $i \in \{1, 2, \ldots, n\}$ represent the workers(W), and $j \in \{1, 2, \ldots, m\}$ represent the tasks(T). We are given a cost matrix $C = (c_{ij})$, where each entry c_{ij} denotes the cost of assigning worker i to task j. Additionally, workers are grouped into teams, where $W_k \subseteq W$ represents the subset of workers belonging to team k. A constraint is imposed such that each team can be assigned at most $T_{\max}$ tasks.

- W_k: Set of workers belonging to team k
- $T_{\max}$: Maximum number of tasks that can be assigned to any team

Decision Variables:
To model the assignment, we define binary decision variables x_{ij}, where $x_{ij} = 1$ if worker i is assigned to task j, and $x_{ij} = 0$ otherwise.

Mathematical Formulation:
The objective of this problem is to minimize the total cost of assigning workers to tasks. This is represented by the sum of $c_{ij}x_{ij}$ over all worker-task pairs. The formulation includes three main constraints: each task must be assigned to exactly one worker to ensure full task coverage (eq. 3.6), each worker can be assigned at most one task to prevent overloading (eq. 3.7), and each team is restricted to handling at most $T_{\max}$ tasks to balance workload across teams (eq. 3.8). The mathematical formulation is given below.

$$\text{Minimize} \quad \sum_{i \in W} \sum_{j \in T} c_{ij} x_{ij} \tag{3.5}$$

$$\text{Subject to} \quad \sum_{i \in W} x_{ij} = 1 \qquad \forall j \in T \tag{3.6}$$

$$\sum_{j \in T} x_{ij} \leq 1 \qquad \forall i \in W \tag{3.7}$$

$$\sum_{i \in W_k} \sum_{j \in T} x_{ij} \leq T_{\max} \qquad \forall \text{ team } k \tag{3.8}$$

$$x_{ij} \in \{0, 1\} \qquad \forall i \in W, \forall j \in T \tag{3.9}$$

Example:
Consider the following assignment problem involving 6 workers and 4 tasks. The cost matrix below represents the cost incurred if a particular worker is assigned to a specific task. Each row corresponds to a worker and each column corresponds to a task.

$$\text{Cost matrix (C)} = \begin{bmatrix} 90 & 76 & 75 & 70 \\ 35 & 85 & 55 & 65 \\ 125 & 95 & 90 & 105 \\ 45 & 110 & 95 & 115 \\ 60 & 105 & 80 & 75 \\ 45 & 65 & 110 & 95 \end{bmatrix}$$

There are two predefined teams: Team 1 consists of workers $\{0,\ 2,\ 4\}$ and Team 2 consists of workers $\{1,\ 3,\ 5\}$. Each team can be assigned to a maximum of 2 tasks ($T_{\text{max}} = 2$).

Solution: On solving the problem under the given constraints in PuLP, the following result was obtained.

- Total cost: 250
- Assignments:

Worker	Task	Cost
0	2	75
1	0	35
4	3	75
5	1	65

Table 3.2: Task assignments and associated costs - Assignment with teams

Solver Comparison:
To evaluate the performance of different solvers in the assignment problem of teams, a benchmarking study was conducted using synthetically generated datasets. A custom Python function was used to generate cost matrices of varying sizes and to assign random workers to different teams.

The team-based assignment problem was solved at multiple scales by increasing the number of workers. We compared the performance of four solvers: PuLP, Gurobi, Google OR-Tools, and CP-SAT Solver. For each solver, the runtime was recorded as the problem size increased. Figure 3.2 illustrates the comparative runtime performance across different solvers for the team-constrained assignment setting.

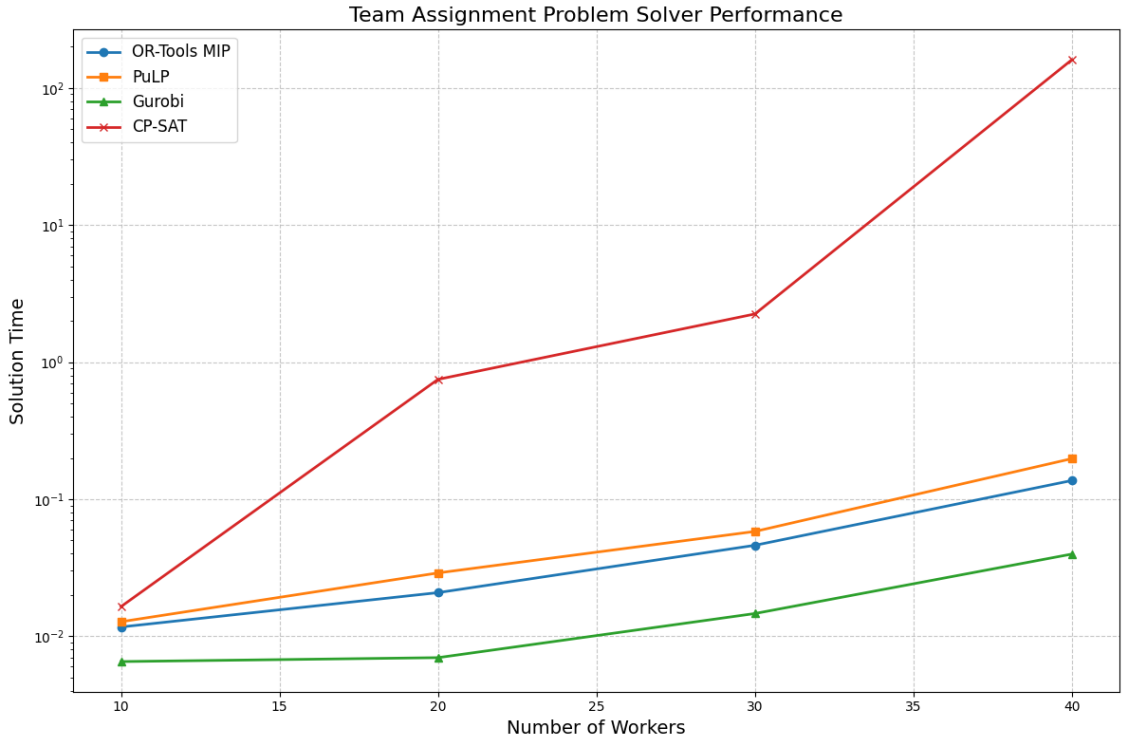

Figure 3.2: Solver Comparison for Team Assignment Problem

3.1.2 Assignment with Task Sizes

In this assignment problem, each task has a size, which represents how much time or effort the task requires. The total size of the tasks performed by each worker has a fixed bound.

Given:
Let $i \in \{1, 2, \ldots, n\}$ represent the workers(W), and $j \in \{1, 2, \ldots, m\}$ represent the tasks(T). We are given the following:

- $C = (c_{ij})$: Cost matrix, where c_{ij} denotes the cost of assigning worker i to task j
- s_j: Size (or workload) of task j
- $S_{\max}$: Maximum total size of tasks that a worker can handle

Decision Variables:
To represent the assignment decisions, we define binary variables x_{ij} where $x_{ij} = 1$ if worker i is assigned to task j, and otherwise.

Mathematical Formulation:
The objective of the assignment problem is to minimize the total cost incurred by assigning workers to tasks. This is represented as the sum of $c_{ij}x_{ij}$ over all workers i and tasks j. The constraints ensure that each task is assigned to exactly one worker (eq. 3.11) and that the total workload assigned to any worker does not exceed their capacity $S_{\max}$ (eq. 3.12). The mathematical formulation is given below:

$$\text{Minimize} \quad \sum_{i=1}^{n}\sum_{j=1}^{m} c_{ij}x_{ij} \tag{3.10}$$

$$\text{Subject to} \quad \sum_{i=1}^{n} x_{ij} = 1 \qquad \forall j = 1, \ldots, m \tag{3.11}$$

$$\sum_{j=1}^{m} s_j x_{ij} \leq S_{\max} \qquad \forall i = 1, \ldots, n \tag{3.12}$$

$$x_{ij} \in \{0, 1\} \qquad \forall i = 1, \ldots, n,\ \forall j = 1, \ldots, m \tag{3.13}$$

Example:
Consider the following assignment problem involving 10 workers and 8 tasks. The cost matrix shown below represents the cost incurred if a particular worker is assigned to a specific task. Each row corresponds to a worker and each column corresponds to a task.

$$\text{Cost matrix (C)} = \begin{bmatrix} 90 & 76 & 75 & 70 & 50 & 74 & 12 & 68 \\ 35 & 85 & 55 & 65 & 48 & 101 & 70 & 83 \\ 125 & 95 & 90 & 105 & 59 & 120 & 36 & 73 \\ 45 & 110 & 95 & 115 & 104 & 83 & 37 & 71 \\ 60 & 105 & 80 & 75 & 59 & 62 & 93 & 88 \\ 45 & 65 & 110 & 95 & 47 & 31 & 81 & 34 \\ 38 & 51 & 107 & 41 & 69 & 99 & 115 & 48 \\ 47 & 85 & 57 & 71 & 92 & 77 & 109 & 36 \\ 39 & 63 & 97 & 49 & 118 & 56 & 92 & 61 \\ 47 & 101 & 71 & 60 & 88 & 109 & 52 & 90 \end{bmatrix}$$

Each task has an associated size representing the workload: [10, 7, 3, 12, 15, 4, 11, 5]. Each worker can handle tasks whose total size does not exceed a maximum limit of $S_{\max} = 15$.

Solution: On solving the problem using the PuLP solver, the following result was obtained.

- Total cost: 326
- Assignments:

Worker	Task	Cost
0	6	12
1	0	35
1	2	55
2	4	59
5	5	31
5	7	34
6	1	51
8	3	49

Table 3.3: Optimal Task Assignments and Associated Costs

Solver Comparison:
The same process was carried out wherein we created a function to generate cost matrix based on the number of workers and tasks. The scaling comparison was done by increasing the number of workers. We compared the performance of four solvers: PuLP, Gurobi, Google OR-Tools, and CP-SAT Solver. For each solver, the runtime was recorded as the problem size increased. Figure 3.3 illustrates the comparative runtime performance across different solvers.

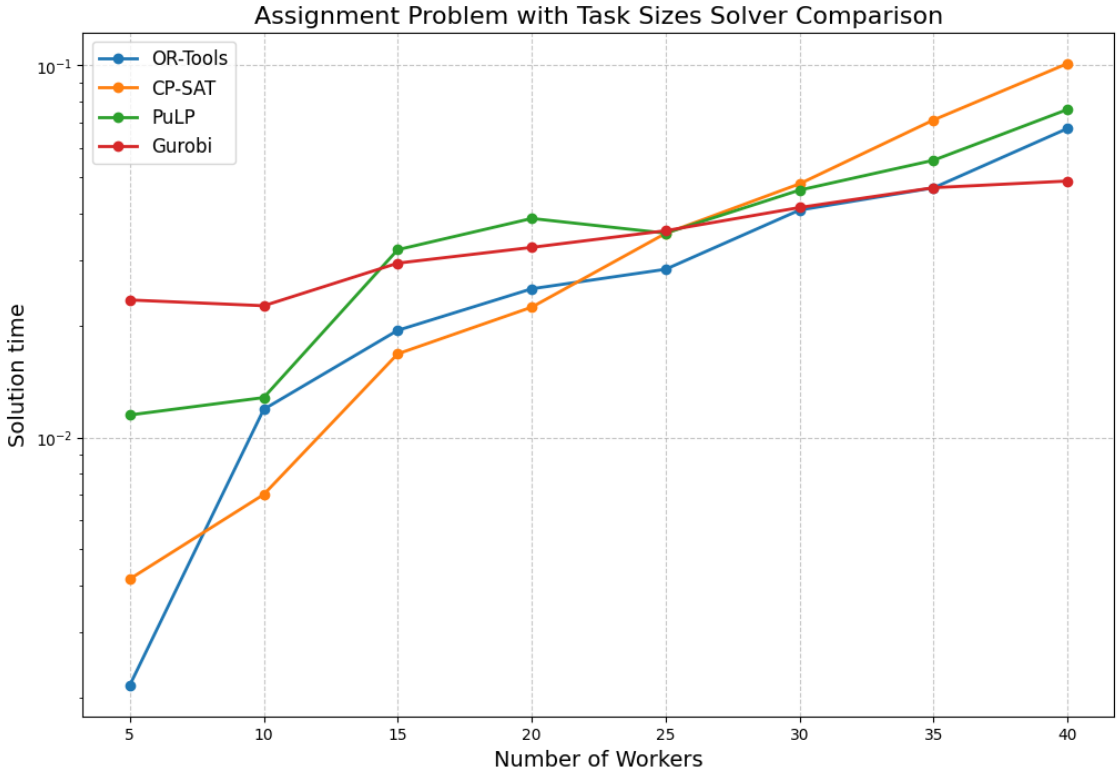

Figure 3.3: Solver Comparison for Assignment Problem with Task Sizes

Results:
Upon analysis of the solver comparison plots for various variants of the assignment problem, the following general trends were observed.

- For all variants of the assignment problem, the commercial solver, Gurobi computationally outperformed every other solver that was considered in the analysis. The solver was particularly very efficient in solving larger scale problems.
- Despite its robustness for constraint programming, the CP-SAT solver took the highest computational time to solve all variants of the assignment problem.
- PuLP and Google OR-Tools took moderate comparable computation times, faster than CP-SAT but slower than Gurobi.

These results highlight the efficiency of commercial solvers like Gurobi for large-scale problems, while open-source alternatives such as PuLP and Google OR-Tools offer a good balance of performance and accessibility.

3.2 Facility Location Problem

The Facility Location Problem is a classic optimization problem that involves determining the best locations to establish facilities (e.g., warehouses, factories, or service centers) to efficiently serve a set of customers while minimizing costs.

Given:
Let $i \in I$ represent potential facility locations, and $j \in J$ represent customer demand points. Each facility i has an associated fixed opening cost f_i, and serving customer j from facility i incurs a cost c_{ij}.

Decision Variables:
To model the facility location and assignment decisions, we define two sets of binary decision variables. Let y_i be a binary variable that equals 1 if facility i is opened and 0 otherwise. Let x_{ij} be a binary variable that equals 1 if customer j is assigned to facility i, and 0 otherwise.

- $x_{ij} \in \{0, 1\}$: 1 if customer j is assigned to facility i, 0 otherwise
- $y_i \in \{0, 1\}$: 1 if facility i is opened, 0 otherwise

Mathematical Formulation:
The goal of the facility location problem is to minimize the total cost, which includes both the fixed costs of opening facilities and the variable costs of serving customers from those facilities. The objective function sums the fixed cost f_i for each facility i if it is opened (i.e., $y_i = 1$), and the assignment cost c_{ij} for each customer-facility pair where $x_{ij} = 1$.

The constraints ensure that each customer is assigned to exactly one facility (eq. 3.15) and guarantee that a customer can only be served by a facility if that facility is open (eq. 3.16). The mathematical formulation is given below.

$$\text{Minimize} \quad \sum_{i \in I} f_i y_i + \sum_{i \in I} \sum_{j \in J} c_{ij} x_{ij} \tag{3.14}$$

$$\text{Subject to} \quad \sum_{i \in I} x_{ij} = 1 \qquad \forall j \in J \tag{3.15}$$

$$x_{ij} \leq y_i \qquad \forall i \in I,\ j \in J \tag{3.16}$$

$$x_{ij},\ y_i \in \{0, 1\} \qquad \forall i \in I,\ \forall j \in J \tag{3.17}$$

Example:
Consider a facility location problem with 3 potential facilities and 4 customers. The fixed costs to open the facilities are given by the vector [100, 200, 150]. The cost of serving each customer from each facility is provided in the matrix below, where each row corresponds to a customer and each column corresponds to a facility.

$$\text{Service Cost Matrix} = \begin{bmatrix} 20 & 24 & 11 \\ 18 & 22 & 30 \\ 25 & 15 & 17 \\ 22 & 19 & 30 \end{bmatrix}$$

Solution: On solving this problem, the optimal configuration involves opening only Facility 0. This solution minimizes the total cost by leveraging the low service costs from Facility 0

and avoiding the higher fixed costs of opening the other facilities. All customers are assigned to this facility, resulting in the following assignment matrix:

$$\text{Assignment Matrix} = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 0 & 0 \\ 1 & 0 & 0 \\ 1 & 0 & 0 \end{bmatrix}$$

Applications:

- **Warehouse selection:** A logistics provider planning to expand its distribution network uses facility location optimization to select which regional warehouses to operate in order to minimize delivery costs while ensuring all customer zones receive goods.
- **Retail expansion:** A retail chain analyzing population density and consumer demand patterns applies the facility location model to choose the best city districts to open new outlets, aiming to maximize accessibility and potential foot traffic.
- **Emergency service station planning:** Plan placement of fire stations, hospitals, or police stations to ensure fast response times across all service areas.

Solver Comparison:
To evaluate the performance of different solvers on the facility location problem, a scaling study was conducted using synthetically generated data. A custom function was used to generate random opening costs and service cost matrices based on the number of facilities and customers. The primary goal was to analyze how solver runtimes change as the problem size increases. We benchmarked the performance of three solvers: PuLP, Gurobi and Google OR-Tools. Figure 3.4 visualizes solver comparison as the problem size increases.

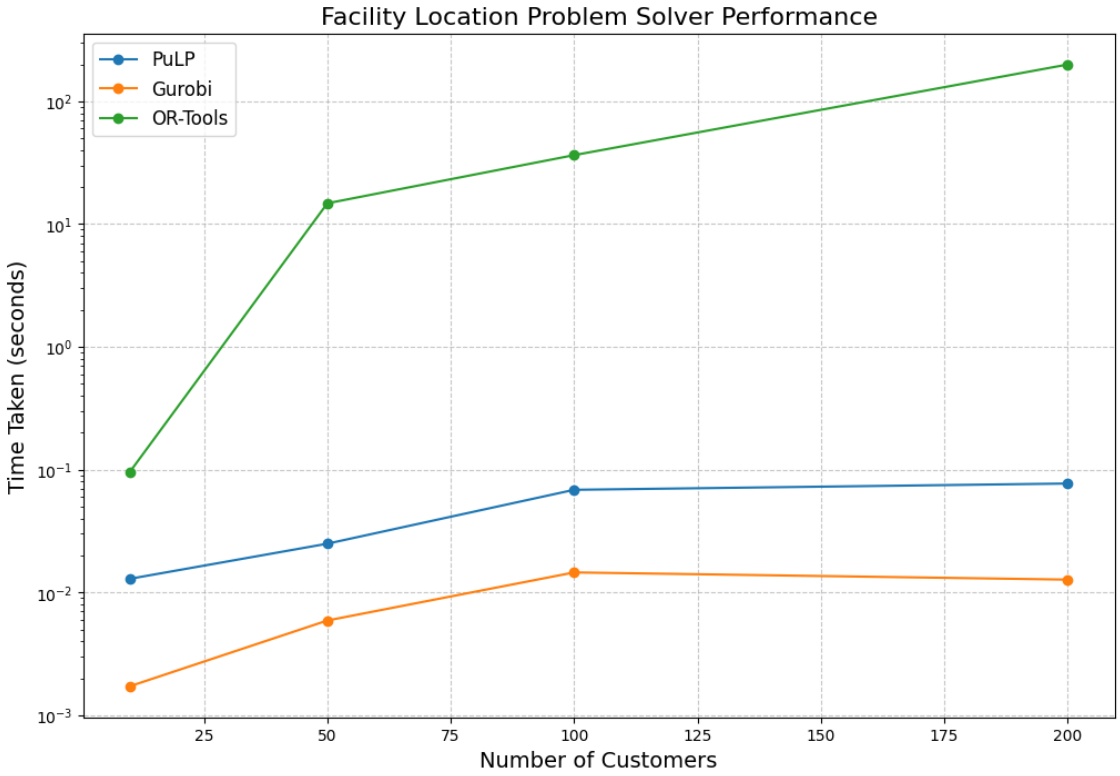

Figure 3.4: Solver Comparison for Facility Location Problem

Results:

Upon analysis of the solver comparison plots for the Facility Location problem, the following general trends were observed.

- The commercial solver, Gurobi computationally outperformed every other solver that was considered in the analysis. The solver was particularly very efficient in solving larger scale problems.
- PuLP was comparatively taking the most computational time in solving the problem.

3.3 Job Scheduling Problem

In parallel machine scheduling, n jobs must be assigned to m identical or unrelated machines to optimize a scheduling objective. The most common goal is to minimize the makespan i.e., the time at which the last job finishes. Each machine can process at most one job at a time, and each job is assigned to exactly one machine.

Given:
Consider a set of n jobs that must be assigned to m identical parallel machines. Each job i has an associated processing time p_i. The objective is to distribute the jobs among the machines in such a way that the the time taken by the machine that finishes last is minimized.
Decision Variables:
We define binary decision variables x_{ij}, where $x_{ij} = 1$ if job i is assigned to machine j, and 0 otherwise. We also define C_j as the total completion time (or load) on machine j, and $C_{\max}$ as the overall makespan.

Mathematical Formulation:
The objective of the job scheduling problem is to assign n jobs to m identical parallel machines such that the makespan (i.e., the maximum completion time across all machines) is minimized. The variable C_j represents the total load on machine j, and $C_{\max}$ captures the maximum of all machine completion times.

The constraints ensures that each job is assigned to exactly one machine (eq. 3.19), defines the load on each machine based on the jobs assigned to it (eq. 3.20) and ensures that $C_{\max}$ is at least as large as the load on any machine (eq. 3.21).

$$\text{Minimize} \quad C_{\max} \tag{3.18}$$

$$\text{Subject to} \quad \sum_{j=1}^{m} x_{ij} = 1 \qquad \forall i = 1, \ldots, n \tag{3.19}$$

$$C_j \geq \sum_{i=1}^{n} p_i \cdot x_{ij} \qquad \forall j = 1, \ldots, m \tag{3.20}$$

$$C_{\max} \geq C_j \qquad \forall j = 1, \ldots, m \tag{3.21}$$

$$x_{ij} \in \{0, 1\} \qquad \forall i = 1, \ldots, n, \ \forall j = 1, \ldots, m \tag{3.22}$$

Example:
Suppose we have 10 jobs and 3 machines. The processing times of the jobs are given by the array [13, 4, 3, 14, 7, 6, 6, 5, 14, 4]. The goal is to allocate these jobs across the machines to minimize the maximum load.

Solution: Solving this using an optimization solver yields an optimal makespan of 26.0. The job assignments to machines are as follows:

Machine	Assigned Jobs (Processing Time)	Total Load
0	Job 0 (13), Job 5 (6), Job 6 (6)	25
1	Job 3 (14), Job 4 (7), Job 7 (5)	26
2	Job 1 (4), Job 2 (3), Job 8 (14), Job 9 (4)	25

Table 3.4: Machine-wise Job Assignments and Total Loads

Solver Comparison:
A similar benchmarking process was conducted for the job scheduling problem, where we generated random job processing times and varied the number of jobs to evaluate scalability. The total number of machines was fixed while increasing the job count. We evaluated the performance of two solvers: PuLP and Google OR-Tools. For each solver, the runtime was measured as the number of jobs increased. Figure 3.5 presents a comparison of runtime performance across the different solvers.

Figure 3.5: Solver Comparison for Job Scheduling Problem

Applications:

- **Manufacturing planning:** In a factory setting with multiple production lines, jobs such as assembling, painting, or packaging must be assigned to machines to ensure that total production time is minimized and no machine remains underutilized.
- **Cloud computing:** When deploying large-scale data processing tasks in a cloud environment, computational jobs need to be scheduled across multiple servers or virtual machines to ensure balanced workload distribution and minimal processing delays.

- **Semiconductor fabrication:** In chip manufacturing, assigning wafer processing tasks to parallel machines with specific constraints to optimize throughput and reduce bottlenecks.

Variants:

3.3.1 Job Scheduling with Eligibility Constraints

The goal is to assign jobs to machines while minimizing the makespan. Each job must be assigned to exactly one machine, and machines can only process eligible jobs.

Given:
Let n be the number of jobs and m the number of machines. Each job $i \in \{1, 2, \ldots, n\}$ has a processing time p_i. Not all machines can process all jobs—each machine $j \in \{1, 2, \ldots, m\}$ has an associated eligibility set $J_j \subseteq \{1, 2, \ldots, n\}$, representing the jobs that can be assigned to it.

Decision Variables:
We define binary decision variables x_{ij}, where $x_{ij} = 1$ if job i is assigned to machine j, and 0 otherwise. We also define C_j as the total completion time on machine j, and $C_{\max}$ as the overall makespan.

Mathematical Formulation:
The objective of the job scheduling problem with eligibility constraints is to assign a set of jobs to a set of machines in such a way that the maximum makespan across all machines is minimized. Each job has a defined processing time and can only be assigned to a subset of machines, based on eligibility.

The constraints ensure that every job is assigned to exactly one machine (eq. 3.24), enforces eligibility by preventing assignments to ineligible machines (eq. 3.25) and computes the total processing time on each machine (eq. 3.26).

$$\text{Minimize} \quad C_{\max} = \max_j (C_j) \tag{3.23}$$

$$\text{Subject to} \quad \sum_{j=1}^{m} x_{ij} = 1 \quad \forall i \in \{1, \ldots, n\} \tag{3.24}$$

$$x_{ij} = 0 \quad \text{if } i \notin J_j \tag{3.25}$$

$$C_j = \sum_{i=1}^{n} p_i x_{ij} \quad \forall j \in \{1, \ldots, m\} \tag{3.26}$$

$$x_{ij} \in \{0, 1\} \quad \forall i, j \tag{3.27}$$

Example:
Consider a scenario with 10 jobs and 3 machines. Each job has a specific processing time, and can only be scheduled on a subset of machines due to eligibility constraints. The objective is to minimize the makespan, the maximum completion time among all machines.

- Processing times: [10, 20, 15, 30, 25, 5, 35, 50, 20, 10]

- Eligibility constraints:

$$\text{Machine 0: } \{0,\ 1,\ 2,\ 3,\ 4\}$$
$$\text{Machine 1: } \{5,\ 6,\ 7,\ 8,\ 9\}$$
$$\text{Machine 2: } \{2,\ 3,\ 4,\ 5,\ 6\}$$

Solution: On solving the above problem using PuLP, it was found that the optimal makespan was 80.0. The job-to-machine assignments are as follows:

- Machine 0: Jobs (0, 1, 2, 3)
- Machine 1: Jobs (7, 8, 9)
- Machine 2: Jobs (4, 5, 6)

Solver Comparison:
To evaluate solver performance on job scheduling with eligibility constraints, we created synthetic instances by varying the number of jobs. We compared the performance of PuLP, Gurobi and Google OR-Tools in terms of runtime as the problem size increased. The results are shown in Figure 3.6.

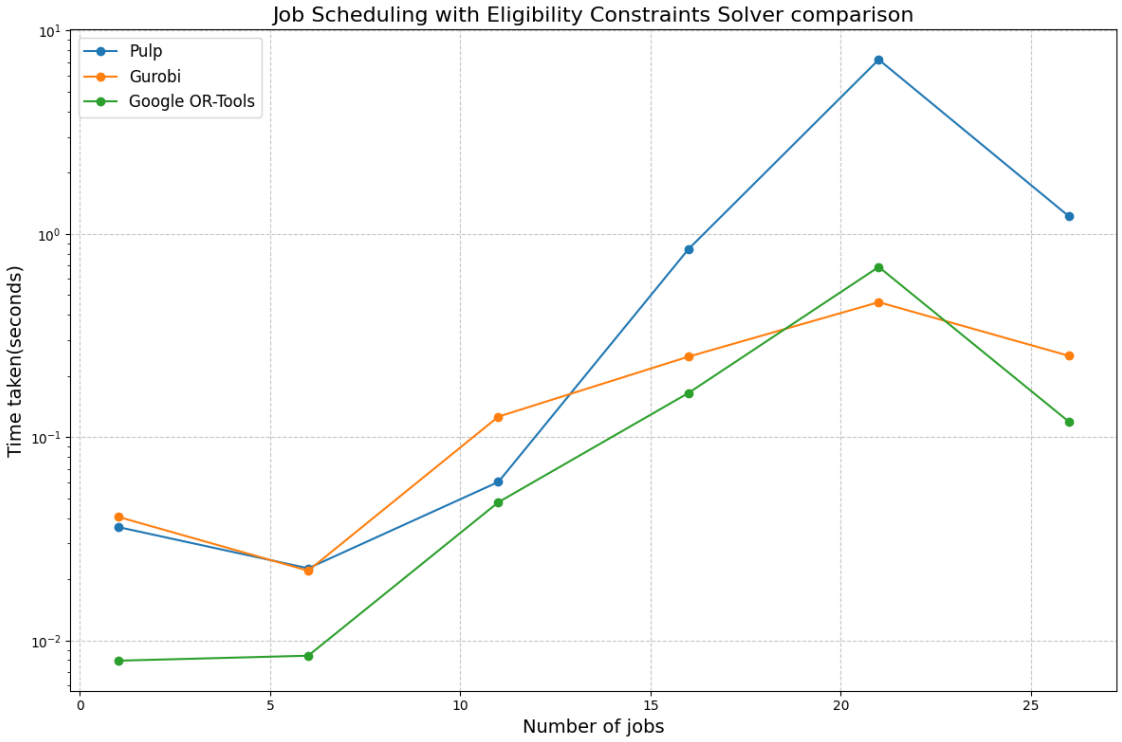

Figure 3.6: Solver Comparison for Job Scheduling with Eligibility Constraints

3.3.2 Parallel Machine Scheduling with Subtasks

This variant involves assigning and sequencing subtasks of jobs across machines, minimizing the makespan while respecting eligibility and ordering constraints.

Given:
Consider a set of n jobs, where each job i consists of multiple subtasks that need to be

assigned to m identical machines. Each subtask s of job i has an associated processing time $p_{i,s}$, and there may be eligibility constraints that determine which machine can process each subtask.

- $n_{\text{subtasks}}[i]$: Number of subtasks for job i
- $p_{i,s}$: Processing time of subtask s for job i
- Eligibility matrix indicating if subtask s of job i can be assigned to machine j

Decision Variables:
We define binary decision variables x_{ijs}, where $x_{ijs} = 1$ if subtask s of job i is assigned to machine j, and 0 otherwise. We also define C_j as the completion time of machine j, representing the total time taken by machine j to complete its assigned subtasks.

Mathematical Formulation:
The objective function minimizes the maximum completion time $C_{\max}$, which represents the time taken by the machine that finishes last. Each subtask is assigned to exactly one machine (eq. 3.29). A subtask can only be allocated to a machine if it is eligible for it (eq. 3.30). Equation 3.31 calculates the completion time for each machine based on the assigned subtask. Equation 3.32 ensures that the completion time of each machine respects the sequential order of subtasks across machines.

$$\text{Minimize} \quad C_{\max} = \max_j(C_j) \tag{3.28}$$

$$\text{Subject to} \quad \sum_{j=1}^{m} x_{ijs} = 1 \quad \forall i \in \{1, \ldots, n\},\ s \in \{1, \ldots, n_{\text{subtasks}}[i]\} \tag{3.29}$$

$$x_{ijs} = 0 \quad \text{if subtask } s \text{ of job } i \text{ is not eligible for machine } j \quad \forall i, j, s \tag{3.30}$$

$$C_j = \sum_{i=1}^{n} \sum_{s=1}^{n_{\text{subtasks}}[i]} p_{i,s} x_{ijs} \quad \forall j \in \{1, ., m\} \tag{3.31}$$

$$C_j \geq C_{j-1} + p_{i,s} \quad \forall j \in \{2, ., m\},\ i \in \{1, ., n\},\ s \in \{1, ., n_{\text{subtasks}}[i]\} \tag{3.32}$$

$$x_{ijs} \in \{0, 1\} \quad \forall i, j, s \tag{3.33}$$

Example:
Let's consider an example with 3 jobs and 3 machines. For each job, we assign 2 subtasks. We also ensure that the processing time for each of the subtasks is same for all the eligible machines of that subtask. The processing times of the subtasks for each job are given by the following matrix. Each row represents a particular job, each column corresponds to a subtask, and the values indicates the processing time.

$$\begin{bmatrix} 10 & 20 \\ 15 & 10 \\ 30 & 25 \end{bmatrix}$$

The eligibility of the subtasks for each job is as follows:

$$\text{Job 0: Subtask 0: } \{0, 1\},\ \text{Subtask 1: } \{1, 2\}$$
$$\text{Job 1: Subtask 0: } \{0, 2\},\ \text{Subtask 1: } \{1, 2\}$$
$$\text{Job 2: Subtask 0: } \{1, 2\},\ \text{Subtask 1: } \{0, 2\}$$

Solution: Solving this problem using PuLP yields an optimal makespan of 55. The subtask assignments to machines are as follows:

Job	Subtask	Assigned Machine	Start Time
0	0	1	0.00
0	1	1	10.00
1	0	0	0.00
1	1	2	15.00
2	0	2	0.00
2	1	0	30.00

Table 3.5: Subtask Assignments with Start Times

Solver Comparison:
To evaluate solver performance on job scheduling with subtasks, we created synthetic instances by varying the number of jobs and their subtasks. We compared the performance of PuLP, Gurobi and Google OR-Tools in terms of runtime as the problem size increased. The results are shown in Figure 3.7.

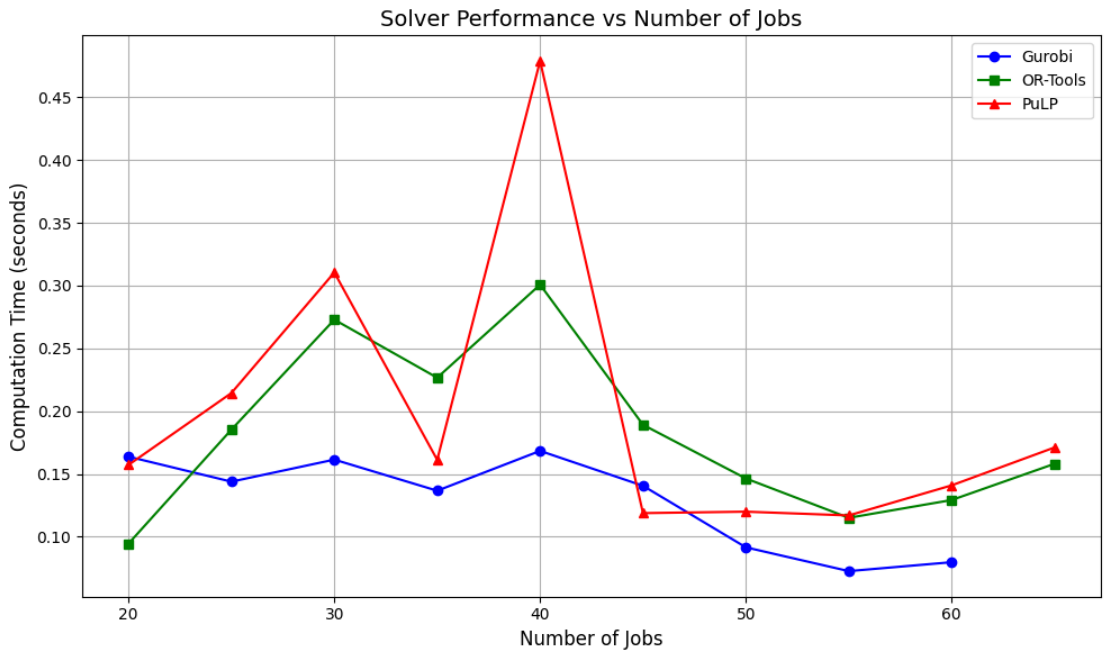

Figure 3.7: Solver Comparison for Job Scheduling with Subtasks

Results:
Upon analysis of the solver comparison plots for various variants of the Scheduling problem, the following general trends were observed.

- For Scheduling problem with Subtasks, Gurobi computationally outperformed every other solver that was considered in the analysis.
- For the problem with eligibility constraints, Google OR-Tools took the least time.

Chapter 4

Financial and Packing Optimization

Consider an investor who wants to grow their wealth over time and has a set amount of capital. A variety of investment options are available to them, including stocks, government bonds, fixed deposits, gold, and even real estate. Each option has its own risk and return characteristics. For instance, while fixed deposits offer stable returns, they may not keep up with inflation. Conversely, stocks can yield substantial returns, but they are associated with a high degree of volatility. The investor's challenge lies in selecting an investment combination that balances risk and reward while staying within budget. This type of decision-making captures the core of financial optimization. Similarly, packing optimization plays a vital role in material transportation, where the goal is to efficiently arrange items within defined space and weight limits. This category involves selecting or packing items to maximize value or minimize cost, often under resource constraints such as budget, space, or risk limits. These problems are widely encountered in finance, logistics, and operations management, where strategic selection and allocation are crucial.
Typical characteristics include:

- A set of items or options, each with associated value, cost, size, or risk
- Decisions involving selection, inclusion, or allocation under constraints
- Objectives such as maximizing returns, minimizing total cost, or optimizing space usage

Three famous problems in this domain are:

- **Portfolio Optimization:** Allocating capital among a range of investment options to maximize expected return under a specified risk threshold. For example, building a portfolio that aligns with an investor's goals balancing investments between low-risk bonds and high-return stocks .
- **Bin Packing Problem:** Packing a set of items of varying sizes into the minimum number of fixed-capacity bins without exceeding capacity limits. A typical use case is in warehouse logistics where different products must be efficiently packed into storage units or containers.
- **Knapsack Problem:** Selecting a subset of items to pack into a knapsack to maximize value without exceeding weight limit. This problem appears in cargo loading scenarios, where each item has a weight and value, and the goal is to optimize what gets shipped.

While all three problems involve making choices under constraints, their structures differ. Portfolio optimization deals with *risk-return trade-offs*, balancing expected returns against

uncertainty. The bin packing problem focuses on *space efficiency*, minimizing the number of bins used. The knapsack problem deals with *value maximization*, selecting the most valuable items within capacity limits.

4.1 Portfolio Optimization

Portfolio optimization is the process of selecting the best mix of assets to achieve a desired return while minimizing risk. The classic approach is Mean-Variance Optimization. The goal is to maximize expected returns for a given level of risk or minimize risk for a target return.

Given:
Consider a portfolio consisting of N financial assets. Each asset i has an associated expected return r_i, and the overall risk of the portfolio is captured by the covariance matrix Σ, which quantifies the variance and correlation between asset returns. The investor also specifies a target return R_{target} that the portfolio is expected to meet or exceed.

- r_i: Expected return of asset i
- Σ: Covariance matrix of asset returns (used as a risk measure)
- R_{target}: Desired target return

Decision Variables:
The main decision is to determine the allocation of capital to each asset. Let $w_i \in [0, 1]$ denote the fraction of total investment assigned to asset i. These weights must sum to 1, representing a fully invested portfolio.

Mathematical Formulation:
The objective is to minimize the portfolio variance, representing the investment risk, subject to constraints that ensure a minimum expected return, full budget allocation, and no short-selling.

The expected return of the portfolio must meet or exceed a specified target R_{target}(eq. 4.2). Full investment of the available capital is enforced by ensuring that the sum of asset weights equals one (eq. 4.3). To prevent short-selling, all asset weights are constrained to be non-negative (eq. 4.4).

$$\text{Minimize} \quad \mathbf{w}^T \Sigma \mathbf{w} \tag{4.1}$$

$$\text{Subject to} \quad \mathbf{w}^T \mathbf{r} \geq R_{\text{target}} \tag{4.2}$$

$$\sum_i w_i = 1 \tag{4.3}$$

$$w_i \geq 0 \quad \forall i \tag{4.4}$$

Example:
Suppose we have 5 assets with expected returns and a correlation matrix as defined below.

The goal is to allocate weights to these assets to minimize portfolio risk while ensuring an expected return of at least 12%.

- Expected returns: [0.12, 0.10, 0.15, 0.11, 0.09]
- Covariance matrix:

$$\begin{bmatrix} 1.00 & 0.60 & 0.50 & 0.70 & 0.25 \\ 0.60 & 1.00 & 0.45 & 0.65 & 0.30 \\ 0.50 & 0.45 & 1.00 & 0.55 & 0.20 \\ 0.70 & 0.65 & 0.55 & 1.00 & 0.35 \\ 0.25 & 0.30 & 0.20 & 0.35 & 1.00 \end{bmatrix}$$

Solution: On solving the optimization model using PuLP solver, we get the following portfolio:

- Expected return: 12.00%
- Expected risk (standard deviation): 17.57%

Asset	Weight (%)
A	20.00
B	0.00
C	40.00
D	0.00
E	40.00

Table 4.1: Optimal Asset Allocation for Target Return of 12%

Applications:

- **Asset management:** Involves strategically distributing investments across a range of financial assets (such as stocks, bonds, and commodities) to achieve optimal returns while managing exposure to risk.
- **Pension fund design:** Focuses on constructing investment portfolios that ensure long-term financial sustainability of retirement funds by balancing expected returns with future liabilities and minimizing volatility.
- **Robo-advisors:** Digital platforms that automate investment decisions using portfolio optimization techniques to recommend diversified, risk-adjusted portfolios tailored to an individual's financial goals and risk tolerance.

Solver Comparison:
To evaluate the performance of different solvers on portfolio optimization problems, we tested them across instances with varying numbers of assets and correlation structures. The solvers compared include PuLP, Gurobi, and Google-OR Tools. The scaling was done by increasing the number of assets and all the solvers were tested on the same instances. The runtime and scalability results are summarized in Figure 4.1.

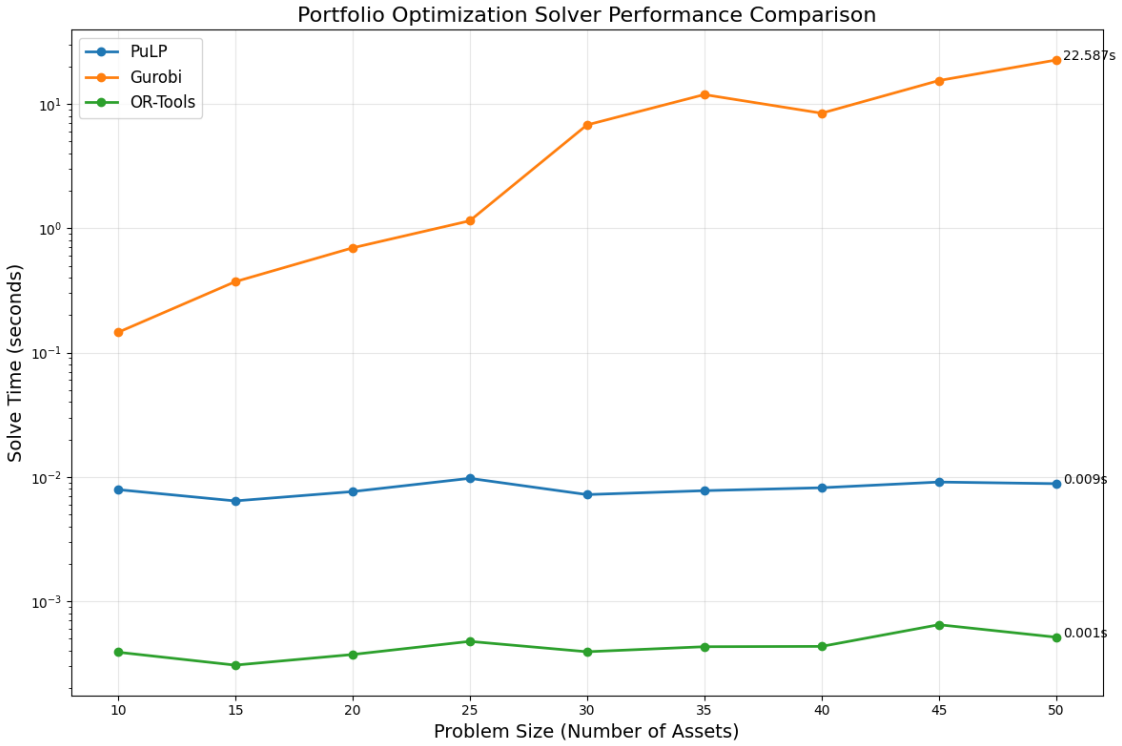

Figure 4.1: Solver Comparison for Portfolio Optimization

Results:

Upon analysis of the solver comparison plots for Portfolio Optimization problem, the following general trends were observed.

- Google OR-Tools computationally outperformed other solvers in solving the problems.
- Gurobi took the most amount of time to solve the problems, with its runtime increasing significantly as the problem size grew.
- PuLP and Google OR-Tools exhibit consistent performance as the problem scales, maintaining nearly constant computational times even as the number of assets grows.

4.2 2D Bin Packing Problem

In the 2D Bin Packing Problem, the goal is to pack a set of rectangular items into the fewest number of identical rectangular bins without overlaps. Each item must lie entirely within its assigned bin. For instance, a shipping company tries to optimize container space when transporting goods, ensuring fewer containers are used resulting in reduced costs.

Given:

Consider a two-dimensional bin packing problem where n items must be placed into a finite set of bins without exceeding their dimensions. Each item has a defined width and height, and rotation by 90 degrees is allowed.

- w_i, h_i: Width and height of item i
- m: Maximum number of bins available

- W, H: Width and height of each bin

Decision Variables:
The decision variables define item placement and orientation within bins. x_{ij} and y_j indicate item-to-bin assignments and bin usage, respectively. X_i, Y_i specify item positions, and r_i allows 90° rotation, enabling more flexible packing.

- $x_{ij} \in \{0, 1\}$: 1 if item i is placed in bin j, 0 otherwise.
- $y_j \in \{0, 1\}$: 1 if bin j is used, 0 otherwise.
- X_i, Y_i: Bottom-left coordinates of item i in its assigned bin.
- $r_i \in \{0, 1\}$: 1 if item i is rotated by 90°, 0 otherwise.

Mathematical Formulation:
The objective is to minimize the number of bins used while packing all items without overlap and within bin boundaries. The formulation accounts for item rotation to better utilize space.

Each item must be assigned to exactly one bin (eq. 4.6). A bin is considered used only if it contains at least one item (eq. 4.7). The constraints in Equations 4.8 and 4.9 make sure that items fit within the bin's width and height, respectively. Equation 4.10 prevents any overlapping among items placed in the same bin. Lastly, the effective width and height of each item, depending on its rotation, are computed using Equations 4.11 and 4.12.

$$\text{Minimize} \quad \sum_{j=1}^{m} y_j \tag{4.5}$$

$$\text{Subject to} \quad \sum_{j=1}^{m} x_{ij} = 1 \quad \forall i = 1, \ldots, n \tag{4.6}$$

$$x_{ij} \leq y_j \quad \forall i, j \tag{4.7}$$

$$X_i + w'_i \leq W \quad \forall i \tag{4.8}$$

$$Y_i + h'_i \leq H \quad \forall i \tag{4.9}$$

$$\begin{aligned} &\text{For all } i < k, \text{ if } x_{ij} = x_{kj} = 1 : \\ &\quad (X_i + w'_i \leq X_k) \ \vee \ (X_k + w'_k \leq X_i) \\ &\quad \vee \ (Y_i + h'_i \leq Y_k) \ \vee \ (Y_k + h'_k \leq Y_i) \end{aligned} \tag{4.10}$$

$$w'_i = (1 - r_i)w_i + r_i h_i \quad \forall i \tag{4.11}$$

$$h'_i = (1 - r_i)h_i + r_i w_i \quad \forall i \tag{4.12}$$

$$x_{ij}, y_j, r_i \in \{0, 1\}, \quad X_i, Y_i \geq 0 \tag{4.13}$$

Heuristics for 2D Bin Packing:
Several heuristic and metaheuristic approaches are commonly used to solve the 2D Bin Packing Problem effectively. These approaches aim to generate high-quality solutions in reasonable computational times.

The **Bottom-Left Fill Algorithm** is a constructive heuristic commonly applied to bin packing problems. Place items sequentially at the lowest possible position, and among those select the leftmost spot. This approach effectively fills the bins in a bottom-left fashion. Despite its simplicity, the algorithm produces reasonably good results and is often used as a stratifying solution for other metaheuristics [12].

Simulated Annealing is inspired by the physical process of annealing in metallurgy [13]. It begins with an initial solution often generated by a heuristic like Bottom-Left Fill and explores the solution space by occasionally accepting worse solutions with a probability governed by a temperature parameter. This temperature gradually decreases, reducing the acceptance of inferior solutions over time.

Genetic Algorithms mimic the process of natural selection and evolve a population of candidate solutions over multiple generations [14]. The algorithm begins with a population of random permutations of items. Through selection, crossover, and mutation, the population evolves toward better solutions.

Tabu Search enhances local search by maintaining a tabu list that records recently visited solutions or moves, thus preventing cycling [15]. Starting from an initial configuration, it explores the neighborhood and moves to the best non-tabu neighbor.

Particle Swarm Optimization (PSO) draws inspiration from the collective behavior of birds and fish [16]. It operates on a swarm of particles, each representing a potential solution. The particles adjust their positions based on individual and global best positions.

Ant Colony Optimization (ACO) is based on the behavior of ants looking for food [17]. Artificial ants build solutions incrementally using heuristic information and pheromone trails, reinforcing good solutions through pheromone updates.

Example:
This example considers a 2D bin packing problem involving 10 rectangular items. Each bin has fixed dimensions of width 10 and height 10. The items to be packed have the following width and height:

Item ID	Dimensions(Width × Height)
0	7 ×1
1	5 ×9
2	1 ×10
3	9 ×2
4	10 ×1
5	3 ×3
6	7 ×9
7	5 ×5
8	9 ×6
9	2 ×7

Table 4.2: Item Dimensions for bin packing example

Solution: This example was solved using PuLP(Fig. 4.2), Google OR-Tools(Fig. 4.3), Bottom-Left Fill(Fig. 4.4), Simulated Annealing(Fig. 4.5), Genetic Algorithm(Fig. 4.6), Tabu Search(Fig. 4.7), Particle Swarm Optimization(Fig. 4.8) and Ant Colony Optimization(Fig. 4.9). It was observed that while all other solvers used 3 bins to pack all the items, Bottom-Left Fill took 4.

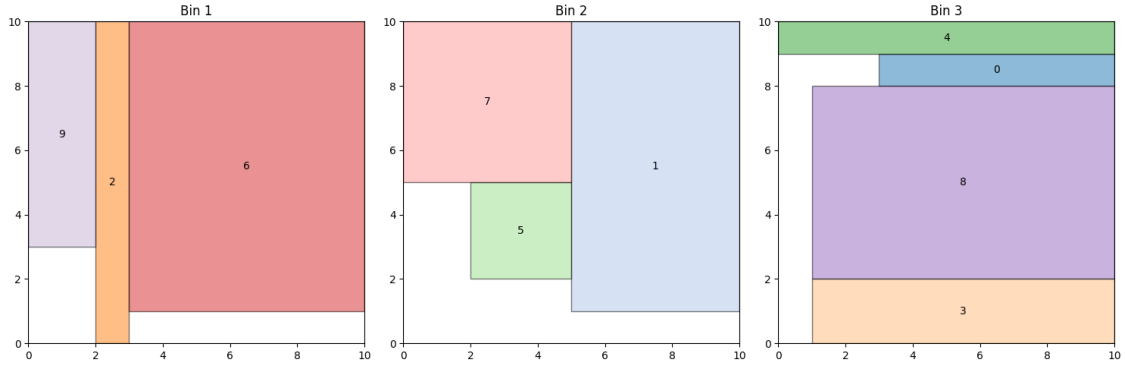

Figure 4.2: PuLP solution

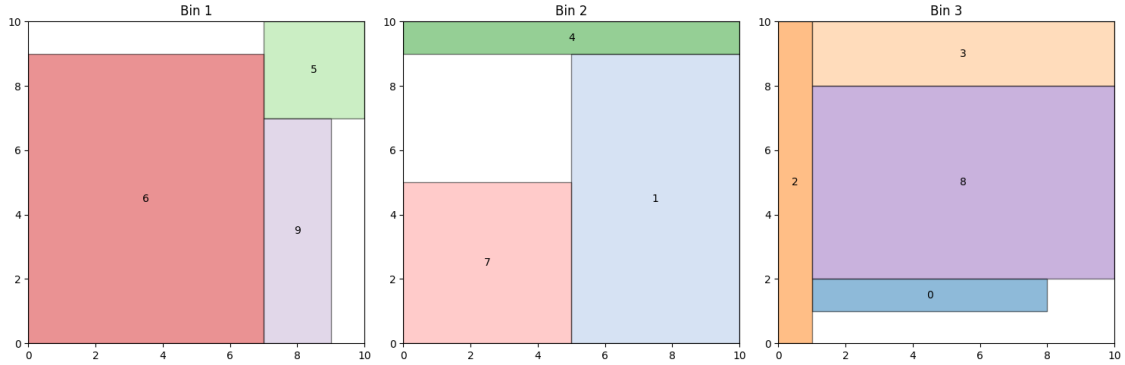

Figure 4.3: Google-OR solution

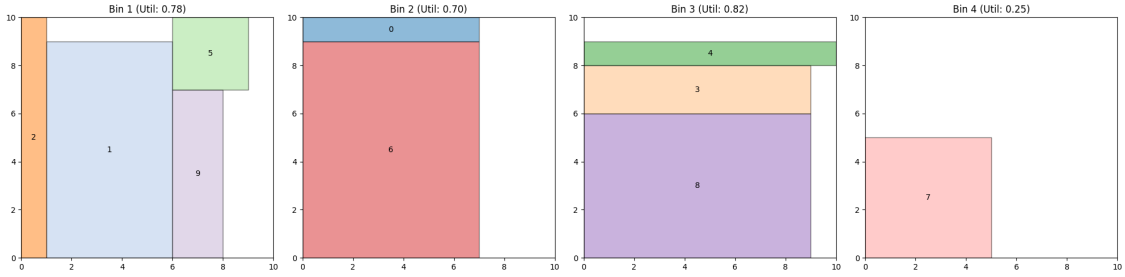

Figure 4.4: Bottom-Left Fill solution

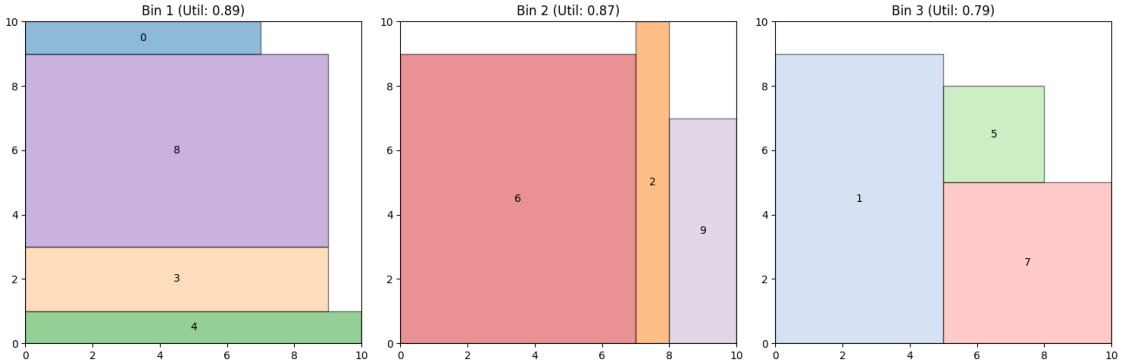

Figure 4.5: Simulated Annealing solution

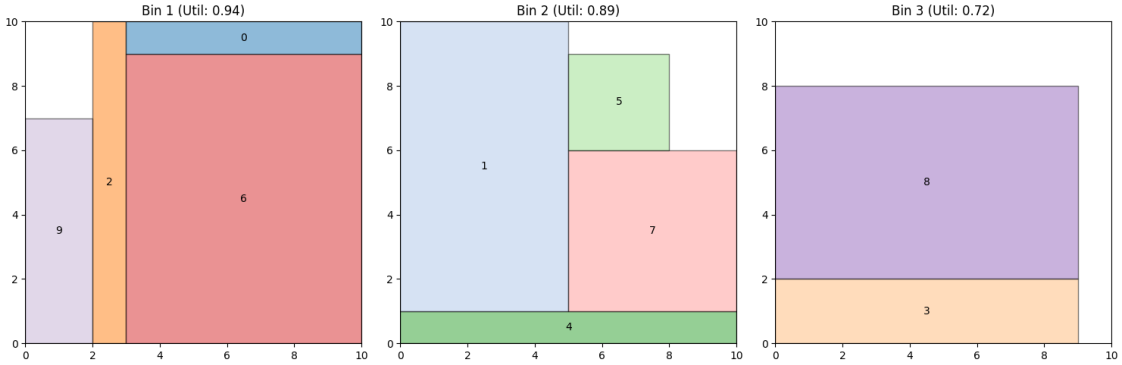

Figure 4.6: Genetic Algorithm solution

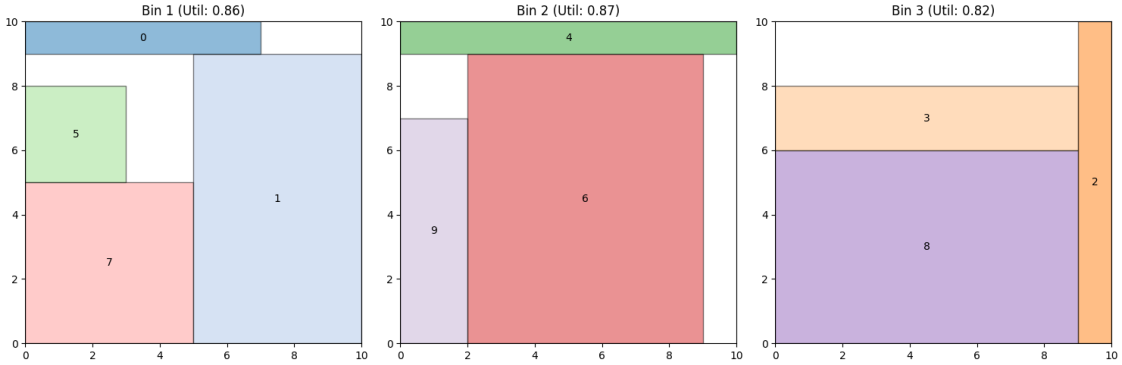

Figure 4.7: Tabu Search solution

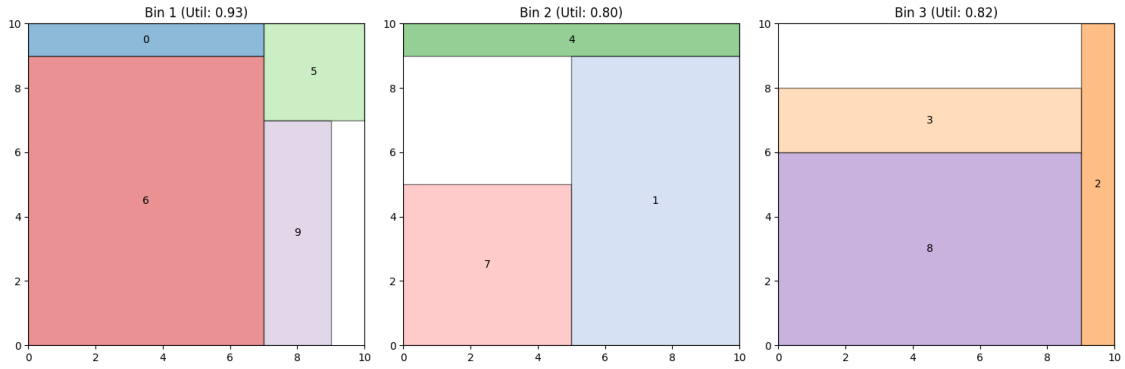

Figure 4.8: Particle Swarm Optimization solution

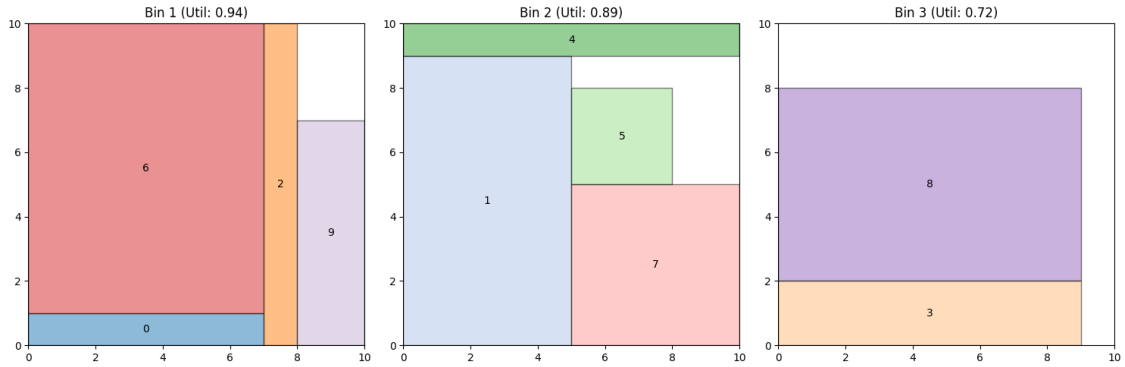

Figure 4.9: Ant Colony Optimization solution

Applications:

- **Container Loading:** Efficiently loading containers with varying item sizes to minimize the number of containers used or space wasted. For example, a shipping company can optimize container space when transporting goods, ensuring fewer containers are used, reducing costs.
- **Memory Management:** Allocating blocks of memory to processes in operating systems to make the best use of available memory while avoiding fragmentation.
- **VLSI Layout:** Arranging modules or components on a silicon chip without overlaps to optimize space and performance. An example of this is placing logic gates on a microchip in a way that minimizes the area used and maximizes the chip's processing power.

Solver Comparison:
To evaluate the performance of different solvers on 2D bin packing problem, we tested them across instances (which were synthetically generated) with varying numbers of items. The solvers compared include PuLP and Google-OR Tools. The scaling was done by increasing the number of items and all the solvers were tested on the same instances. The runtime and scalability results are summarized in Figure 4.10.

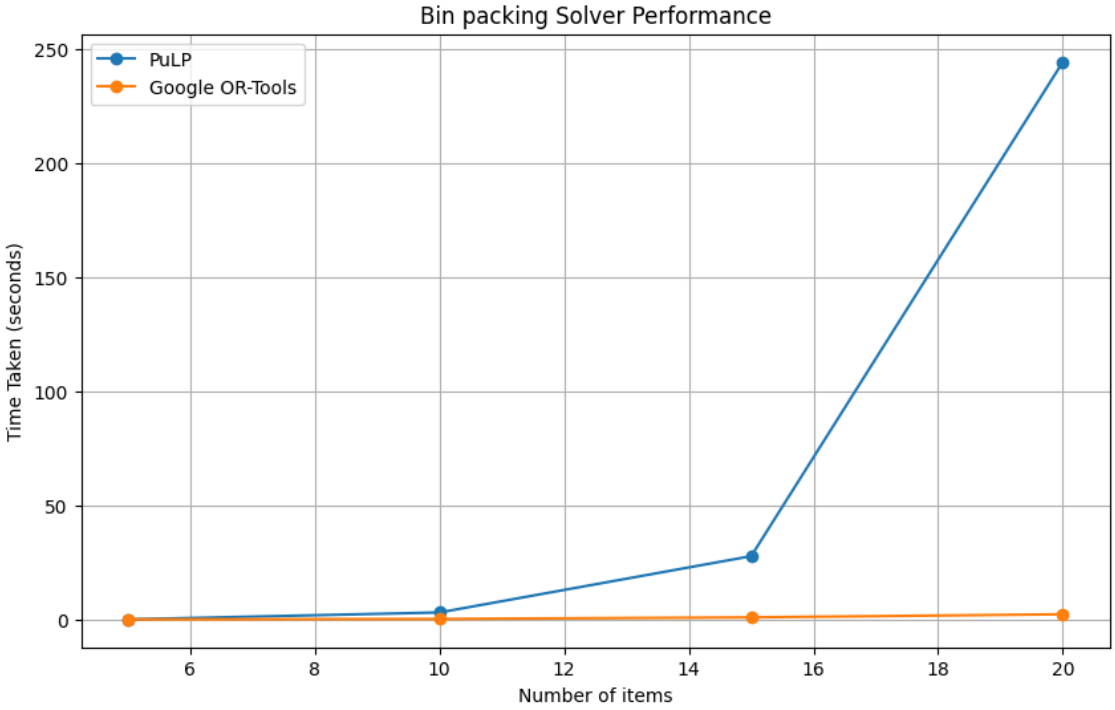

Figure 4.10: Solver Comparison for 2D Bin packing problem

Results:

Upon analysis of the solver comparison plots for the Bin Packing problem, the following general trends were observed.

- As the problem size increased, PuLP required significantly more time to solve the packing problem.
- Google OR-Tools consistently performed the best computationally, solving even large problems in comparatively less time.

4.3 Knapsack Problem

The Knapsack Problem involves selecting a subset of items, each with a given weight and value, to maximize the total value without exceeding a specified weight capacity. For instance, in emergency evacuation situations, a person has to optimally choose the things that he/she could carry. Each item will have a weight and a value associated with it and there will be capacity limitation on the things that one could carry.

Given:

Let n be the number of items. Each item $i \in \{1, 2, \ldots, n\}$ has a value v_i and weight w_i. The knapsack has a maximum capacity W.

Decision Variables:

We define binary decision variables x_i, where $x_i = 1$ if item i is included in the knapsack, and 0 otherwise.

Mathematical Formulation:

The objective function aims to maximize the total value of the items to be placed in a knapsack

subject to constraints. The constraints limit the total weight of selected items to not exceed the knapsack capacity W (eq. 4.15) and enforces a binary selection for each item, indicating whether it is chosen or not (eq. 4.16).

$$\text{Maximize} \quad \sum_{i=1}^{n} v_i x_i \tag{4.14}$$

$$\text{Subject to} \quad \sum_{i=1}^{n} w_i x_i \leq W \tag{4.15}$$

$$x_i \in \{0, 1\} \quad \forall i \in \{1, \ldots, n\} \tag{4.16}$$

Example:
Suppose we have 5 items with associated values and weights as defined below. The knapsack has a weight capacity of 100. The goal is to select items to maximize the total value while keeping the total weight within the knapsack's capacity.

- Values: $[50, 100, 150, 200, 250]$
- Weights: $[10, 20, 30, 40, 50]$

Solution: On solving the 0/1 knapsack problem using PuLP, we found that the optimal value is 500, which is obtained by selecting the 0^{th}, 3^{rd}, and 4^{th} indexed items.

- Item Selection (0/1): [1, 0, 0, 1, 1]

Applications:

- **Budget Allocation:** Selecting investments or projects to maximize return without exceeding a total budget. For example, a company may choose a subset of proposed marketing campaigns to fund in order to stay within a limited budget while maximizing potential revenue.
- **Cargo Loading:** Choosing a subset of items to load onto a vehicle without exceeding weight limit. For instance, a logistics company may load high-value packages into a delivery truck while ensuring the total weight does not exceed the vehicle's capacity.
- **Project Selection:** Determining which projects to pursue within a limited resource budget to maximize overall value. An example would be a firm selecting key R&D initiatives to fund within a fiscal year to get the highest expected return on innovation.

Solver Comparison:
A similar benchmarking process was conducted for the Knapsack problem, where we generated random values and weights for the items. The knapsack weight capacity was kept as half of the sum of weights. We evaluated the performance of four solvers: PuLP, Gurobi, Google OR-Tools and CP-SAT. For each solver, the runtime was measured as the number of items increased. Figure 4.11 presents a comparison of runtime performance across the different solvers.

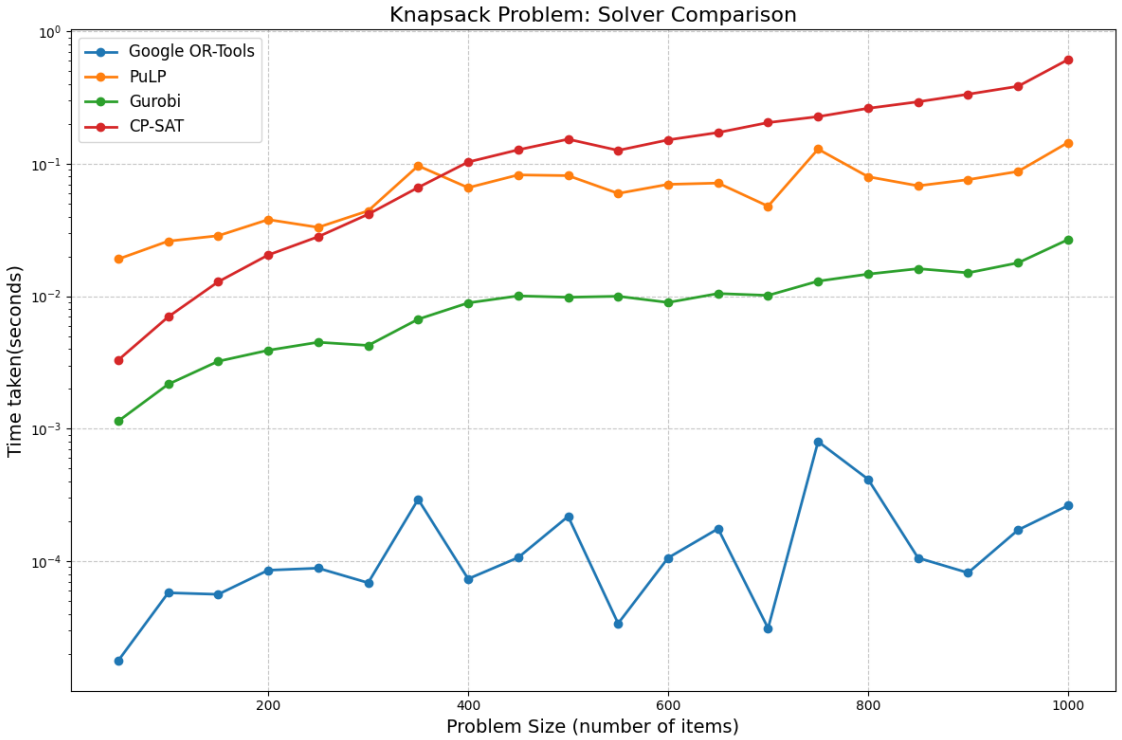

Figure 4.11: Solver Comparison for Knapsack problem

Variants:

4.3.1 Multiple Knapsack

The Multiple Knapsack Problem is an extension of the single Knapsack Problem where there are multiple knapsacks, each with its own capacity. The goal is to assign items to knapsacks in a way that maximizes the total value while respecting the capacity constraints of each knapsack.

Given:
Let n be the number of items and m be the number of knapsacks. Each item $i \in \{1, 2, \ldots, n\}$ has a value v_i and weight w_i. Each knapsack $j \in \{1, 2, \ldots, m\}$ has a capacity C_j

Decision Variables:
We define binary decision variables x_{ij}, where $x_{ij} = 1$ if item i is included in the knapsack j, and 0 otherwise.

Mathematical Formulation: The objective is to maximize the total value of selected items, subject to the constraints. The constraints ensures that the total weight of items assigned to each knapsack does not exceed its capacity (eq. 4.18) and that each item is assigned to at most one knapsack (eq 4.19).

$$\text{Maximize} \quad \sum_{j=1}^{m}\sum_{i=1}^{n} v_i x_{ij} \tag{4.17}$$

$$\text{Subject to} \quad \sum_{i=1}^{n} w_i x_{ij} \leq C_j \quad \forall j \in \{1, \ldots, m\} \tag{4.18}$$

$$\sum_{j=1}^{m} x_{ij} \leq 1 \quad \forall i \in \{1, \ldots, n\} \tag{4.19}$$

$$x_{ij} \in \{0, 1\} \quad \forall i, j \tag{4.20}$$

Example:
Consider a multiple knapsack problem with 8 items and 3 knapsacks. The knapsacks have different capacities. The weights, values of the items and the capacities of the knapsacks are given below.

- Weights = [20, 10, 40, 30, 25, 15, 35, 50]
- Values = [60, 100, 120, 80, 75, 95, 105, 90]
- Knapsack capacities = [60, 80, 70]

Solution: On solving the problem using Google OR-Tools, we obtained an optimal packing value of 665 and the knapsacks could hold all the items except item 0.

- Total packed value: 665
- Total packed weight: 205

Knapsack	Items	Packed Weight	Packed Value
0	1, 7	$10 + 50 = 60$	$100 + 90 = 190$
1	2, 6	$40 + 35 = 75$	$120 + 105 = 225$
2	3, 4, 5	$30 + 25 + 15 = 70$	$80 + 75 + 95 = 250$

Table 4.3: Multiple knapsack packing solution

Solver Comparison:
To evaluate the performance of different solvers on Multiple knapsack problem, we tested them across instances synthetically generated with varying numbers of items. The knapsacks were given equal capacities. We evaluated the performance of four solvers: PuLP, Gurobi, Google OR-Tools and CP-SAT. For each solver, the runtime was measured as the number of items increased. Figure 4.12 presents a comparison of runtime performance across the different solvers.

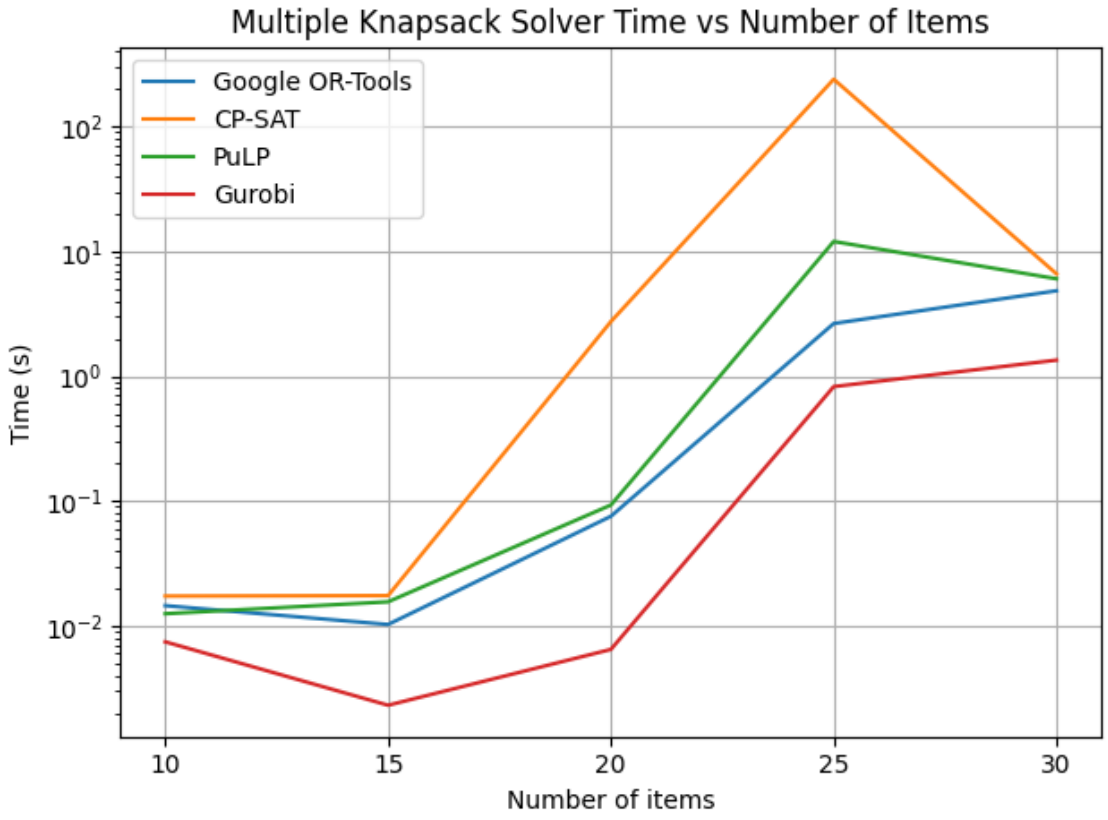

Figure 4.12: Solver comparison for Multiple Knapsack problem

We also performed evaluations to understand the performance of the PuLP solver as the number of knapsacks increased as well as when we gave the knapsacks equal and different capacities. The result is visualized in Figure 4.13.

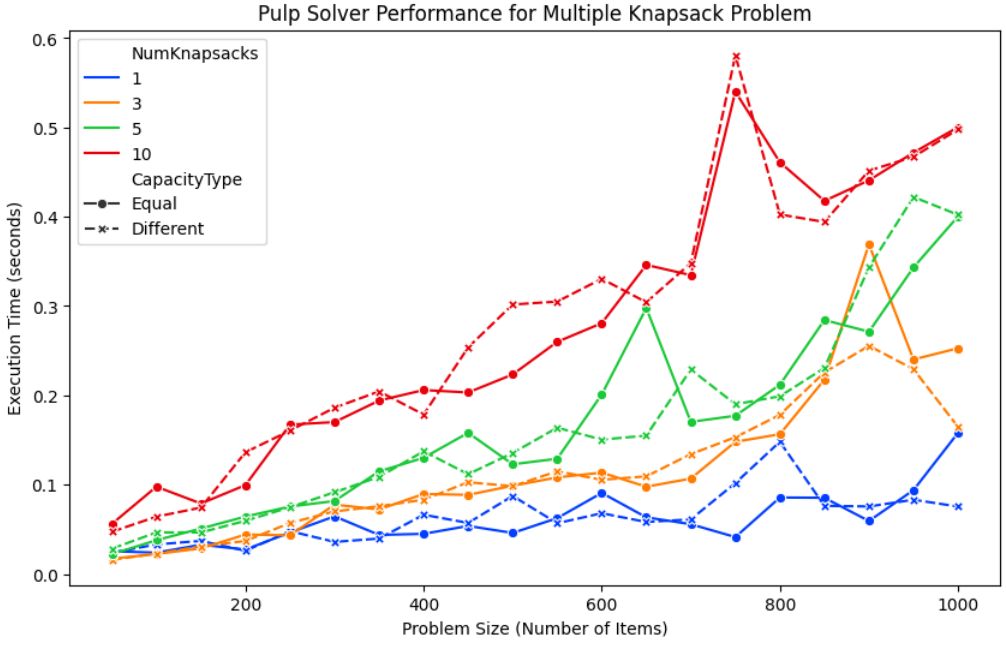

Figure 4.13: Multiple Knapsack problem - PuLP

Results:

Upon analysis of the solver comparison plots for the Knapsack problem, the following general trends were observed.

- Gurobi and Google OR-Tools performed computationally well in solving the Knapsack problems.
- CP-SAT solver took the highest computational time to solve both the single and multiple knapsack problems.
- For a given problem size, multiple knapsack problems with differing capacities took more computational time to solve than those with equal capacities.
- As the number of knapsacks increased, for a given problem size, the time taken to solve the problem also increased.

Chapter 5

Graph-based Problems

Many real-world optimization problems can be naturally modeled using graphs, where entities are represented as nodes and their relationships or interactions as edges. For instance, consider a telecommunications company optimizing bandwidth across a network of routers, or a university planning exam schedules to avoid clashes. Both problems rely on the underlying structure of graphs to formulate constraints and objectives.

Graph theory provides powerful tools for modeling connectivity, flow, coloring, and traversal. This enables efficient solution techniques for problems in areas such as transportation, telecommunications, scheduling, and network design.

Typical characteristics include the use of nodes and edges to represent interactions or capacities, with objectives often involving routing, coloring, or matching. Many of these problems can be effectively solved using network flow or graph traversal algorithms.
Two classic problems in this category are:

- **Network Flow Problem:** This problem involves finding the optimal flow through a directed graph to achieve goals like maximizing flow or minimizing cost. A practical example is data packet routing in a computer network, where bandwidth constraints must be respected while ensuring the fastest data transfer from source to destination.
- **Graph Coloring Problem:** This problem aims to assign colors to nodes in a graph such that no two connected nodes share the same color, with the aim of minimizing the total number of colors used. An example application is exam scheduling, where each course is a node and an edge indicates a shared student; the goal is to assign exam times (colors) so no student has overlapping exams.

Although graph structures ground both problems, their goals and constraints differ. The network flow problem focuses on *capacity management and path optimization* across edges, focusing on how much flow can be pushed across edges. In contrast, the graph coloring problem deals with *conflict mitigation and partitioning*, focusing on how nodes can be grouped or separated under adjacency restrictions.

5.1 Network Flow Problems

Optimize the flow of goods, data, or resources through a network from source nodes to destination nodes while respecting capacity constraints on the edges. For instance, a minimum cost flow model is used to model the transmission of electricity where the goal is to route power from generating stations to cities across transmission lines minimizing transmission losses or costs. In internet data routing, on the other hand, maximum flow models are used

to maximize the data flow from a source server to a destination without exceeding bandwidth capacity on network links.

5.1.1 Maximum Flow

The Maximum Flow problem seeks to find the maximum amount of flow that can be sent from a source node to a sink node through a network while respecting capacity constraints on each arc. It ensures flow conservation at each node and is solved using algorithms like Ford–Fulkerson [18] or Edmonds–Karp [19]. Applications of maximum flow include optimizing data throughput in telecommunication networks, managing traffic flow, maximizing goods shipment in logistics etc.

Given:
Consider a directed graph $G = (V, E)$, where V is the set of nodes and E is the set of directed edges. Each edge $(i, j) \in E$ has an associated non-negative capacity $c(i, j)$, representing the maximum flow that can pass through it. We are also given a source node $s \in V$ and a sink node $t \in V$, and the goal is to determine how much flow can be sent from s to t without violating the capacity limits.

Decision Variables:
The decision variable for the problem is $f(i, j)$ which denote the amount of flow sent along edge (i, j).

Mathematical Formulation:
The objective is to maximize the total flow from the source node s to the sink node t while satisying the constraints. Equation 5.2 enforces flow conservation at every intermediate node, i.e., the amount of flow entering the node must equal the amount leaving it (excluding the source and sink). Equation 5.3 ensures that the flow on any edge does not exceed its specified capacity and guarantees that no negative flow is allowed on any edge.

$$\text{Maximize} \quad \sum_{(s,j)\in E} f(s, j) \tag{5.1}$$

$$\text{Subject to} \quad \sum_{j:(j,i)\in E} f(j, i) = \sum_{j:(i,j)\in E} f(i, j) \qquad \forall i \in V \setminus \{s, t\} \tag{5.2}$$

$$0 \leq f(i, j) \leq c(i, j) \qquad \forall (i, j) \in E \tag{5.3}$$

Example:
Suppose we have a network with 5 nodes labeled from 0 to 4. The set of edges is defined using two lists: ‘Start nodes‘ and ‘End nodes‘. Each pair start[i], end[i] defines a directed edge from node start[i] to node end[i]. The list ‘Capacities‘ specifies the maximum allowable flow on each corresponding edge. Each entry in these lists corresponds to a directed edge. For example, edge $(0 \rightarrow 1)$. The network has node 0 as the source and node 4 as the sink.

- Start nodes: [0, 0, 0, 1, 1, 2, 2, 3, 3]
- End nodes: [1, 2, 3, 2, 4, 3, 4, 2, 4]
- Capacities: [20, 30, 10, 40, 30, 10, 20, 5, 20]

Solution: On solving the problem using PuLP, the maximum flow that can be sent from node 0 to node 4 is 60 units. Table 5.1 and Figure 5.1 shows the optimal allocation of flow in the network.

Edge	Flow	Capacity
$0 \rightarrow 1$	20	20
$0 \rightarrow 2$	30	30
$0 \rightarrow 3$	10	10
$1 \rightarrow 2$	0	40
$1 \rightarrow 4$	20	30
$2 \rightarrow 3$	10	10
$2 \rightarrow 4$	20	20
$3 \rightarrow 2$	0	5
$3 \rightarrow 4$	20	20

Table 5.1: Optimal Flow Allocation

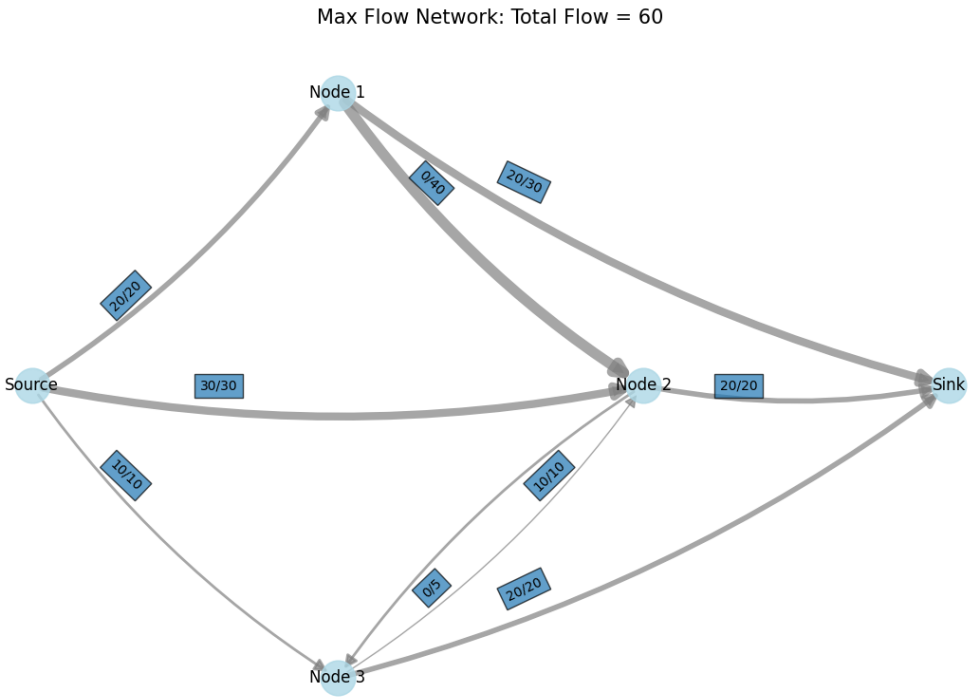

Figure 5.1: Network representation of the solution

Applications:

- **Traffic Routing:** Maximize the number of cars that can flow through a traffic network from an entry point to a destination during rush hour. Roads have limited capacities, and by modeling the traffic network as a flow graph, planners can identify bottlenecks and re-route traffic to improve flow.
- **Telecommunications Network Bandwidth:** Maximizing the amount of data transmitted between two servers through a network with limited bandwidth on each link.

- **Water Distribution:** Maximize the amount of water that can flow from a reservoir to cities through a network of pipes with flow capacities.

Solver Comparison:
We performed a similar benchmarking analysis for the Maximum Flow problem by generating random networks with varying numbers of nodes and randomly assigned capacities (maximum allowable flow) on the links. The performance of three solvers: PuLP, Gurobi, and Google OR-Tools was evaluated by measuring their runtime as the network size increased. Figure 5.2 shows a comparative overview of how each solver performed in terms of computation time.

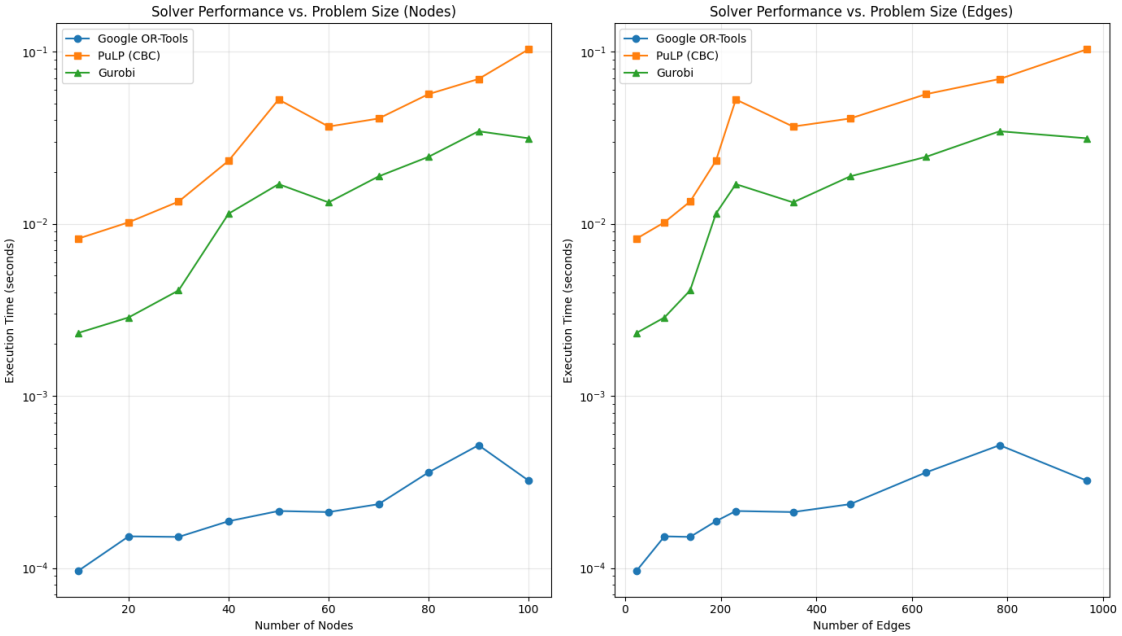

Figure 5.2: Solver Comparison for Maximum Flow problem

Results:
The solver comparison plots for the Maximum Flow problem reveal the following key observations:

- Although Gurobi showed strong performance on resource allocation and scheduling problems, it required more time than Google OR-Tools for solving maximum flow instances
- Google OR-Tools computationally outperformed other solvers in solving the problems.
- Similar to the previous problems, PuLP is taking the most time to solve the maximum flow problem.

5.1.2 Minimum Cost Flow

The minimum cost flow problem involves finding the cheapest way to send a certain amount of flow through a network while respecting capacity constraints on the arcs of the network. This problem is widely used in logistics and transportation systems, where the goal is to minimize shipping costs while meeting supply and demand at various locations. It also finds applications in energy distribution networks and telecommunication routing, where flow must be allocated efficiently across constrained infrastructure.

Given:
We are given a directed network $G = (V, A)$, where V is the set of nodes (or vertices) and A is the set of directed arcs connecting pairs of nodes. Each arc $(i, j) \in A$ is associated with two parameters: a non-negative cost per unit of flow c_{ij}, and a non-negative capacity (upper bound) u_{ij} representing the maximum amount of flow that can pass through the arc. Additionally, each node $i \in V$ has an associated value b_i that indicates its net supply or demand. Specifically, if $b_i > 0$, node i is a supply node ; if $b_i < 0$, node i is a demand node ; and if $b_i = 0$, node i is a transshipment node, through which flow can pass but which neither supplies nor consumes flow.

Decision Variables:
The decision variable for this problem is x_{ij}, representing the amount of flow sent from node i to node j along arc (i, j).

Mathematical Formulation:
The objective is to minimize the total cost of transporting flow through the network. The total cost is the sum of flow on each arc multiplied by the corresponding cost per unit of flow. The constraints ensure that flow is conserved at each node by balancing incoming and outgoing flows according to supply or demand (eq. 5.5), and that the flow on each arc respects the given capacity limits (eq. 5.6).

$$\text{Minimize} \quad \sum_{(i,j)\in A} c_{ij} \cdot x_{ij} \tag{5.4}$$

$$\text{Subject to} \quad \sum_{j:(i,j)\in A} x_{ij} - \sum_{j:(j,i)\in A} x_{ji} = b_i \qquad \forall i \in V \tag{5.5}$$

$$0 \leq x_{ij} \leq u_{ij} \qquad \forall (i,j) \in A \tag{5.6}$$

Example: Consider a network with 6 nodes. Node 0 represents the source node with supply of 15 units and node 5 is the sink node with a demand of 15 units. All the other four nodes are transshipment nodes. The arcs, their capacity and the cost per unit flow are given in Table 5.2.

Arc	Capacity	Cost per unit flow
(0, 1)	5	2
(0, 2)	10	1
(1, 2)	5	1
(1, 3)	10	3
(2, 3)	5	2
(2, 4)	8	4
(3, 4)	7	1
(3, 5)	10	5
(4, 5)	12	2

Table 5.2: Arc Capacities and Costs

Node	Supply/Demand
0	+15 (Source)
1	0
2	0
3	0
4	0
5	-15 (Sink)

Table 5.3: Node Supply and Demand

Solution: The problem was solved using the PuLP solver and the optimum cost found was 111 with a flow of 15 units. Table 5.4 and Figure 5.3 shows the arc flow for the network's minimum cost optimal solution.

Arc	Flow	Capacity
$0 \rightarrow 1$	5.0	5
$0 \rightarrow 2$	10.0	10
$1 \rightarrow 3$	5.0	10
$2 \rightarrow 3$	5.0	5
$2 \rightarrow 4$	5.0	8
$3 \rightarrow 4$	7.0	7
$3 \rightarrow 5$	3.0	10
$4 \rightarrow 5$	12.0	12

Table 5.4: Optimal Flow on Each Arc

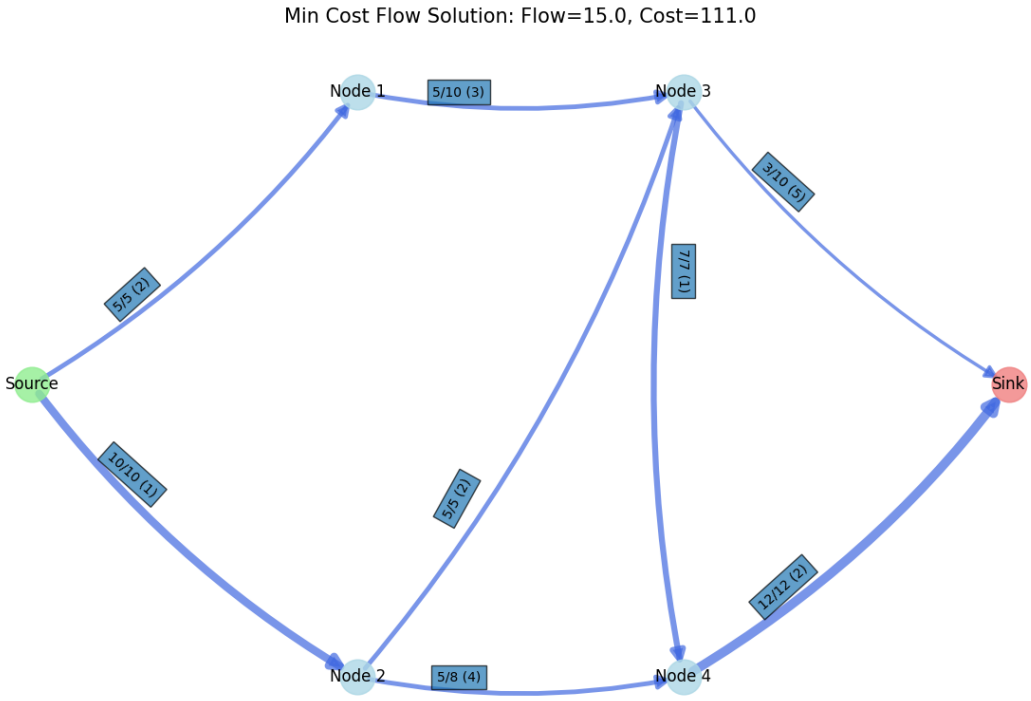

Figure 5.3: Network representation of the solution

Applications:

- **Cash Flow Management** Banks face the challenge of moving funds between branches to meet demand. The minimum cost flow problem helps manage these transfers in such a way that the operational and transaction costs are minimized, while still meeting the required cash demands at each branch.
- **Logistics and Supply Chain Optimization:** A retail company has to deliver products from multiple warehouses to retail stores, ensuring that the transportation routes are optimized to minimize costs while meeting demand at each retail location.
- **Electricity Grid Optimization:** Transmit electricity from power stations to cities while minimizing power loss and distribution cost.

Solver Comparison:
we conducted a scaling study using synthetically generated network instances to evaluate the performance of different solvers on the Minimum Cost Flow problem. The networks were constructed by randomly creating links between nodes, with random capacities and costs per unit flow assigned to each arc. A subset of nodes was randomly selected as supply (source) and demand (sink) nodes, with random supply and demand values assigned such that the total supply equaled total demand. The performance of three solvers: PuLP, Gurobi, and Google OR-Tools were evaluated by measuring runtime as we scaled the size of the network. Figure 5.4 presents a comparative analysis of runtime performance across the solvers.

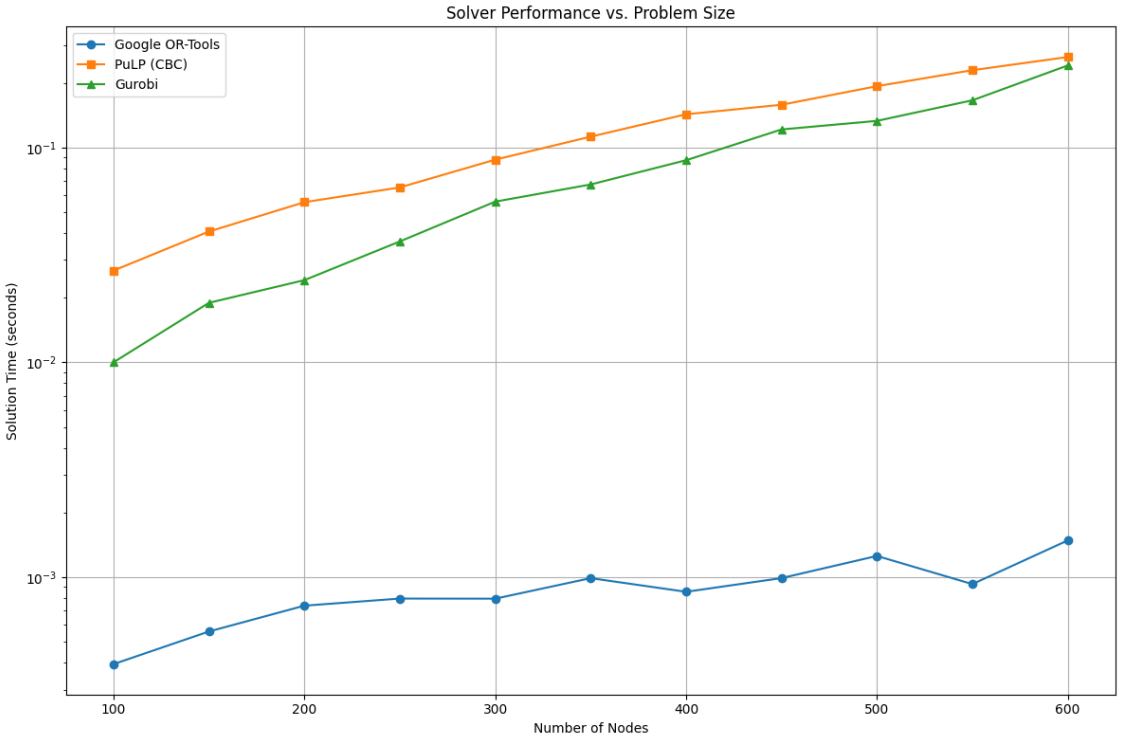

Figure 5.4: Solver Comparison for Minimum Cost problem

Results: The solver comparison plots for the Minimum Cost Flow problem reveal the following key trends:

- Consistent with the results from the Maximum Flow problem, Google OR-Tools demonstrated superior computational performance, while PuLP exhibited the slowest performance across all tested sizes
- For larger problem sizes, Gurobi and PuLP took approximately similar time to solve the problems.

5.2 Graph Coloring Problem

The graph coloring problem involves assigning colors to the vertices of a graph such that no two connected vertices share the same color, while minimizing the number of colors used.

Given:
We are given an undirected graph $G = (V, E)$, where V is the set of vertices (nodes) and $E \subseteq$

$\{\{u, v\} \mid u, v \in V, u \neq v\}$ is the set of edges, each representing a connection between a pair of vertices. Additionally, we are provided a finite set of k colors denoted as $C = \{1, 2, \ldots, k\}$, which are available for coloring the vertices.

Decision Variables:
We define binary decision variables to model the graph coloring problem. The variable $x_{v,c} \in \{0, 1\}$ indicates whether vertex $v \in V$ is assigned color $c \in C$; it takes the value 1 if the color is assigned, and 0 otherwise. To keep track of which colors are actually used in the solution, we introduce an additional variable $y_c \in \{0, 1\}$, which is set to 1 if color c is used to color any vertex in the graph, and 0 otherwise.

Mathematical Formulation:
The objective is to minimize the total number of colors used in the graph coloring. This is done by summing over all binary variables y_c that indicate whether a color $c \in C$ is used. The constraints ensure that each vertex is assigned exactly one color (eq. 5.8), and that no two adjacent vertices share the same color (eq. 5.9).

$$\text{Minimize} \quad \sum_{c \in C} y_c \tag{5.7}$$

$$\text{Subject to} \quad \sum_{c \in C} x_{v,c} = 1 \qquad \forall v \in V \tag{5.8}$$

$$x_{u,c} + x_{v,c} \leq y_c \qquad \forall (u, v) \in E,\ \forall c \in C \tag{5.9}$$

$$x_{v,c} \in \{0, 1\},\ y_c \in \{0, 1\} \qquad \forall v \in V,\ \forall c \in C \tag{5.10}$$

Example:
Let us consider a graph with 5 vertices. The connections between these vertices are represented using an adjacency matrix. In the matrix, a value of 1 at position (i, j) indicates that there is an edge between vertex i and vertex j, while a 0 indicates no direct connection. For instance, vertex 0 is connected to vertices 1, 2, and 3 as seen in the first row of the matrix. This matrix representation is symmetric since the graph is undirected.

- Adjacency Matrix:

$$\begin{bmatrix} 0 & 1 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 \\ 1 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 \end{bmatrix}$$

Solution: The graph coloring problem was solved using three solvers: PuLP, Google OR-Tools, and Gurobi. All solvers returned the same optimal solution, indicating that a minimum of 3 colors is required to color the graph such that no two adjacent vertices share the same color. The colored graph representation of the solution is shown in Figure 5.5.

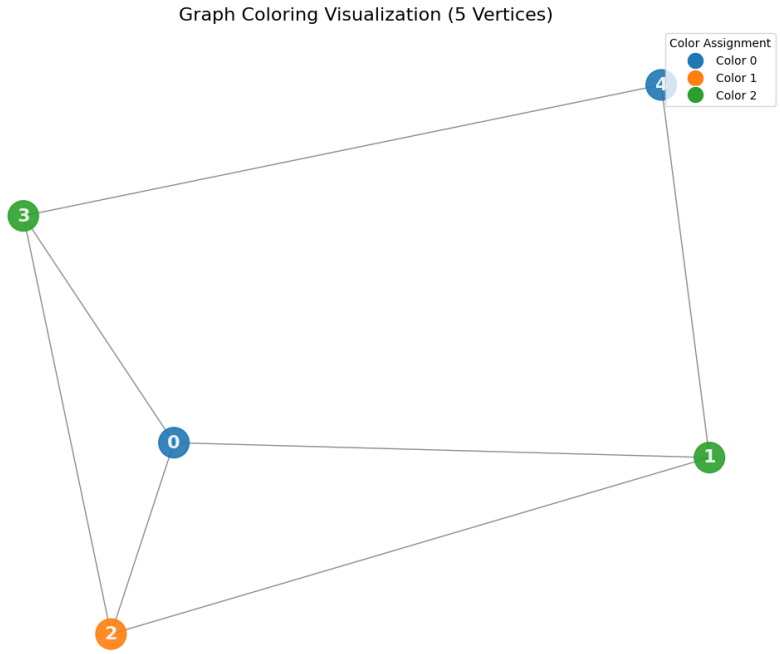

Figure 5.5: Graph Coloring Solution with 3 Colors

Applications:

- **Exam Timetabling:** In universities, exam scheduling can be modeled as a graph coloring problem, where each exam is a node and an edge connects two exams if at least one student is enrolled in both. This ensures that no student has overlapping exams.
- **Frequency Assignment:** Assigning frequencies to transmitters such that nearby transmitters do not interfere with each other is critcal in wireless communication systems. This can be modeled as a graph where each node is a transmitter, and edges connect transmitters within interference range.
- **Register Allocation:** In compiler design, variables that are live at the same time must be stored in different CPU registers. This can be modeled as a graph where nodes represent variables and edges represent interference (simultaneous usage). Coloring this graph helps allocate registers efficiently by ensuring conflicting variables are stored in different registers.

Solver Comparison:
A similar benchmarking process was conducted for the graph coloring problem. For each instance, a random network was generated and represented using an adjacency matrix, which served as the input for the solvers. We evaluated the performance of three solvers: PuLP, Gurobi and Google OR-Tools. For each solver, the runtime was measured as the number of nodes/vertices increased. Figure 5.6 presents a comparison of runtime performance across the different solvers.

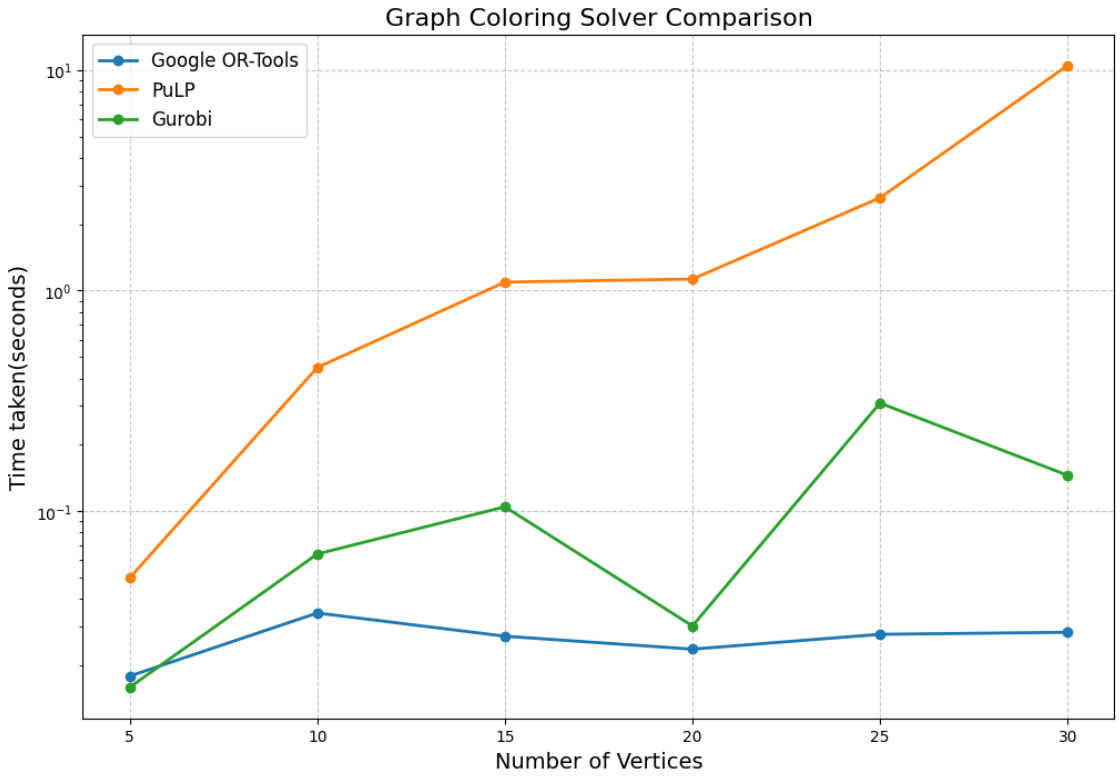

Figure 5.6: Solver Comparison for Graph Coloring problem

Results:
Upon analysis of the solver comparison plots for the Graph Coloring problem, the following general trends were observed.

- Across all graph-based problems, a consistent trend was observed: Google OR-Tools consistently outperformed the other solvers in terms of computational efficiency, while PuLP required the most time to solve these problems.
- Although Gurobi showed exceptional performance in resource allocation and scheduling problems, its performance was only moderate when applied to the graph coloring problem.

Chapter 6

Routing Problems

Consider a logistics firm in charge of a fleet of delivery trucks assigned to visit hundreds of customers throughout a city. The company must determine which vehicles serve which customers and also in what order, all the while making sure that delivery deadlines are fulfilled, routes are effective, and vehicles don't exceed capacity. This kind of problems lie at the core of routing optimization.

Routing problems are concerned with the best movement of vehicles, products, or agents through a network in order to reach a specific set of locations under logistical limitations. These issues are crucial to transportation, distribution, ride-sharing, and even robotics.

Routing problems share several key characteristics: they require finding the optimal sequence in which a set of locations should be visited, minimize operational metrics such as total distance, travel time, and satisfy the constraints like vehicle capacity, satisfy customer demands, or specific service time windows.
Two widely studied problems in this domain are:

- **Traveling Salesman Problem (TSP):** A classic optimization problem where a single agent must visit a set of locations exactly once and return to the starting point while minimizing the total travel cost. For example, a postman has to visit a set of locations to deliver the letters and return to the office with minimal travel.
- **Capacitated Vehicle Routing Problem (CVRP):** This is a generalization of the TSP where multiple vehicles cater to customer demands without exceeding vehicle capacities. An example includes a grocery delivery service optimizing its delivery fleet to satisfy customer orders within a locality while ensuring vehicles are not overloaded.

While both problems aim to determine the most efficient routes, they differ in complexity. TSP involves a single route for one agent, whereas CVRP includes multiple routes with additional constraints on capacity and distribution across vehicles.

6.1 Traveling Salesman Problem (TSP)

The Traveling Salesman Problem (TSP) is a classic optimization problem in which a salesman must visit a set of cities exactly once and return to the starting point, minimizing the total travel distance or cost. This problem is critical in logistics and route planning, where minimizing travel time or cost is of utmost importance such as a delivery driver optimizing to loop through all assigned delivery locations.

Given:
Let n be the number of cities. For each pair of cities $i, j \in \{1, 2, \ldots, n\}$, let d_{ij} denote the distance or cost of traveling from city i to city j.

Decision Variables:
We define binary decision variables x_{ij}, where $x_{ij} = 1$ if the tour goes directly from city i to city j, and $x_{ij} = 0$ otherwise.

Mathematical Formulation:
The goal is to minimize the total travel cost of visiting each city exactly once and returning to the starting point. The constraints ensure that each city is entered and left exactly once (eqs. 6.2 and 6.3), and that no subtours (smaller loops excluding the full tour) are formed (eq. 6.4). The decision variables are binary, indicating whether a direct path between two cities is included in the tour or not (eq. 6.5).

$$\text{Minimize} \quad \sum_{i=1}^{n} \sum_{j=1}^{n} d_{ij} x_{ij} \tag{6.1}$$

$$\text{Subject to} \quad \sum_{j=1, j \neq i}^{n} x_{ij} = 1 \quad \forall i \in \{1, \ldots, n\} \tag{6.2}$$

$$\sum_{i=1, i \neq j}^{n} x_{ij} = 1 \quad \forall j \in \{1, \ldots, n\} \tag{6.3}$$

$$\text{Subtour elimination constraints (e.g., MTZ formulation)} \tag{6.4}$$

$$x_{ij} \in \{0, 1\} \quad \forall i, j \in \{1, \ldots, n\}, i \neq j \tag{6.5}$$

Example: Let us consider a Traveling Salesman Problem with five cities. The coordinates define city locations in a 2D space, and the corresponding distance matrix gives the cost of traveling (Euclidean distance) between each pair of locations. The nodes and the distance matrix are given below.

- nodes = { 0: (0, 0), 1: (1, 5), 2: (5, 3), 3: (6, 6), 4: (8, 2) }
- Distance Matrix

$$\begin{bmatrix} 0.00 & 5.10 & 5.83 & 8.49 & 8.25 \\ 5.10 & 0.00 & 4.47 & 5.10 & 7.62 \\ 5.83 & 4.47 & 0.00 & 3.16 & 3.16 \\ 8.49 & 5.10 & 3.16 & 0.00 & 4.47 \\ 8.25 & 7.62 & 3.16 & 4.47 & 0.00 \end{bmatrix}$$

Solution: On solving the problem using PuLP, the optimal route was found with a total travel distance of 23.66 units. The optimal route is visualized in Figure 6.1

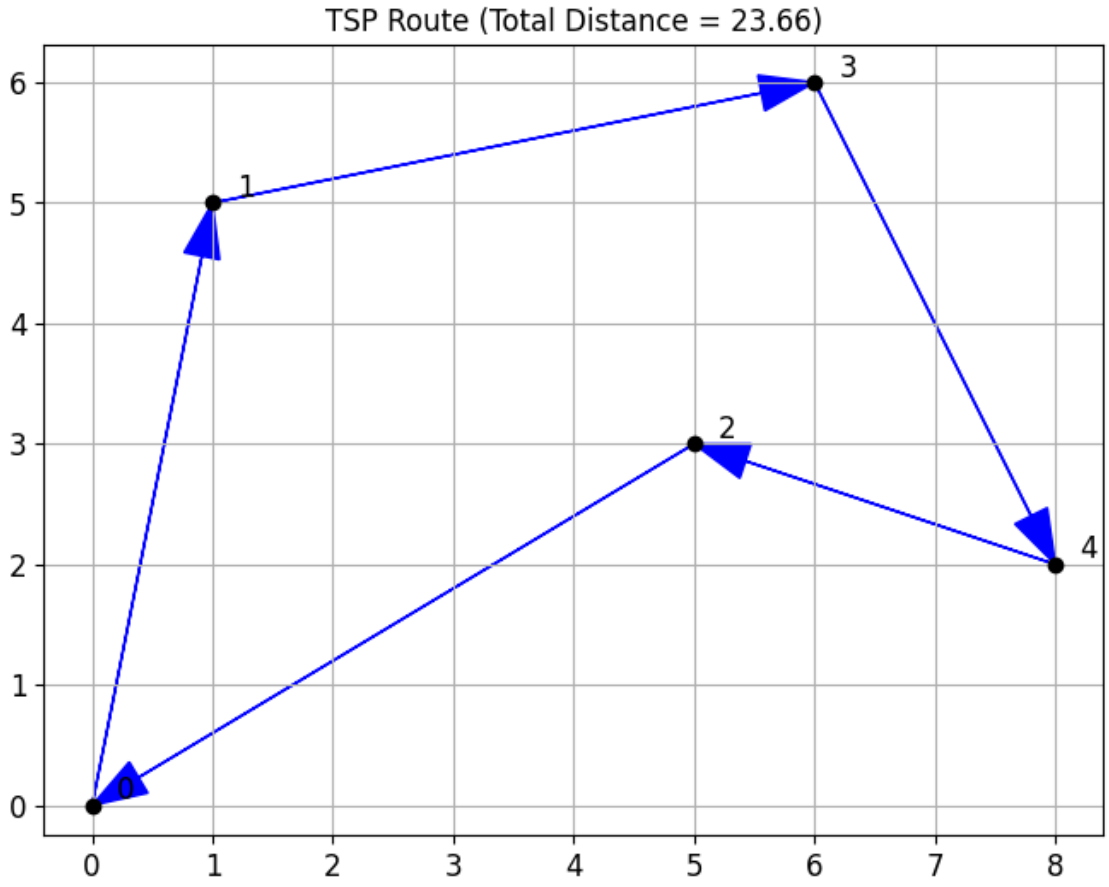

Figure 6.1: TSP example solution

Applications:

- **Logistics and Delivery Routing:** Delivery companies must plan efficient routes for trucks to visit multiple destinations. Solving the TSP helps in minimizing travel distance, fuel consumption, and delivery time, leading to cost effective operations.
- **Manufacturing Optimization:** In circuit board manufacturing, optimizing the order in which holes are drilled in a circuit board results in reduced movement time of the drill head, reduced production time and wear on machinery.
- **Robotics Path Planning:** In automated warehouses or factories, robots perform tasks at various stations. Determining the optimal path for robots to perform tasks at various locations in a factory results in minimizing travel time and battery usage.

Solver Comparison:
To evaluate the performance of different solvers on the Traveling Salesman Problem, a similar benchmarking process was conducted using synthetic instances with increasing numbers of cities. For a particular number of cities, a set of nodes were created and the corresponding distance matrix was also generated. For a given problem size, each of the solvers: PuLP, Gurobi, and Google OR-Tools was tested on identical problem inputs, and the runtime required to find an optimal tour was recorded. Figure 6.2 presents a comparison of runtime performance across the different solvers.

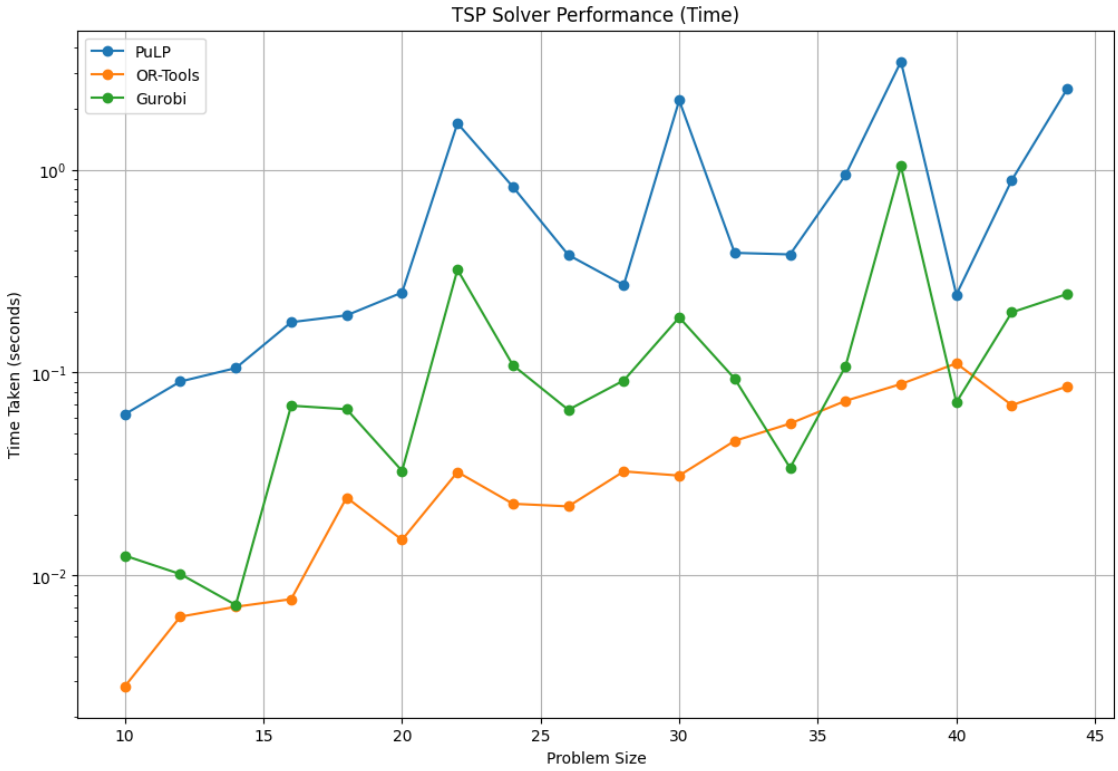

Figure 6.2: Solver Comparison for Traveling Salesman Problem

Results:
Upon analysis of the solver comparison plots for the Traveling Salesman problem, the following general trends were observed.

- Google OR-Tools computationally outperformed other solvers.
- PuLP took computationally the most time to solve the problems.
- PuLP and Gurobi showed high variability in runtime across simulations while Google OR-Tools maintained a smoother growth in computation time as problem size scaled.

6.2 Capacitated Vehicle Routing Problem (CVRP)

The Capacitated Vehicle Routing Problem (CVRP) involves minimizing the total cost (e.g., distance, time, or fuel) of serving a set of customers with known demands, using a fleet of vehicles starting and ending at a depot, without exceeding the vehicle capacity.

Given:
Let N be the set of customers and V be the set of vehicles. Each customer $i \in N$ has a demand d_i, and each vehicle has a fixed capacity Q. Let c_{ij} denote the cost (e.g., distance, time, or fuel) of traveling from node i to node j.

Decision Variables:
We define binary decision variables x_{kij}, where $x_{kij} = 1$ if vehicle $k \in V$ travels directly from node i to node j, and 0 otherwise.

Mathematical Formulation:
The objective function minimizes the total routing cost incurred by all vehicles while serving the customers. Each customer must be visited exactly once by any vehicle (eq. 6.7), and flow conservation is maintained for each vehicle across the network (eq. 6.8). The total demand served by any vehicle must not exceed its capacity Q (eq. 6.9).

$$\text{Minimize} \quad \sum_{k \in V} \sum_{i \in N} \sum_{j \in N} c_{ij} \cdot x_{kij} \tag{6.6}$$

$$\text{Subject to} \quad \sum_{k \in V} \sum_{j \in N} x_{kij} = 1 \quad \forall i \in N \setminus \{0\} \tag{6.7}$$

$$\sum_{i \in N} x_{kij} = \sum_{l \in N} x_{kjl} \quad \forall k \in V,\ \forall j \in N \tag{6.8}$$

$$\sum_{i \in N} d_i \sum_{j \in N} x_{kij} \leq Q \quad \forall k \in V \tag{6.9}$$

$$x_{kij} \in \{0, 1\} \quad \forall k \in V,\ \forall i, j \in N \tag{6.10}$$

Example:
Consider a Capacitated Vehicle Routing Problem (CVRP) with 5 nodes including depot (node 0) and 2 vehicles. The following distance matrix represents the cost of travel between the nodes. Each customer has a specific demand, and each vehicle has a capacity of 5 units.

- Distance Matrix:

$$\begin{bmatrix} 0 & 72 & 74 & 50 & 47 \\ 72 & 0 & 41 & 33 & 37 \\ 74 & 41 & 0 & 2 & 93 \\ 50 & 33 & 2 & 0 & 77 \\ 47 & 37 & 93 & 77 & 0 \end{bmatrix}$$

- Demand: [0, 2, 2, 2, 1]

Solution: On solving the CVRP using Google OR-Tools, the optimal total distance covered by the two vehicles is 282 units. The vehicles are assigned the following routes that satisfy all capacity and routing constraints:

Vehicle	Route	Load	Distance (m)
0	$0 \to 3 \to 2 \to 0$	4	126
1	$0 \to 4 \to 1 \to 0$	3	156

Table 6.1: CVRP Example Solution with Two Vehicles

Applications:

- **Last-mile Delivery:** Logistics companies often face the challenge of delivering packages from distribution centers to customers. CVRP helps optimize delivery routes to reduce fuel consumption, delivery time, and operational costs.

- **Food Distribution:** A food distribution company must transport items to multiple destinations while minimizing the delivery time and avoiding spoilage. Solving CVRP ensures cost-effective and timely food distribution.
- **Ride-sharing Services:** A ride sharing company aims to assign passengers to vehicles and determine optimal pickup and drop off sequences. The CVRP framework assists in minimizing total travel time and maximizing vehicle utilization.

Solver Comparison:
A similar benchmarking experiment was conducted using synthetic instances with increasing numbers of cities to evaluate the performance of different solvers on CVRP. For a particular number of cities, a set of nodes was created, the corresponding distance matrix was also generated, and demand values were assigned to the customers. For a given problem size, each of the solvers: Gurobi, and Google OR-Tools was tested on identical problem inputs, and the runtime required to find an optimal tour was recorded. Figure 6.3 presents a comparison of runtime performance across the different solvers.

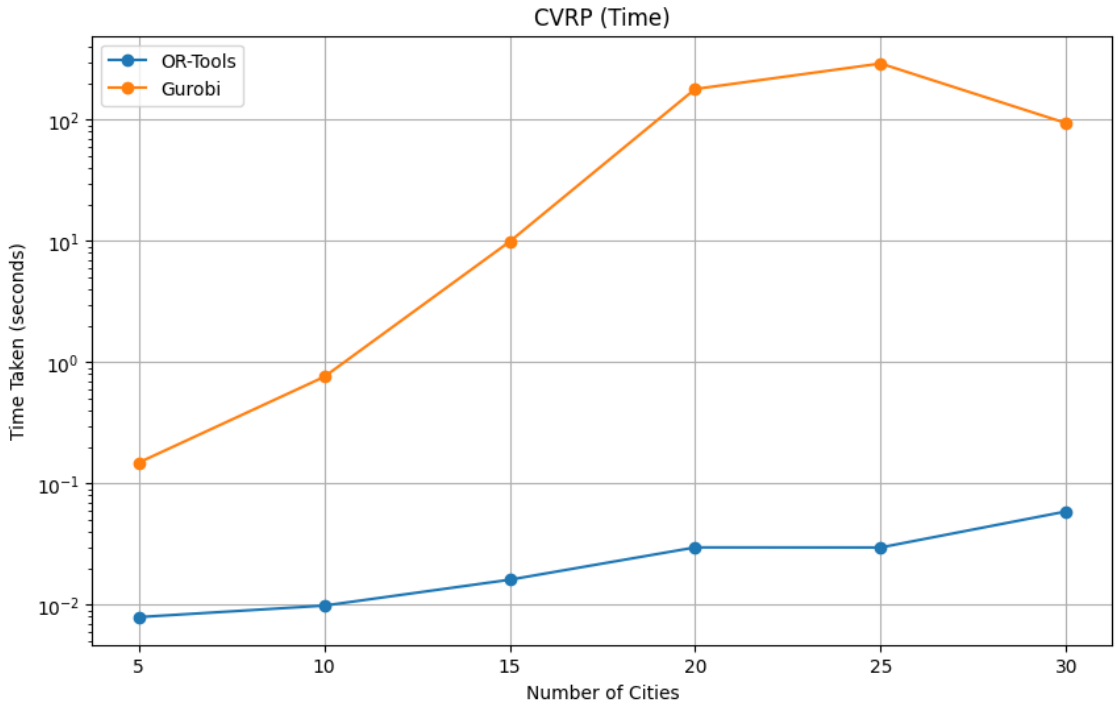

Figure 6.3: Solver Comparison for CVRP

RL4CO Approach:

Combinatorial optimization problems (such as TSP and CVRP) are hard because the number of possible solutions grow non-linearly with problem size. Classical solvers can be slow or inflexible. RL learns policies that can build or improve solutions efficiently and generalize across instances, making it powerful for large-scale real-world combinatorial optimization tasks.

RL4CO is a unified benchmark and framework designed for training and evaluating reinforcement learning (RL) algorithms specifically tailored to combinatorial optimization (CO) problems[11]. It supports both constructive methods, which build solutions from scratch, and improvement methods, which refine existing solutions. RL4CO is built on modern machine learning libraries such as TorchRL, PyTorch Lightning, Hydra, and TensorDict, enabling efficient large-scale training on GPUs.

Training Process on VRPLib:
The VRP library[1] is a specialized Python library that solves a variety of Vehicle Routing Problems (VRP), including conventional VRP, Capacitated VRP (CVRP), and more sophisticated versions such as Time Windows and Pickup and Delivery. It offers a high-level interface for defining problem instances, configuring constraints like vehicle capacity or route length, and integrating solvers such as Google OR-Tools. The library is especially valuable for researchers and practitioners working on logistics and transportation optimization since it allows for quick development and comparison of various routing options with minimal coding effort. The training process using RL4CO on VRPLib instances involves several key steps:

- **Environment Setup:** VRP instances are generated or loaded using VRPLib datasets. The environment defines the problem constraints (like vehicle capacity, demands, etc.) and tracks solution construction.
- **Policy Selection:** A neural network model (e.g. Attention Model, Policy Optimization with Multiple Optima (POMO)) is chosen to learn a policy that sequentially decides actions, such as selecting the next customer to visit.
- **Algorithm Choice:** Reinforcement learning algorithms such as REINFORCE or PPO are used to train the model. The policy is updated based on reward signals derived from the quality of the solution.
- **Training Execution:** The training is managed through the `RL4COTrainer` class. The trainer is configured to control key aspects such as the number of epochs, device settings (e.g., using a GPU), and logging.

Instance	No. of Nodes	No. of Vehicles	BKS
A-n53-k7	53	7	1010
A-n54-k7	54	7	1167
A-n55-k9	55	9	1073
A-n60-k9	60	9	1354
A-n61-k9	61	9	1034
A-n62-k8	62	8	1288
A-n63-k9	63	9	1616
A-n63-k10	63	10	1314
Li_30	1041	10	31742.64
Li_31	1121	10	34330.94
Li_32	1201	11	37159.41

Table 6.2: CVRP Instances with No: of Nodes, Vehicles, and Best Known Solution(BKS)

RL4CO, Google OR-Tools solutions for some instances
The RL model was trained on CVRP instances with 50 customer nodes using the environment `CVRPEnv(generator_params= {'num_loc': 50})`. After training, the model was evaluated on instances with larger number of nodes. Table 6.3 shows the comparison between the solutions obtained by the trained RL model and Google OR-Tools.

Instance	No. of Nodes	BKS	OR Solution	RL Solution	RL Time(sec)	OR Time(sec)
A-n53-k7	53	1010	1098	1172	0.1458	0.1752
A-n54-k7	54	1167	1226	1338	0.1385	0.1747
A-n55-k9	55	1073	1148	1212	0.2013	0.1397
A-n60-k9	60	1354	1494	1567	0.2210	0.1836
A-n61-k9	61	1034	1189	1244	0.2193	0.1103
A-n62-k8	62	1288	1491	1503	0.2147	0.1069
A-n63-k9	63	1616	1799	1812	0.2051	0.1165
A-n63-k10	63	1314	1400	1437	0.2113	0.1011
A-n64-k9	64	1401	1477	1575	0.2499	0.1389
A-n65-k9	65	1174	1258	1347	0.2440	0.1365
A-n69-k9	69	1159	1203	1372	0.2634	0.1563
A-n80-k10	80	1763	1915	2108	0.2762	0.2484

Table 6.3: Comparison of Google OR-Tools and RL Solutions on Selected Instances

Gap (%) with Varying Training Node Sizes

Table 6.4 shows the percentage gap between the solution of the RL model and the best known solution (BKS) when the model is trained with different numbers of nodes. We have considered the cases when the model is trained on instances with 50, 60, 70 and 80 nodes.

Problem	Gap % (50)	Gap % (60)	Gap % (70)	Gap % (80)
A-n53-k7	16.04	14.55	20.69	24.55
A-n54-k7	14.65	12.85	11.14	16.54
A-n55-k9	12.95	16.50	12.12	11.74
A-n60-k9	15.73	17.28	11.23	16.84
A-n61-k9	20.31	23.40	17.70	18.47
A-n62-k8	16.69	14.21	18.01	13.59
A-n63-k9	12.13	11.63	15.22	17.51
A-n63-k10	9.36	16.74	13.47	15.83
A-n64-k9	12.42	16.13	8.21	12.42
A-n65-k9	14.74	17.04	18.48	15.33
A-n69-k9	18.38	17.60	14.93	17.69
A-n80-k10	19.57	17.87	14.01	18.38

Table 6.4: Gap(%) v/s Training node size

For instances with larger node sizes,Table 6.5 shows the result obtained.

Instance	No. of nodes	BKS	Gap % (50)	Gap % (100)
Li_30	1041	31742.64	665.30	392.44
Li_31	1121	34330.94	724.33	419.74
Li_32	1201	37159.41	779.59	468.35

Table 6.5: Gap(%) v/s Training node size - for larger node sizes

Gap (%) with Varying Number of Samples (Solutions) generated

Table 6.6 shows the impact of increasing the number of generated solutions on the performance

(Gap %) for various instances. We have discussed the cases where the model generated 64, 128, 256, 512 and 1024 solutions.

Instance	gap_64	gap_128	gap_256	gap_512	gap_1024
A-n53-k7	15.60	15.97	15.53	12.58	13.67
A-n54-k7	18.00	15.99	18.26	10.09	10.54
A-n55-k9	10.73	8.72	7.92	7.24	7.33
A-n60-k9	10.32	7.21	8.01	8.92	7.22
A-n61-k9	16.60	14.58	14.61	12.85	11.32
A-n62-k8	17.19	14.39	12.69	12.72	13.39
A-n63-k9	12.63	12.17	10.96	11.91	11.04
A-n63-k10	13.23	13.80	13.83	12.95	12.22
A-n64-k9	16.67	13.31	11.55	13.92	13.43
A-n65-k9	15.41	14.12	15.63	13.49	13.99
A-n69-k9	16.22	16.38	14.18	14.52	13.83
A-n80-k10	18.44	18.16	16.13	14.83	12.64

Table 6.6: Gap(%) v/s No. of solutions generated

Results:
Upon analysis of the solver comparison plots for the Capacitated Vehicle Routing Problems including results from the RL4CO approach and the experiments conducted on them, the following general trends were observed.

- Google OR-Tools computationally outperformed Gurobi solvers.
- RL4CO took computationally less time to solve the problems but with a compromise on the solution quality.
- As the RL4CO model was trained on instances with a larger number of nodes, the solution gap (%) notably decreased for problems of similar or larger sizes.
- Increasing the number of solutions generated by the RL4CO model led to a significant reduction in the solution gap.

Chapter 7

Results and Discussion

This project focused on comparison of traditional solvers, metaheuristic methods and ML based methods across various combinatorial optimization problems.The solvers benchmarked include PuLP, Gurobi, Google OR-Tools, and CP-SAT. Problems considered belong to the broad categories of resource allocation and scheduling, financial and packing optimization, graph based problems and routing problems. Each problem was formulated mathematically, implemented using multiple solvers, and tested on synthetically generated datasets to assess scalability and computational efficiency.

7.1 Summary of the Results

In resource allocation and scheduling tasks such as job scheduling and the Assignment Problem Gurobi showed outstanding performance. Although Google OR-Tool performed rather well, its runtime efficiency was sligntly behind Gurobi. Conversely, PuLP showed limited scalability by showing notable performance restrictions, especially as the problem size grew. For financial and packing problems, it was observed that Google OR-Tools outperformed other solvers in terms of computational efficiency. However, Gurobi was the best for Knapsack problem. In the case of all graph based problems, Google OR-Tools consistently performed best, followed by Gurobi, with PuLP taking the longest runtime. The same trend was observed in the Routing problem as well.

In addition to classical solvers, we explored the use of reinforcement learning for combinatorial optimization through the RL4CO framework. Although the RL model generated feasible solutions much faster than Google OR-Tools, especially for larger instances, the quality of the solutions, expressed by the gap from the Best Known Solution (BKS), was consistently lower. However, performance improved with increased training node sizes and sample diversity. This demonstrates that while RL4CO based models might not yet match exact solvers in optimality, they hold promise for rapidly generating near optimal solutions, so valuable for real-time or large-scale applications where speed is vital.

7.2 General Trends and Insights

Across all problem categories, Google OR-Tools consistently demonstrated superior performance in terms of computation time. Its efficiency is particularly visible in network flow and routing problems. Gurobi, although powerful in scheduling and allocation problems, showed only moderate efficiency in routing and graph based problems. PuLP, while accessible and easy to use, lacks the scalability and optimization strength required for large or complex problem

instances. This behavior can be attributed to the underlying optimization engines of these solvers.

Apart from conventional solvers, reinforcement learning-based methods especially RL4CO indicate a promising path for large-scale combinatorial optimization. RL4CO enables neural policies to learn solution strategies straight from problem instances, allowing for fast inference during test time. Though in terms of optimality RL4CO solutions currently lag behind exact solvers like Google OR-Tools, especially for large instances, they have major generalization and rapid approximation advantages. This makes them attractive for real time or resource limited environments where exact optimization may be infeasible.

These trends highlight the importance of selecting an appropriate solver based on the problem type. For scheduling and allocation, Gurobi is highly recommended. For graph-based and routing problems, Google OR-Tools provides a clear advantage. When approximate yet rapid solutions are acceptable or when models must generalize across problem instances, RL4CO and similar learning-based methods are compelling alternatives. PuLP remains suitable for small-scale problems and educational purposes, but is not recommended for large-scale industrial use.

Chapter 8

Conclusions and Future Work

8.1 Conclusions

The benchmarking study offers valuable insights on solver selection for combinatorial optimization problems. Solver performance varies greatly across problem types, and using the appropriate tool can result in substantial reductions in computation time. For practitioners, knowing the nature of the problem and the strength of available solvers is critical for effective problem solving.

In addition to classical solvers, reinforcement learning approaches such as RL4CO provide a new paradigm for dealing with combinatorial problems. By learning policies that construct or refine solutions, RL4CO enables fast inference and flexibility across several problem instances. While the present gap in optimality compared to precise solutions is significant, RL4CO indicates potential scalability and flexibility, notably in routing problems such as CVRP. As neural models and training frameworks improve, learning based solvers may become a useful addition or possibly a replacement for classic optimization techniques in dynamic or large-scale situations.

8.2 Future work

One interesting area for future research is the use of reinforcement learning based frameworks to solve a larger variety of combinatorial optimization problems besides CVRP and TSP. Problems such as the Job Shop Scheduling Problem, Bin packing present further real world difficulties where learning based solutions might be extremely useful, particularly in large-scale or dynamic settings.

Another important area of research is to improve the realism of optimization models. This involves enhancing the CVRP formulation to account for real-world restrictions such as time windows, driver working hour limits, actual traffic changes, and diverse vehicle capacities. Future research can be done into EV routing which includes charging rates, station congestion, battery degradation over time, and smart charging decisions.

Furthermore, exploring hybrid strategies in which RL4CO-generated solutions serve as warm start for classical solvers (e.g., Gurobi, OR-Tools) may provide the best of both worlds: near-optimal performance and rapid solutions. Also, advancements in RL4CO training methodologies, such as curriculum learning, fine-tuning on domain-specific data, and using advanced RL architectures, have the potential to greatly improve both generalization and solution quality while preserving model computational efficiency.

References

[1] "CVRPLib: Capacitated Vehicle Routing Problem Library," http://vrp.galgos.inf.puc-rio.br/index.php/en/.

[2] IBM Corporation, *IBM ILOG CPLEX Optimization Studio*, 2024, version 22.1. [Online]. Available: https://www.ibm.com/products/ilog-cplex-optimization-studio

[3] Gurobi Optimization, LLC, *Gurobi Optimizer Reference Manual*, 2024. [Online]. Available: https://www.gurobi.com

[4] A. O. Makhorin, *GNU Linear Programming Kit, Version 4.65*, 2020. [Online]. Available: https://www.gnu.org/software/glpk/

[5] J. Forrest *et al.*, *CBC (Coin-or branch and cut) Solver*, 2024. [Online]. Available: https://github.com/coin-or/Cbc

[6] S. Mitchell, *PuLP: A Linear Programming Toolkit for Python*, 2024. [Online]. Available: https://coin-or.github.io/pulp/

[7] M. L. Bynum, G. A. Hackebeil, W. E. Hart, C. D. Laird, B. L. Nicholson, J. D. Siirola, J.-P. Watson, and D. L. Woodruff, *Pyomo – Optimization Modeling in Python*, 3rd ed., ser. Springer Optimization and Its Applications. Springer, 2021, vol. 67.

[8] C. Prud'homme, J.-G. Fages, and X. Lorca, "Choco-solver: A Java library for constraint programming," *Journal of Open Source Software*, vol. 7, no. 78, p. 4708, 2022.

[9] P. Laborie, J. Rogerie, P. Shaw, and P. Vilím, "IBM ILOG CP optimizer for scheduling: 20 years of scheduling with constraints at IBM/ILOG," *Constraints*, vol. 23, no. 2, pp. 210–250, 2018.

[10] G. O. Tools, "OR-Tools: Google's Operations Research Tools," 2024, version 9.6. [Online]. Available: https://developers.google.com/optimization

[11] F. Berto, C. Hua, J. Park, L. Luttmann, Y. Ma, F. Bu, J. Wang, H. Ye, M. Kim, S. Choi, N. G. Zepeda, A. Hottung, J. Zhou, J. Bi, Y. Hu, F. Liu, H. Kim, J. Son, H. Kim, D. Angioni, W. Kool, Z. Cao, J. Zhang, K. Shin, C. Wu, S. Ahn, G. Song, C. Kwon, L. Xie, and J. Park, "RL4CO: an Extensive Reinforcement Learning for Combinatorial Optimization Benchmark," *arXiv preprint arXiv:2306.17100*, 2024, https://arxiv.org/abs/2306.17100.

[12] B. S. Baker, E. G. Coffman Jr, and R. L. Rivest, "A new algorithm for the two-dimensional stock cutting problem," *Operations Research Letters*, vol. 1, no. 5, pp. 195–200, 1980.

[13] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, "Optimization by simulated annealing," *Science*, vol. 220, no. 4598, pp. 671–680, 1983.

[14] J. H. Holland, *Adaptation in natural and artificial systems*. University of Michigan Press, 1975.

[15] F. Glover, "Tabu search—part i," *ORSA Journal on Computing*, vol. 1, no. 3, pp. 190–206, 1989.

[16] J. Kennedy and R. Eberhart, "Particle swarm optimization," in *Proceedings of ICNN'95 - International Conference on Neural Networks*, vol. 4. IEEE, 1995, pp. 1942–1948.

[17] M. Dorigo, "Ant colony optimization," Ph.D. dissertation, Université libre de Bruxelles, 1996.

[18] L. R. Ford Jr and D. R. Fulkerson, "Maximal flow through a network," *Canadian journal of Mathematics*, vol. 8, pp. 399–404, 1956.

[19] J. Edmonds and R. M. Karp, "Theoretical improvements in algorithmic efficiency for network flow problems," *Journal of the ACM (JACM)*, vol. 19, no. 2, pp. 248–264, 1972.