Implementation of Dynamic Programming, Greedy, and Branch and Bound Algorithms for Courier Package Scheduling with Capacity and Deadline Constraints

Shannon Aurellius Anastasya Lie - 13523019
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
E-mail: 13523019@std.stei.itb.ac.id, shannonlie23@gmail.com

***Abstract*— Efficient scheduling of courier packages is an essential part in modern logistics but remains challenging due to limitations such as vehicle capacity and tight delivery deadlines. Conventional approaches often fail to sufficiently balance these factors, resulting in inefficiencies and higher operational costs. Efficient scheduling can be applied by using Dynamic Programming, Greedy, and Branch and Bound Algorithm to enhance courier scheduling within these constraints. Their effectiveness is assessed through simulations on various datasets, focusing on scalability, solution quality, and computational demands.**

Keywords—courier scheduling, Dynamic Programming, Greedy Algorithm, Branch and Bound Algorithm, capacity constraints, deadline constraints, optimization.

I. Introduction

In recent years, the logistics and courier delivery segment has seen rapid growth, driven by the expanding dependence on e-commerce and computerized retail. With the surge in online transactions, courier companies are expected to deliver a massive number of packages each day under strict time constraints. The complexity of the issue increases when each delivery must not only meet a due date but also consider the capacity limitations of the available vehicles. Efficient scheduling of courier deliveries where each package has a goal, deadline, and size thus becomes a critical issue to solve, especially as failure to do so can lead to late deliveries, underutilized armadas, and increased operational costs.

Courier scheduling tasks fall under the broader umbrella of combinatorial optimization issues, where a solution must be chosen from a limited but extremely huge set of possibilities. Such issues are typically NP-hard in nature, meaning that finding the exact optimal solution within a reasonable time frame is computationally expensive, especially as the number of deliveries and constraints increase. To overcome this, various algorithmic strategies have been proposed that attempt to supply optimal or near-optimal solutions efficiently.

This paper proposes three approaches to optimizing the courier's revenue by applying Dynamic Programming (DP), Greedy, and Branch and Bound (B&B) algorithms. Each has different strengths and weaknesses depending on the nature of the problem.

Dynamic Programming provides a more structured and reliable way to solve optimization problems that exhibit optimal substructure and overlapping subproblems with two key properties in many scheduling scenarios. By solving smaller subproblems and storing their solutions, DP avoids redundant computations and ensures optimal results for certain classes of problems. In the context of courier scheduling, it can be used to determine the best combination of packages that maximize delivery within capacity and time limits[1].

Branch and Bound is an algorithm that systematically explores all possible configurations of solutions using a tree structure. It evaluates partial solutions and prunes branches of the tree that cannot possibly lead to better outcomes than the current best. While this method is capable of finding the optimal solution, its performance is highly dependent on the quality of the bounding and branching strategies used. For large problem sizes, it can become computationally intensive, although it guarantees correctness where heuristic or greedy methods may fail[2].

These different characteristics become essential to analyse which approach is most appropriate under what conditions. This paper focuses on implementing and comparing these three algorithms Dynamic Programming, Greedy, and Branch and Bound for solving a constrained courier package scheduling problem, where each delivery has a specific deadline and size, and each courier has limited capacity. The goal is to determine how each algorithm performs in terms of solution quality, execution time, and scalability, and to explore the trade-offs between computational cost and scheduling effectiveness in realistic delivery scenarios.

II. LIMITATION

This paper has several limitations that should be acknowledged. Due to the computational complexity of Dynamic Programming and Branch and Bound algorithms, the implementation is limited to handling only a single delivery trip. Supporting multiple trips would extremely expand the solution space and lead to excessive processing time. As a result, the number of packages was restricted to 25 to maintain reasonable performance. Additionally, the dataset used in this paper is entirely synthetic and based on simulations, not derived from real-world logistics operations, which may limit the external validity of the results. The scheduling is also confined to a working hour range from 08:00 to 18:00, with a maximum vehicle load capacity of 20 kg. These constraints reduce the scalability and practical applicability of the proposed method in more complex and realistic delivery settings.

III. THEORITICAL BASIS

A. Dynamic Programming

Dynamic programming is a problem-solving method that works by breaking down the solution into a series of stages. Each stage represents a step in the decision-making process, where the overall solution is viewed as a sequence of interconnected decisions. The term "program" in this context has no relation to computer programming. The word "dynamic" refers to the use of tables, often growing incrementally to store computed results and optimize the calculation process. Dynamic programming is particularly useful for solving optimization problems, whether they involve maximization or minimization objectives. In these problems, the solution typically involves evaluating multiple possible sequences of decisions.[3]

An optimal sequence of decisions in dynamic programming is derived based on the Principle of Optimality, which states that if a total solution is optimal, then any part of that solution leading up to a certain stage must also be optimal. This principle allows computations to move forward efficiently: when progressing from stage k to stage $k+1$, the algorithm can use the optimal results of stage k without having to recompute earlier stages. The cost at stage $k+1$ is defined as the cost obtained at stage k, plus the cost of transitioning from stage k to stage $k+1$ (denoted as $c_{k,k+1}$).[3]

Dynamic programming is widely applied in various fields to solve complex problems efficiently. Some prominent examples include determining the shortest path in a graph, solving the Integer Knapsack Problem to maximize value underweight constraints, addressing Capital Budgeting problems for optimal investment decisions under limited budgets, and tackling the Travelling Salesperson Problem (TSP), which aims to find the shortest possible route that visits a set of cities exactly once and returns to the origin. These problems share key characteristics such as overlapping subproblems and optimal substructure that make them well-suited to dynamic programming techniques.[4]

There are several characteristics of problems that are solvable with Dynamic Programming:

1. The problem can be divided into multiple stages, with only one decision made at each stage.
2. Each stage contains several states associated with that stage. Generally, a state represents the possible input configurations at a stage.
3. The result of the decision taken at a stage transforms the current state into a state at the next stage.
4. The cost at each stage increases steadily with the number of stages.
5. The cost at a given stage depends on the cumulative cost of the preceding stages and the cost of moving to the next stage.
6. There is a recursive relationship that determines the best decision for each state at stage k, which yields the best decision for each state at stage $k+1$.
7. The principle of optimality applies to the problem.[3]

Dynamic programming has two general approaches: forward (or top-down) and backward (or bottom-up). In the forward approach, calculations are performed starting from stage 1, progressing through stages 2, 3, and so on up to stage n. The decision variables in this sequence are represented as $x_1, x_2, \ldots, x_n$. In the backward approach, the algorithm starts from the final stage and moves in reverse order: from stage n to $n-1$, $n-2$, and eventually to stage 1. The sequence of decision variables in this case is $x_n, x_{n-1}, \ldots, x_1$.[3]

To solve a solvable problem, there are a few steps in developing Dynamic Programming Algorithm:

1. Characterize the structure of the optimal solution that identify the stages, decision variables, states, etc.
2. Define the value of the optimal solution recursively that express the optimal value of a stage in terms of the previous stages.
3. Compute the optimal value either forward or backward that use a table to store intermediate results.
4. Reconstruct the optimal solution (optional) to perform a backward trace to identify the decision path taken.[3]

B. Greedy Algorithm

The greedy algorithm is one of the most popular and straightforward methods for solving optimization problems. This algorithm is typically used to find optimal solutions in such problems, which can be classified into two types: maximization and minimization. The greedy algorithm solves problems in a step-by-step manner by selecting the best option available at each stage without considering future consequences (following the principle of "take what you can get now!"), with the *hope* that choosing a local optimum at every step will ultimately lead to a global optimum.[5]

There are several elements of Greedy Algorithm:

1. Candidate set (C) that contains the potential candidates to be selected at each step (e.g., nodes/edges in a graph, jobs, tasks, coins, items, characters, etc.).

2. Solution set (S) that consists of candidates that have already been selected to form the current solution.
3. Solution function that determines whether the current set of selected candidates forms a complete and valid solution.
4. Selection function that chooses the next candidate to add to the solution based on a specific greedy strategy. This strategy is heuristic in nature.
5. Feasibility function that checks whether a selected candidate can be feasibly added to the solution set (i.e., whether it satisfies the problem's constraints).
6. Objective function that defines the goal to be optimized either maximized or minimized.[5]

It's important to note that greedy algorithms generally produce locally optimal solutions, which do not always guarantee globally optimal results. In many cases, the solution may only be sub-optimal or even a pseudo-optimum. This happens for several reasons. First, greedy algorithms do not exhaustively explore all possible solutions like exhaustive search methods do. Second, there may be multiple selection functions available for a given problem, and choosing the correct one is crucial for achieving an optimal solution.[5]

As a result, greedy algorithms may not always yield the best possible outcome for every problem. However, they are often valuable for finding approximate solutions, especially when an exact solution would require exponential computation time. While a greedy approach may not find the absolute minimum-weight tour, it can still produce a reasonably good approximation of the optimal solution. In cases where the greedy algorithm does provide an optimal solution, its optimality must be proven mathematically. This can be challenging, and in many situations, it is easier to demonstrate that a greedy algorithm is not always optimal by providing a counterexample that a specific case where the solution it produces is clearly not the best.[5]

Greedy algorithms are widely used to solve various optimization problems. Common examples include the coin exchange problem, activity selection, and minimizing time in a system. They are also applied to the knapsack problem, job scheduling with deadlines, minimum spanning tree, shortest path, Huffman coding, and the Egyptian fraction problem. These problems benefit from greedy strategies that make locally optimal choices in hopes of finding an efficient overall solution.[5]

C. Branch and Bound Algorithm

The Branch and Bound (B&B) algorithm is used for solving optimization problems, either to minimize or maximize an objective function without violating any of the given constraints. Conceptually, Branch and Bound can be viewed as a combination of Breadth-First Search (BFS) and least-cost search. In pure BFS, the next node to be expanded is chosen based on the order in which it was generated (i.e., using a FIFO queue). However, in Branch and Bound, each node is assigned a cost value, denoted as ĉ(i), which represents an estimate of the minimum possible path cost to a goal node passing through node *i*. The next node selected for expansion is not based on generation order, but rather the node with the smallest estimated cost which is the essence of the least-cost search in the context of minimization problems.[6]

In solving optimization problems, each node in the state-space tree must have a mechanism to determine a bound on the best possible value of the objective function for any potential solution formed by extending the partial solution represented by that node. This includes keeping track of the best solution value found so far, to compare against new candidates.[6]

Branch and Bound also incorporate pruning techniques to discard paths that are unlikely to lead to an optimal solution. In general, pruning is applied when a node's cost is worse than the best-known solution, when the node violates problem constraints, or when the node represents a complete solution (a leaf node) but offers a worse outcome than the current best. In such cases, the node is eliminated from further consideration.[7]

In most real-world problems, the exact location of the solution node is unknown. Some problems where the solution location is better defined include the N-Queens problem, Knapsack problem, Graph Coloring, 8-puzzle game, and Travelling Salesperson Problem (TSP). For such cases, the cost or bound ĉ(i) of a node *i* serves as a heuristic estimate of the cheapest cost to reach a solution from node *i*. This estimate, ĉ(i), represents the lower bound of the cost required to complete a solution from that state. Branch and Bound has been effectively applied in a variety of problems, such as the N-Queens problem, 15-puzzle, Travelling Salesperson Problem (TSP), Assignment Problem, and Integer Knapsack Problem. [6,7,8,9]

There are a few steps to use Branch and Bound Algorithm:

1. Start from the root node. If the root node is a solution, stop. The solution has been found.
2. If there are no live nodes left, terminate the process (no solution found).
3. Select a live node with the smallest cost bound (ĉ) as the next node to expand. If there are multiple nodes with equal cost, choose one arbitrarily.
4. If the selected node is a solution node, compare its objective function value with the best solution found so far. If it is better, update the best solution. If only one solution is needed, stop.
5. If the selected node is not a solution, generate all of its children (expanding the node).
6. For each child node, compute its cost bound ĉ, and mark it as a live node.
7. Return to Step 2. [6]

Notes: The "live nodes" are typically maintained in a list or priority queue sorted by cost, but the algorithm itself doesn't prescribe a specific data structure, this depends on implementation. Pruning is applied to eliminate nodes whose cost bound is worse than the best solution found so far or that violate constraints. [6]

D. The Differences Between Dynamic Programming (DP), Greedy, and Branch and Bound (B&B) Algorithm

TABLE I. THE DIFFERENCES BETWEEN DYNAMIC PROGRAMMING (DP), GREEDY, AND BRANCH AND BOUND (B&B) ALGORITHM

Aspect	DP	Greedy	B&B
Problem Type	Optimization with overlapping subproblems and optimal substructure	Optimization with greedy-choice property and optimal substructure	Optimization (esp. combinatorial and constrained)
Approach	Bottom-up or top-down with memoization	Step-by-step, always takes locally optimal choice	Tree-based search with pruning and bounding
Solution Guarantee	Always finds optimal solution (if correctly applied)	Often approximate; not always optimal	Guaranteed to find optimal solution (with full exploration or correct bound)
Exploration Strategy	Exhaustive (but optimized via memoization)	Single path (no backtracking)	Selective exploration based on best bound (e.g., best-first search)
Subproblem Reuse	Yes	No	No
Constraint Handling	Limited, usually assumes relaxed constraints	Limited, must fit greedy criteria	Strong – handles constraints explicitly
Speed	Fast (polynomial) for suitable problems	Very fast (linear or greedy-specific)	Slower – may be exponential in worst case
Space Usage	Can be high (stores table of subproblems)	Low	Moderate to high (stores tree nodes + bounds)

E. Package

A package refers to a specific quantity of goods prepared for shipment. Rather than being transported individually, items are grouped together using loading platforms such as pallets, crates, wire mesh containers, or roller cages to form a single unit. Each package typically requires its own shipping document and is assigned a unique identifier for tracking and tracing throughout the delivery process. In real-world applications, the term package is often used interchangeably with parcel or packet. Using external packaging and handling aids, multiple individual packaging units can be consolidated into a complete package. In this context, the package also functions as the shipping unit.[10]

F. Courier

Based on Cambridge Dictionary Online, courier is a person or company that takes messages, letters, or parcels from one person or place to another.[11]

G. Scheduling

Based on Cambridge Dictionary Online, scheduling is the job or activity of planning the times at which particular tasks will be done or events will happen such as production/work/crew scheduling.[12]

IV. METHODOLOGY

A. Data Sample

The dataset used in this study is synthetically generated and formatted in JSON to simulate a controlled yet realistic packet delivery scenario. It comprises 25 distinct delivery entries, deliberately limited in size to accommodate the computational complexity of the algorithms applied namely, Dynamic Programming (DP) and Branch and Bound (B&B). Both techniques are known for their ability to solve optimization problems effectively, but they come with substantial computational overhead, especially as the problem size grows. Dynamic Programming requires building and maintaining large state tables, while Branch and Bound involves exploring a potentially large solution space with bounding and pruning operations. Therefore, a dataset size of 25 was chosen as a practical upper limit to ensure that algorithm performance could be analyzed within reasonable runtime and resource constraints, without sacrificing the complexity needed for meaningful evaluation.

Each data entry is uniquely identified by a packet_id and contains several key fields relevant to delivery operations. The weight_in_kg ranges from 0.5 kg to 15 kg, reflecting realistic load variations. The deadline_time spans from 08:00 to 18:00, simulating typical same-day delivery windows. Packet sizes are classified as *small*, *medium*, or *large*, which influence the estimated_time_in_minute required for delivery—10 minutes for small packets, 15 for medium, and 20 for large ones. Additionally, each packet is assigned a destination_address, which consists of randomly generated addresses within the Dago area of Bandung, emphasizing a localized delivery scenario ideal for route planning and schedule optimization. This structured and semantically rich dataset supports detailed experimentation on time-constrained delivery scheduling, vehicle load balancing, and the effectiveness of the selected algorithms in addressing logistics challenges under controlled conditions.

TABLE II. PACKET.SON'S PREVIEW

JSON's Preview

```
[
  {
    "packet_id": "PKT001",
    "weight_in_kg": 7.35,
    "deadline_time": "16:30",
    "size": "medium",
    "estimated_time_in_minute": 15,
    "destination_address": "Jl. Teuku Umar No.12, Dago"
  },
  // ... (23 additional data entries omitted for brevity) ...
  {
    "packet_id": "PKT025",
    "weight_in_kg": 2.30,
    "deadline_time": "17:30",
    "size": "small",
    "estimated_time_in_minute": 10,
    "destination_address": "Jl. Pagergunung No.30, Dago"
  }
]
```

This table displays the structure and key attributes of the packet.json the data used in this study. For full dataset details, please refer to folder data at GitHub Repository.

B. Problem Modeling

1. Dynamic Programming (DP)

   Step 1 (Characterize the Optimal Solution).
   The solution is modeled by representing each delivery combination as a bitmask. States represent subsets of delivered packets, and decision variables involve selecting which packet to add next. Constraints like weight limit and delivery deadlines define feasible transitions.
   Step 2 (Define the Recursive Relation).
   A DP table dp[mask] is initialized where each entry holds (completion_time, total_weight) for a given subset of packets. The base case (dp[0]) is set to the start of the delivery time. Other states are built by extending smaller subsets and updating values based on feasibility and improvement.
   Step 3 (Compute Using Table Iteration).
   The algorithm iterates through all subsets (mask) and their possible additions. It checks constraints (weight, deadline, total time) and updates dp[mask] if a better completion time or efficiency is achieved. This uses a bottom-up approach to reuse previously computed results.
   Step 4 (Reconstruct the Optimal Solution).
   After computing all values, the algorithm finds the best subset (best_mask) with the highest delivery value. It then traces back to identify the packets involved. The output includes delivery count, weight, time, and total revenue.

2. Greedy Algorithm

   Step 1 (Define Candidate Set).
   The candidate set (C) is the initial collection of all available packets (packets), which are potential elements to be selected and considered for the solution.
   Step 2 (Initialize Solution Set).
   The solution set (S), represented by selected_packets, begins as an empty list. It will be populated with the packets chosen by the algorithm throughout its process.
   Step 3 (Evaluate Solution Completeness).
   The solution is considered complete when all packets in the sorted candidate list (sorted_packets) have been considered for inclusion, and the algorithm has attempted to select the best ones according to its strategy.
   Step 4 (Implement Greedy Selection Strategy).
   The selection function is implemented by sorting the packets based on deadline_minutes (primary priority), followed by estimated_time_in_minute, and then weight_in_kg (in ascending order). The for packet in sorted_packets: loop then sequentially picks the "best" next candidate based on this established order.
   Step 5 (Check Candidate Feasibility).
   The feasibility function verifies whether the packet about to be added meets the capacity constraint (current_weight + packet['weight_in_kg'] <= MAX_CAPACITY_KG), the packet's individual deadline constraint (potential_absolute_completion_time <= packet['deadline_minutes']), and the end-of-day constraint (potential_absolute_completion_time <= END_OF_DAY_MINUTES).
   Step 6 (Define Optimization Objective).
   The primary objective (Objective Function) of this algorithm is to maximize the total_revenue_idr, which is calculated based on the total number of successfully delivered packages (total_packages_delivered).

3. Branch and Bound (B&B) Algorithm

   Step 1 (Initialize Root Node).
   The algorithm starts from the root node (current_index = 0), representing an initial state with no packets selected (current_weight = 0.0, current_cumulative_delivery_time_param = 0, current_value_base = 0, and an empty current_selected_packets_list). best_overall_value is initialized to a minimum value (-1) to track the best solution found so far.
   Step 2 (Define Branching Strategy).
   At each node in the search tree (explore_bnb_node), the algorithm generates two branches namely include branch and exclude branch. Include branch attempts to include the current packet (packet) if it satisfies all constraints (capacity, deadline, end-of-day). Exclude branch skips the current packet. Both branches are explored recursively (explore_bnb_node(current_index + 1, ...)) until all packets have been considered.
   Step 3 (Calculate Upper Bound).
   The calculate_upper_bound function computes an optimistic upper bound (upper_bound) on the value (revenue) that can potentially be achieved from the current node onwards to the end of the tree. This is done by greedily adding remaining feasible packets based on a simple heuristic (e.g., pay-per-weight).
   Step 4 (Implement Pruning (Bounding) Logic).
   At each node, the calculated upper_bound is compared against the best_overall_value found so far. If the upper_bound is less than or equal to best_overall_value, the branch is pruned (return), as it cannot possibly lead to a better solution. This significantly reduces the search space.
   Step 5 (Evaluate Solution Node).
   When the algorithm reaches a leaf node (current_index == num_packets_global), it signifies that a subset of packets has been fully considered.

The final revenue (final_revenue) for this subset is calculated. If this final_revenue is greater than the current best_overall_value, best_overall_value and best_overall_selected_packets are updated.

Step 6 (Recursive Exploration).

The processes of branching, bound calculation, and pruning are recursively executed until the entire relevant search space has been explored or pruned. Upon completion of the recursion, the final optimal results (including selected packets, total weight, time, and revenue) are returned.

C. Program Implementation to Find The Most Optimal Courier's Revenue

In this implementation, the project consists of five Python files located in the src folder and one JSON file stored in the data folder. The core algorithms are separated into three modules: bnb_algorithm.py for the Branch and Bound method, dp_algorithm.py for the Dynamic Programming approach, and greedy_algorithm.py for the Greedy strategy. Supporting functions such as data parsing and utility helpers are placed in utils.py, while main.py serves as the entry point to execute and compare the performance of the algorithms. The dataset, packet.json, is located in the data directory and contains structured information on the delivery packets, including attributes such as weight, size, deadline, and destination. This modular organization promotes clarity, maintainability, and facilitates experimentation across different algorithmic strategies.

File main.py serves as the primary entry point for the Packet Scheduling Algorithm System. It orchestrates the user interface, handles loading packet data, and calls the various optimization algorithms. Key functions in this file include format_minutes_to_hhmm for time conversion, print_results for displaying algorithm outputs, and run_algorithm which executes the chosen solver. The main program logic is encapsulated within the if __name__ == "__main__": block.

File utils.py contains utility functions used across the packet scheduling application. It includes parse_time_to_minutes for converting time strings, calculate_revenue for computing earnings, and load_packet_data for reading and preprocessing packet information from a JSON file.

File greedy_algorithm.py implements the Greedy algorithm for the packet scheduling problem. Its main function, solve_greedy, sorts packets based on a heuristic (earliest deadline, then shortest time, then lightest weight) and iteratively selects packets if they fit within current constraints, aiming for a locally optimal solution at each step.

File dp_algorithm.py contains the implementation of the Dynamic Programming algorithm. It includes print_dp_debug_info for structured debugging output and solve_dp, the core function that uses a bitmask approach to represent and optimize subsets of packets, finding the globally optimal solution.

File bnb_algorithm.py implements the Branch and Bound algorithm, a systematic search technique for optimization problems. It includes calculate_upper_bound for estimating potential values, explore_bnb_node which recursively branches through the solution space, and solve_bnb which orchestrates the search. It uses calculated upper bounds to prune branches that cannot lead to a better overall solution, significantly reducing the search space to find the optimal set of packets.

The implementation includes step-by-step output tracing to illustrate how each algorithm progresses through the problem space. However, due to the large number of iterations involved especially in the Dynamic Programming and Branch and Bound approaches that only a limited portion of the trace is displayed. Typically, this includes several steps from the beginning and a few from the end, providing a representative overview of the process without overwhelming the reader with excessive detail. This selective logging helps in understanding the algorithm's behavior while maintaining readability. For more details of the code implementation, it can be accessed at the GitHub Repository.

V. Testing and Analysis

A. Testing

The following section presents the results of code execution testing, which are illustrated through the images below. These visuals showcase the outputs produced by each algorithm when applied to the dataset, highlighting differences in performance, selected packets, total weight, time taken, and overall delivery value.

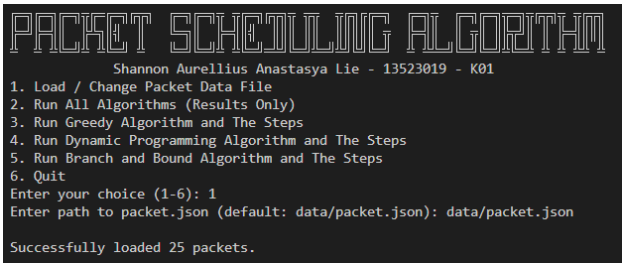


Fig. 1. Load data file (packet.json)

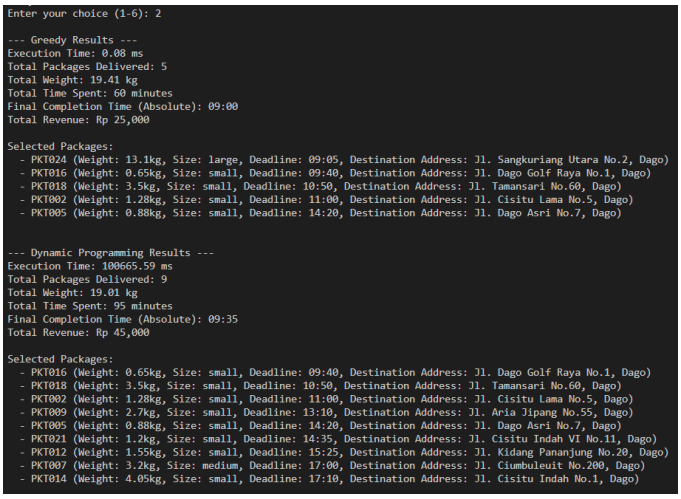


Fig. 2. Run All Algorithms Result Only (part 1)

```
--- Branch and Bound Results ---
Execution Time: 0.53 ms
Total Packages Delivered: 9
Total Weight: 19.96 kg
Total Time Spent: 95 minutes
Final Completion Time (Absolute): 09:35
Total Revenue: Rp 45,000

Selected Packages:
  - PKT016 (Weight: 0.65kg, Size: small, Deadline: 09:40, Destination Address: Jl. Dago Golf Raya No.1, Dago)
  - PKT018 (Weight: 3.5kg, Size: small, Deadline: 10:50, Destination Address: Jl. Tamansari No.60, Dago)
  - PKT002 (Weight: 1.28kg, Size: small, Deadline: 11:00, Destination Address: Jl. Cisitu Lama No.5, Dago)
  - PKT020 (Weight: 5.9kg, Size: medium, Deadline: 11:50, Destination Address: Jl. Sekeloa Utara No.30, Dago)
  - PKT009 (Weight: 2.7kg, Size: small, Deadline: 13:10, Destination Address: Jl. Aria Jipang No.55, Dago)
  - PKT005 (Weight: 0.88kg, Size: small, Deadline: 14:20, Destination Address: Jl. Dago Asri No.7, Dago)
  - PKT021 (Weight: 1.2kg, Size: small, Deadline: 14:35, Destination Address: Jl. Cisitu Indah VI No.11, Dago)
  - PKT012 (Weight: 1.55kg, Size: small, Deadline: 15:25, Destination Address: Jl. Kidang Pananjung No.20, Dago)
  - PKT025 (Weight: 2.3kg, Size: small, Deadline: 17:30, Destination Address: Jl. Pagergunung No.30, Dago)
```

Fig. 3. Run All Algorithms Result Only (part 2)

```
Enter your choice (1-6): 3

--- Greedy Algorithm Debugging Output ---
1. Packets sorted by (deadline, estimated_time, weight):
  PKT024: Deadline=09:05 (545 min), Est.Time=20 min, Weight=13.10 kg
  PKT016: Deadline=09:40 (580 min), Est.Time=10 min, Weight=0.65 kg
  PKT008: Deadline=09:50 (590 min), Est.Time=20 min, Weight=11.20 kg
  PKT011: Deadline=10:00 (600 min), Est.Time=20 min, Weight=7.00 kg
  PKT003: Deadline=10:15 (615 min), Est.Time=20 min, Weight=14.99 kg
  PKT018: Deadline=10:50 (650 min), Est.Time=10 min, Weight=3.50 kg
  PKT002: Deadline=11:00 (660 min), Est.Time=10 min, Weight=1.28 kg
  PKT013: Deadline=11:15 (675 min), Est.Time=15 min, Weight=10.10 kg
  PKT020: Deadline=11:50 (710 min), Est.Time=15 min, Weight=5.90 kg
  PKT023: Deadline=12:10 (730 min), Est.Time=15 min, Weight=4.95 kg
  PKT006: Deadline=12:30 (750 min), Est.Time=20 min, Weight=9.95 kg
  PKT009: Deadline=13:10 (790 min), Est.Time=10 min, Weight=2.70 kg
  PKT015: Deadline=13:35 (815 min), Est.Time=20 min, Weight=13.00 kg
  PKT005: Deadline=14:20 (860 min), Est.Time=10 min, Weight=0.88 kg
  PKT021: Deadline=14:35 (875 min), Est.Time=10 min, Weight=1.20 kg
  PKT012: Deadline=15:25 (925 min), Est.Time=10 min, Weight=1.55 kg
  PKT017: Deadline=16:05 (965 min), Est.Time=15 min, Weight=8.15 kg
  PKT001: Deadline=16:30 (990 min), Est.Time=15 min, Weight=7.35 kg
  PKT022: Deadline=16:40 (1000 min), Est.Time=20 min, Weight=10.80 kg
  PKT007: Deadline=17:00 (1020 min), Est.Time=15 min, Weight=3.20 kg
  PKT014: Deadline=17:10 (1030 min), Est.Time=10 min, Weight=4.05 kg
  PKT025: Deadline=17:30 (1050 min), Est.Time=10 min, Weight=2.30 kg
  PKT004: Deadline=17:45 (1065 min), Est.Time=15 min, Weight=5.10 kg
  PKT010: Deadline=18:00 (1080 min), Est.Time=15 min, Weight=12.50 kg
  PKT019: Deadline=18:20 (1100 min), Est.Time=20 min, Weight=14.80 kg
------------------------------------------------------------
```

Fig. 4. Run Greedy Algorithm and The Steps (part 1)

```
------------------------------------------------------------
2. Iterating through sorted packets and making greedy choices:
  Initial State: Current Time=480 min, Current Weight=0.00 kg

--- Iteration 1: Considering PKT024 ---
  Selected PKT024: Weight=13.10 kg, Est.Time=20 min
    New State: Current Time=500.00 min, Current Weight=13.10 kg

--- Iteration 2: Considering PKT016 ---
  Selected PKT016: Weight=0.65 kg, Est.Time=10 min
    New State: Current Time=510.00 min, Current Weight=13.75 kg

--- Iteration 3: Considering PKT008 ---
  Skipped PKT008: FAILED Weight Constraint.
    Current Weight=13.75 kg, Packet Weight=11.20 kg (Max Capacity=20)

--- Iteration 4: Considering PKT011 ---

... 55 additional iterations omitted ...

    Current Weight=19.41 kg, Packet Weight=2.30 kg (Max Capacity=20)

--- Iteration 23: Considering PKT004 ---
  Skipped PKT004: FAILED Weight Constraint.
    Current Weight=19.41 kg, Packet Weight=5.10 kg (Max Capacity=20)

--- Iteration 24: Considering PKT010 ---
  Skipped PKT010: FAILED Weight Constraint.
    Current Weight=19.41 kg, Packet Weight=12.50 kg (Max Capacity=20)

--- Iteration 25: Considering PKT019 ---
  Skipped PKT019: FAILED Weight Constraint.
    Current Weight=19.41 kg, Packet Weight=14.80 kg (Max Capacity=20)
------------------------------------------------------------
```

Fig. 5. Run Greedy Algorithm and The Steps (part 2)

```
3. Final Greedy Solution:

--- Greedy Results ---
Execution Time: 2.29 ms
Total Packages Delivered: 5
Total Weight: 19.41 kg
Total Time Spent: 60 minutes
Final Completion Time (Absolute): 09:00
Total Revenue: Rp 25,000

Selected Packages:
  - PKT024 (Weight: 13.1kg, Size: large, Deadline: 09:05, Destination Address: Jl. Sangkuriang Utara No.2, Dago)
  - PKT016 (Weight: 0.65kg, Size: small, Deadline: 09:40, Destination Address: Jl. Dago Golf Raya No.1, Dago)
  - PKT018 (Weight: 3.5kg, Size: small, Deadline: 10:50, Destination Address: Jl. Tamansari No.60, Dago)
  - PKT002 (Weight: 1.28kg, Size: small, Deadline: 11:00, Destination Address: Jl. Cisitu Lama No.5, Dago)
  - PKT005 (Weight: 0.88kg, Size: small, Deadline: 14:20, Destination Address: Jl. Dago Asri No.7, Dago)
```

Fig. 6. Run Greedy Algorithm and The Steps (part 3)

```
Enter your choice (1-6): 4

--- DP Algorithm Debugging Output ---
1. Initializing DP Table:
  dp[0] (empty set): Completion Time=480.00 min, Total Weight=0.00 kg
  Other states initially: (inf, inf)

2. Iterating through masks and updating DP table:
  Each mask represents a subset of packets. The algorithm builds optimal solutions
  for larger subsets from smaller ones, ensuring constraints are met.

3. Tracking Best Global Solution:
  The 'Best overall revenue' is updated whenever a mask results in a higher total revenue.
------------------------------------------------------------

--- Final DP Table States (Reachable Masks) ---
Mask (Binary)          Mask (Dec)   Completion Time (min)     Total Weight (kg)
--------------------------------------------------------------------------------
00000000000000000000000000 0            480.00                    0.00
00000000000000000000000001 1            500.00                    13.10
00000000000000000000000010 2            490.00                    0.65
00000000000000000000000011 3            510.00                    13.75
00000000000000000000000100 4            500.00                    11.20
00000000000000000000000110 6            510.00                    11.85
00000000000000000000001000 8            500.00                    7.00
00000000000000000000001010 10           510.00                    7.65
00000000000000000000001100 12           520.00                    18.20
00000000000000000000001110 14           530.00                    18.85

... 12075 additional reachable states omitted ...

10010000001000000000010 18890754     530.00                    18.95
10010000001000000001000000 18890816     530.00                    19.58
10010000001100000000000000 18898944     530.00                    19.18
10010000001100000000000010 18898946     540.00                    19.83
10010000001000000000000000 18907136     520.00                    18.65
10010000001000000000000010 18907138     530.00                    19.30
10010000001000000001000000 18907200     530.00                    19.93
10010000001010000000000000 18915328     530.00                    19.53
10010000001100000000000000 18923520     530.00                    19.85
10100000000000000000000000 20971520     515.00                    19.90
--------------------------------------------------------------------------------
```

Fig. 7. Run Dynamic Programming Algorithm and The Steps (part 1)

```
Final Optimal Solution Found:
Best Mask: 00000100011101010010010 (Decimal: 584290)
Maximum Revenue: 45000 IDR
------------------------------------------------------------

--- Dynamic Programming Results ---
Execution Time: 108711.05 ms
Total Packages Delivered: 9
Total Weight: 19.91 kg
Total Time Spent: 100 minutes
Final Completion Time (Absolute): 09:40
Total Revenue: Rp 45,000

Selected Packages:
  - PKT016 (Weight: 0.65kg, Size: small, Deadline: 09:40, Destination Address: Jl. Dago Golf Raya No.1, Dago)
  - PKT018 (Weight: 3.5kg, Size: small, Deadline: 10:50, Destination Address: Jl. Tamansari No.60, Dago)
  - PKT002 (Weight: 1.28kg, Size: small, Deadline: 11:00, Destination Address: Jl. Cisitu Lama No.5, Dago)
  - PKT023 (Weight: 4.95kg, Size: medium, Deadline: 12:10, Destination Address: Jl. Kyai Gede No.45, Dago)
  - PKT009 (Weight: 2.7kg, Size: small, Deadline: 13:10, Destination Address: Jl. Aria Jipang No.55, Dago)
  - PKT005 (Weight: 0.88kg, Size: small, Deadline: 14:20, Destination Address: Jl. Dago Asri No.7, Dago)
  - PKT021 (Weight: 1.2kg, Size: small, Deadline: 14:35, Destination Address: Jl. Cisitu Indah VI No.11, Dago)
  - PKT012 (Weight: 1.55kg, Size: small, Deadline: 15:25, Destination Address: Jl. Kidang Pananjung No.20, Dago)
  - PKT007 (Weight: 3.2kg, Size: medium, Deadline: 17:00, Destination Address: Jl. Ciumbuleuit No.200, Dago)
```

Fig. 8. Run Dynamic Programming Algorithm and The Steps (part 2)

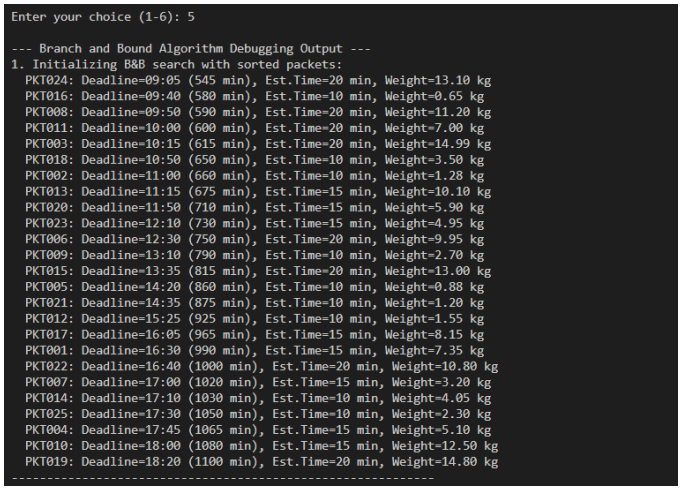


Fig. 9. Run Branch and Bound Algorithm and The Steps (part 1)

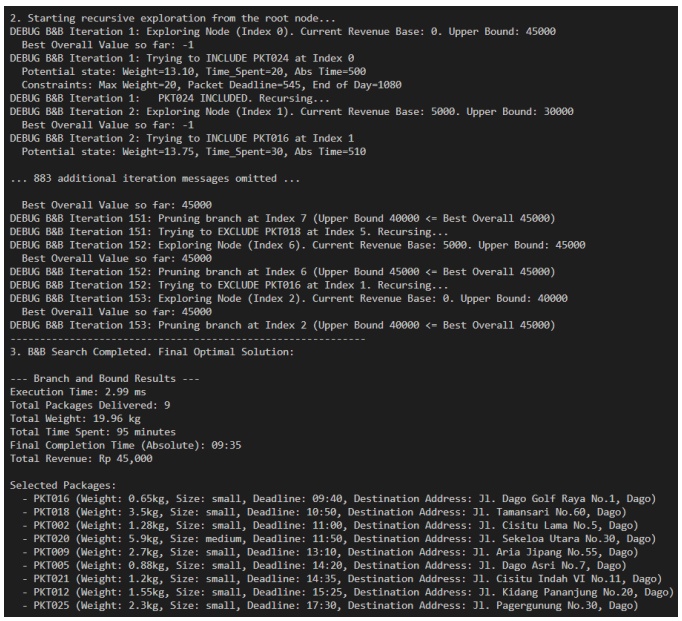


Fig. 10. Run Branch and Bound Algorithm and The Steps (part 2)

B. Testing Explanation

Testing was held on a packet scheduling algorithm application to evaluate the performance of three primary approaches: Greedy, Dynamic Programming (DP), and Branch and Bound (B&B). The packet data, comprising 25 individual packets, was successfully loaded from data/packet.json as input for all test runs.

The Greedy Algorithm successfully delivered 5 packages, resulting in a total weight of 19.41 kg. The total time spent was 60 minutes, yielding a revenue of Rp 25,000. The absolute final completion time was determined to be 09:00. In contrast, the Dynamic Programming Algorithm demonstrated markedly superior performance, with 9 packages delivered and a total weight of 19.01 kg. This process consumed 95 minutes and generated a substantial revenue of Rp 45,000, with an absolute completion time of 09:35. Similarly, the Branch and Bound Algorithm achieved comparable optimal results, delivering 9 packages with a total weight of 19.96 kg. It also required 95 minutes to complete, resulting in the same maximum revenue of Rp 45,000, and an absolute completion time of 09:35. Based on these findings, it is evident that both the Dynamic Programming and Branch and Bound algorithms consistently attained the optimal solution with maximum revenue, significantly outperforming the Greedy approach which yielded considerably lower revenue.

The Greedy Algorithm commenced by sorting packets primarily by their deadline, then by estimated time, and finally by weight. The iterative process involved sequentially considering each sorted packet. For instance, in Iteration 1, PKT024 was selected, increasing the current time to 500 minutes and the current weight to 13.10 kg. However, as the debugging output illustrates, several packets (e.g., PKT008, PKT004, PKT010, PKT019) were skipped due to violating the maximum weight constraint (Max Capacity=20 kg). This demonstrates the inherent "locally optimal" decision-making of the greedy approach, which may, at times, preclude reaching a globally optimal solution.

The Dynamic Programming Algorithm initiated its process with the careful initialization of its DP table. Specifically, dp[0], representing an empty set of packets, was assigned a completion time of 480 minutes and a weight of 0.00 kg. The algorithm then proceeded to iterate through binary masks, each corresponding to a unique subset of packets. During these iterations, the DP table was systematically updated, progressively constructing optimal solutions by building upon smaller subsets. A crucial aspect of this process was the continuous tracking and updating of the "Best overall revenue." This mechanism ensured that any mask yielding a higher total revenue would lead to an update in the global optimal, effectively navigating the solution space towards the maximum possible revenue. The final optimal solution identified by this method was marked by a Best Mask of 00001000111010100110010 (Decimal: 584290), confirming a Maximum Revenue of 45,000 IDR.

The Branch and Bound Algorithm began its execution by initializing and sorting the packets, similar to the Greedy algorithm, based on criteria like deadline, estimated time, and weight. The debugging output comprehensively illustrated the "Starting recursive exploration from the root node." This iterative process involved exploring various nodes, calculating the current revenue base, and dynamically determining upper bounds. A key efficiency feature, "Pruning branch" messages, clearly indicated instances where branches of the search tree were systematically eliminated. This pruning occurred when the upper bound of a given branch was found to be less than or equal to the "Best Overall Value so far," thereby significantly reducing the computational load by discarding non-promising paths. The successful completion of the search was confirmed by the message "B&B Search completed. Final Optimal Solution," which ultimately converged on the same optimal revenue of Rp 45,000, aligning with the results from the Dynamic Programming approach.

C. Algorithm and Method Analysis

The Greedy Algorithm demonstrated remarkably fast execution times, notably 0.08 ms and 2.29 ms in different test runs. This efficiency is inherent in its design. The primary computational bottleneck for the Greedy algorithm often lies in the initial sorting step. If N is the number of packets, sorting typically takes O(NlogN) time. The subsequent iterative selection process involves a single pass through the sorted packets, leading to an O(N) operation. Therefore, the overall time complexity of the Greedy algorithm is dominated by the sorting step, making it O(NlogN). This explains its rapid performance, though it comes at the cost of not guaranteeing a globally optimal solution.

The Dynamic Programming Algorithm exhibited significantly longer execution times, recorded at 10065.59 ms and 10871.05 ms. This considerable increase in time is expected due to its exhaustive exploration of subproblems to guarantee optimality. For problems like the knapsack problem (which packet scheduling often resembles), where the total number of items is N, the Dynamic Programming approach typically involves constructing a table that considers all possible subsets of items or all possible states up to a certain capacity. In the context of selecting subsets of packets, the algorithm often iterates through all 2^N possible subsets (represented by binary masks). For each subset, operations involving weight and time calculations are performed. Consequently, the time complexity of a typical Dynamic Programming solution for this type of problem is $O(2^N \cdot W)$ or $O(2^N \cdot N)$, where W is related to the maximum capacity or N is for iterating over items within subsets. Given the 25 packets, 2^{25} is a very large number, explaining the long execution times observed, despite achieving the optimal solution.

The Branch and Bound Algorithm presents an interesting case in terms of execution time. While it also aims for an optimal solution like Dynamic Programming, its performance can vary widely based on the effectiveness of its pruning strategy. In the tests conducted, the recorded execution times were notably fast, at 0.53 ms and 2.99 ms, comparable to, or even faster than, the Greedy algorithm in some instances. This indicates that the pruning mechanism was highly effective in drastically reducing the search space for the given dataset. In the worst-case scenario, Branch and Bound can still explore the entire solution space, leading to a complexity of $O(2^N)$, similar to exhaustive search or some DP approaches. However, in practice, with effective bounding and pruning heuristics, its average-case performance can be significantly better, often performing much closer to polynomial time complexity for many real-world instances. The observed quick execution times suggest that many non-optimal branches were efficiently discarded, allowing it to find the optimal solution without fully exploring the exponential search space. This makes Branch and Bound a powerful technique for achieving optimality with potentially better average-case performance than pure Dynamic Programming for specific problem instances.

VI. Conclusion

The implementation and testing of the Greedy, Dynamic Programming, and Branch and Bound algorithms for packet scheduling revealed a clear trade-off between computational efficiency and solution optimality. While the Greedy algorithm offered exceptionally fast execution times (e.g., 0.08 ms) due to its O(NlogN) complexity, its "locally optimal" decision-making led to sub-optimal revenue (Rp 25,000) and fewer packages delivered. Conversely, both the Dynamic Programming and Branch and Bound algorithms consistently achieved the optimal solution, yielding the maximum revenue of Rp 45,000 and delivering 9 packages. Although Dynamic Programming incurred significantly longer execution times (over 10 seconds) reflecting its $O(2^N)$ complexity, the Branch and Bound algorithm remarkably managed to achieve the same optimal results with very rapid execution (as low as 0.53 ms), demonstrating the powerful impact of its pruning heuristics in efficiently navigating the exponential search space. This suggests that for complex optimization problems where optimality is paramount, Branch and Bound can provide an effective balance between solution quality and practical performance.

VII. Appendix

Video Link at Youtube

Youtube link: https://youtu.be/Kzd3rwgAcHs

Code Repository at Github

Github Repository link:
https://github.com/shanlie20/packet_scheduling

Acknowledgment

The author would like to express sincere gratitude to the following individuals whose support and contributions have been vital throughout the completion of this paper:

1. To God Almighty, for the continual blessings and guidance that have provided strength and direction during every stage of this journey. The author's faith has been a constant foundation in overcoming challenges throughout the writing process.
2. To the author's parents, for their endless encouragement, moral support, and belief in the author's capabilities. Their unwavering presence and motivation have been a key factor in sustaining the author's dedication and perseverance.
3. To Dr. Ir. Rinaldi Munir, M.T., Dr. Ir. Rila Mandala, and Dr. Nur Ulfa Maulidevi, for their exceptional guidance as lecturers of the IF2211 Algorithm Strategies course. Their mentorship, clarity in teaching, and deep expertise have played a significant role in shaping the author's understanding of algorithmic principles and approaches.

The author is deeply appreciative of the meaningful influence and assistance provided by these individuals, without whom this paper would not have reached its final form.

REFERENCES

[1] CodeCrucks, "Dynamic Programming vs Branch and Bound," *CodeCrucks.com*, [Online]. Available: https://codecrucks.com/dynamic-programming-vs-branch-and-bound/.

[2] GeeksforGeeks, "Branch and Bound in DSA," *GeeksforGeeks.org*, [Online]. Available: https://www.geeksforgeeks.org/dsa/branch-and-bound-meaning-in-dsa/.

[3] R. Munir, *Program Dinamis (2025) Bagian 1*. Institut Teknologi Bandung, 2025. [Online]. Available: https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/25-Program-Dinamis-(2025)-Bagian1.pdf.

[4] R. Munir, *Program Dinamis (2025) Bagian 2*. Institut Teknologi Bandung, 2025. [Online]. Available: https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/26-Program-Dinamis-(2025)-Bagian2.pdf.

[5] R. Munir, *Algoritma Greedy (2025) Bagian 1*. Institut Teknologi Bandung, 2025. [Online]. Available: https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/04-Algoritma-Greedy-(2025)-Bag1.pdf.

[6] R. Munir, *Algoritma Branch and Bound (2025) Bagian 1*. Institut Teknologi Bandung, 2025. [Online]. Available: https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/17-Algoritma-Branch-and-Bound-(2025)-Bagian1.pdf

[7] R. Munir, *Algoritma Branch and Bound (2025) Bagian 2*. Institut Teknologi Bandung, 2025. [Online]. Available: https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/18-Algoritma-Branch-and-Bound-(2025)-Bagian2.pdf

[8] R. Munir, *Algoritma Branch and Bound (2025) Bagian 3*. Institut Teknologi Bandung, 2025. [Online]. Available: https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/19-Algoritma-Branch-and-Bound-(2025)-Bagian3.pdf

[9] R. Munir, *Algoritma Branch and Bound (2025) Bagian 4*. Institut Teknologi Bandung, 2025. [Online]. Available: https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/20-Algoritma-Branch-and-Bound-(2025)-Bagian4.pdf

[10] DHL Freight Connections, "Package – Logistics Dictionary," *DHL Freight Connections*, [Online]. Available: https://dhl-freight-connections.com/en/logistics-dictionary/package/.

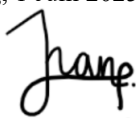
[11] Cambridge University Press, "Courier," *Cambridge Dictionary*, [Online]. Available: https://dictionary.cambridge.org/dictionary/english/courier.

[12] Cambridge University Press, "Scheduling," *Cambridge Dictionary*, [Online]. Available: https://dictionary.cambridge.org/dictionary/english/scheduling.

PERNYATAAN

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 1 Juni 2025



Shannon Aurellius Anastasya Lie
13523019