HANOI UNIVERSITY OF SCIENCE AND TECHNOLOGY
SCHOOL OF INFORMATION AND COMMUNICATION TECHNOLOGY

PLANNING OPTIMIZATION

Topic: Order Picking up route in Warehouse

List of Students:

No.	**Full Name**	**Student ID**
1	Vu Truong An	20220058
2	Le Danh Vinh	20220051
3	Truong Xuan Thong	20220044
4	Nguyen Dang Phuc	20220040

Supervisor: Dr. Bui Quoc Trung

Ha Noi - 2025

Contents

1 Problem statement **2**

2 Problem Formulation **3**

2.1 Sets and Parameters . . . 3

2.2 Decision Variables . . . 3

2.3 Objective Function . . . 3

2.4 Constraints . . . 3

3 Proposed Methods **5**

3.1 Backtracking . . . 5

3.2 Branch and Bound . . . 7

3.3 Integer Programming . . . 12

3.3.1 Advantages and Computational Characteristics . . . 12

3.4 Constraint Programming . . . 13

3.4.1 Improvements and Advantages . . . 13

3.5 Greedy . . . 14

3.5.1 Greedy by distance . . . 14

3.5.2 Greedy by score . . . 16

3.6 Beam Search . . . 18

3.7 Local Search . . . 21

3.7.1 Neighborhood Operators . . . 21

3.7.2 Hill Climbing . . . 22

3.7.3 Simulated Annealing . . . 22

3.7.4 Tabu Search . . . 23

3.8 Genetic Algorithm . . . 25

4 Results and Evaluation **32**

4.1 Test Data Generation Strategy . . . 32

4.1.1 Spatial Modeling . . . 32

4.1.2 Inventory and Feasibility Assurance . . . 32

4.2 Dataset Classification . . . 33

4.3 Experimental Results . . . 33

4.4 Analysis . . . 37

4.4.1 Visual Analysis of Solution Quality . . . 38

5 Conclusion **39**

1 Problem statement

There are M shelves in a large warehouse indexed $1, 2, \dots, M$. Shelf j is located at a specific point in the warehouse ($j = 1, \dots, M$). There are N types of products indexed $1, 2, \dots, N$. The amount of product i available in shelf j is denoted by Q_{ij}.

The warehouse staff starts from the door (point 0) of the warehouse, wants to visit a subset of shelves (each shelf is visited at most once), and comes back to the door to pickup products for customer orders. The total amount of product i that must be picked up is q_i ($i = 1, 2, \dots, N$).

The travel distance from point i to point j is given by d_{ij} ($0 \leq i, j \leq M$). The goal is to find the sequence of shelves to visit such that the total travel distance is minimal.

Input Format

- Line 1: Two positive integers N and M ($1 \leq N \leq 50, 1 \leq M \leq 1000$).
- Line $1 + i$ ($i = 1, \dots, N$): Contains the i-th row of matrix Q.
- Line $N + i + 2$ ($i = 0, 1, \dots, M$): Contains the i-th row of the distance matrix d.
- Line $N + M + 3$: Contains the required amounts $q_1, q_2, \dots, q_N$.

Output Format

- Line 1: Contains a positive integer n (the number of shelves visited).
- Line 2: Contains n positive integers $x_1, x_2, \dots, x_n$ representing the sequence of shelves to be visited.

2 Problem Formulation

2.1 Sets and Parameters

- M: Number of shelves.
- N: Number of products.
- d_{ij}: Traveling cost (distance) from point i to point j.
- Q_{ij}: The amount of product i in shelf j.
- q_i: The total amount of product i required.

2.2 Decision Variables

- $x_{ij} \in \{0, 1\}$: Binary variable, equal to 1 if we travel from point i to point j, 0 otherwise ($\forall i, j \in \{0, 1, \ldots, M\}$).
- $y_j \in \{0, 1\}$: Binary variable, equal to 1 if shelf j is visited, 0 otherwise ($\forall j \in \{1, \ldots, M\}$).
- $u_j \in \{1, \ldots, M\}$: Integer variable representing the position of shelf j in the tour ($\forall j \in \{1, \ldots, M\}$).

2.3 Objective Function

Minimize the total travel distance:

$$\min \sum_{i=0}^{M} \sum_{j=0}^{M} d_{ij} \cdot x_{ij} \tag{1}$$

2.4 Constraints

1. Demand Satisfaction: For each product i, we must collect at least the required amount q_i:

$$\sum_{j=1}^{M} Q_{ij} \cdot y_j \geq q_i, \quad \forall i \in \{1, \ldots, N\} \tag{2}$$

2. Flow Conservation: Each shelf j, if visited ($y_j = 1$), must have exactly one incoming edge and one outgoing edge:

$$\sum_{i=0}^{M} x_{ij} = y_j, \quad \forall j \in \{1, \ldots, M\} \tag{3}$$

$$\sum_{j=0}^{M} x_{ij} = y_i, \quad \forall i \in \{1, \ldots, M\} \tag{4}$$

3. Depot Constraints: The tour must start and end at the door (point 0):

$$\sum_{j=1}^{M} x_{0j} = 1 \tag{5}$$

$$\sum_{i=1}^{M} x_{i0} = 1 \tag{6}$$

4. No Self-loops:

$$x_{ii} = 0, \quad \forall i \in \{0, \ldots, M\} \tag{7}$$

5. Subtour Elimination (MTZ Constraints):

$$u_i - u_j + M \cdot x_{ij} \leq M - 1, \quad \forall i, j \in \{1, \ldots, M\}, i \neq j \tag{8}$$

3 Proposed Methods

The warehouse selection problem can be approached as a *constraint satisfaction and optimization problem*. Given the mathematical model, our solution strategy involves:

1. **Feasibility:** Ensure all product demands are satisfied by selected warehouses
2. **Optimality:** Minimize total travel distance among feasible solutions

This dual nature influences our algorithm design:

- **Exact methods** systematically explore solutions, pruning infeasible branches early
- **Heuristics** construct or improve feasible solutions while minimizing distance
- **IP formulation** explicitly encodes constraints and objective in a unified model

All proposed methods maintain feasibility as a hard constraint while seeking to optimize the travel distance objective.

3.1 Backtracking

Backtracking is a fundamental exhaustive search technique that systematically explores the solution space by constructing candidate solutions incrementally. The algorithm builds a solution step by step. When a partial solution satisfies all requirements, it is recorded; if a partial solution reaches the maximum depth without satisfying the constraints, the algorithm abandons that branch and explores other possibilities.

Key Idea:

The algorithm attempts to explore all relevant permutations of warehouse visits in a depth-first manner to find the optimal route. At each step, it selects an unvisited warehouse to add to the current route.

- If the **product demands are satisfied**, the current route is evaluated (distance calculated) to see if it is the best solution found so far, and the search on this branch terminates (as adding more warehouses would only increase cost).
- If the **demands are not yet satisfied**, the algorithm continues to add the next available warehouse.
- The algorithm only backtracks when it has either found a valid solution or exhausted all available warehouses in the current branch.

The complete procedure is summarized in Algorithm 1

Algorithm 1 Backtracking for Warehouse Selection Problem

```
function BACKTRACK($depth, current\_distance$)
    if all product demands are satisfied then ▷ Feasible solution found
        $total \leftarrow current\_distance + d$[last warehouse][depot]
        if $total < best\_distance$ then
            $best\_distance \leftarrow total$
            $best\_route \leftarrow$ current route
        end if
        return
    end if
    if $depth >$ number of warehouses then
        return ▷ All warehouses considered
    end if
    for each warehouse $w \in \{1, 2, \ldots, m\}$ do
        if $w$ is not yet visited then
            ▷ Try adding warehouse $w$ to route
            Mark $w$ as visited
            $previous \leftarrow$ last warehouse in current route (or depot if empty)
            $new\_distance \leftarrow current\_distance + d[previous][w]$
            Add $w$ to current route
            Update stock levels
            ▷ Recursive exploration
            BACKTRACK($depth + 1, new\_distance$)
            Backtrack: restore state
            Remove $w$ from current route
            Mark $w$ as unvisited
        end if
    end for
end function
```

Algorithm Analysis:

- **Completeness:** The algorithm is complete. It guarantees finding the global optimal solution (if one exists) because it exhaustively checks every valid permutation sequence until demands are met.

- **Time Complexity:** In the worst-case scenario (e.g., when demands are only met after visiting almost all warehouses), the algorithm explores a search space proportional to the number of permutations of m warehouses. The number of nodes in the search tree is bounded by:

$$\sum_{k=0}^{m} P(m,k) = \sum_{k=0}^{m} \frac{m!}{(m-k)!} \approx O(m! \cdot e)$$

  Multiplying by $O(n)$ operations per node (for feasibility checks/updates), the overall time complexity is $O(m! \cdot m \cdot n)$.

- **Space Complexity:** $O(m+n)$ is required to store the recursion stack, the visited array, and the current state of demands.

- **Practical Limitations:** Due to the factorial growth ($O(m!)$), this approach is strictly limited to very small instances. For example, $m = 15$ yields over 1.3×10^{12} permutations, which is computationally infeasible.

Advantages:

- Guaranteed to find the optimal solution.
- Serves as a baseline to measure the effectiveness of advanced algorithms (like Branch and Bound).

Disadvantages:

- Exponential time complexity makes it impractical for large instances.
- Does not utilize information about the "cost" to prune bad branches early (unlike Branch and Bound).

3.2 Branch and Bound

Branch and Bound is an intelligent exhaustive search technique that systematically explores the solution space like backtracking, but uses bounding functions to eliminate large portions of unpromising search branches. This pruning mechanism dramatically reduces the number of nodes explored while still guaranteeing optimal solutions.

Key Idea:

Branch and Bound is an intelligent exhaustive search technique that systematically explores the solution space similarly to backtracking, while leveraging bounding functions to prune large subsets of unpromising search branches. By maintaining valid upper and lower bounds, the algorithm significantly reduces the search space while still guaranteeing optimality.

In our solution, we adopt **two complementary lower-bounding strategies**, both of which are admissible and can be used independently or jointly to improve pruning effectiveness.

Core Components:

1. **Global Upper Bound (UB):** The cost of the best valid solution found so far (initialized via a Greedy heuristic).
2. **Local Lower Bound (LB):** A mathematical estimation of the *minimum possible cost* to complete a partial solution.
3. **Pruning Rule:** If $LB(\text{partial solution}) \geq UB$, prune this branch

Whenever the lower bound of a partial solution exceeds or equals the current upper bound, the entire subtree rooted at that partial solution can be safely pruned.

Upper Bound Initialization. To avoid excessive exploration in early stages, the upper bound is initialized using a greedy heuristic that incrementally selects the nearest unvisited warehouse until all product demands are satisfied:

$$UB_{\text{initial}} = \text{cost of greedy solution}.$$

This guarantees an initial feasible solution and enables effective pruning from the beginning of the search.

Lower Bounding Strategies.

Approach 1: Minimum Edge Cost Bound. The first bounding strategy uses a simple yet efficient estimation based on the minimum non-zero edge cost in the distance matrix. The complete procedure is summarized in Algorithm 2.

For a partial solution that has already visited k warehouses, with accumulated distance d_{current}, there remain exactly $m - k$ unvisited warehouses. In the worst case, all of them must be visited before returning to the depot. The lower bound is therefore computed as:

$$LB_{\min} = d_{\text{current}} + c_{\min} \times (m - k + 1),$$

where $c_{\min} = \min_{i,j:d_{ij}>0} d_{ij}$ is the minimum non-zero travel cost, and the additional $+1$ term corresponds to the mandatory return edge to the depot.

Rationale. Even under the most optimistic assumption that every remaining traversal incurs the minimum possible cost $c_{\min}$, any feasible completion must include at least $m-k$ transitions to visit the remaining warehouses and one final transition to return to the depot. Consequently, the term $c_{\min} \times (m - k + 1)$ constitutes a worst-case underestimate of the remaining travel cost, ensuring that the bound never overestimates the true optimal cost and is therefore admissible.

Approach 2: Minimum Spanning Tree (MST)–Based Bound. The second, tighter bounding strategy is based on a Minimum Spanning Tree (MST) constructed over the set of nodes that must be connected in any feasible completion. The complete procedure is summarized in Algorithm 3.

For a partial solution that has already visited k warehouses, let u denote the current location (the last visited warehouse, or the depot if $k = 0$). Let $\mathcal{U}$ be the set of unvisited warehouses. Since the route must eventually return to the depot (node 0), any feasible completion must connect all nodes in the following set:

$$\mathcal{V} = \{u\} \cup \mathcal{U} \cup \{0\}.$$

The lower bound is then defined as:

$$LB_{\text{MST}} = d_{\text{current}} + \text{MST}(\mathcal{V}),$$

where $\text{MST}(\mathcal{V})$ denotes the total weight of a minimum spanning tree over the node set $\mathcal{V}$.

Rationale. In the most optimistic completion scenario, the remaining route connects the current location, all $m-k$ unvisited warehouses, and the depot without any redundant traversal. Any valid completion path necessarily induces a connected graph over the node set $\mathcal{V}$. The minimum cost required to achieve such connectivity is exactly the MST cost over $\mathcal{V}$. Since any feasible route is a connected supergraph of this MST, its total travel cost cannot be lower than $\text{MST}(\mathcal{V})$. Therefore, LB_{MST} is a worst-case underestimate of the remaining cost and is admissible.

Algorithm 2 Branch and Bound with Minimum Edge Cost Bound

```
1: function BranchAndBound($depth, current\_distance$)
2:     $k \leftarrow depth - 1$ ▷ Number of warehouses already visited
3:     if all product demands are satisfied then
4:         $total \leftarrow current\_distance + d[\text{last warehouse}][0]$
5:         if $total < best\_solution$ then
6:             $best\_solution \leftarrow total$ ▷ Update upper bound
7:             $best\_route \leftarrow$ current route
8:         end if
9:         return
10:    end if
11:    ▷ Termination conditions
12:    if $k > m$ then
13:        return
14:    end if
15:    ▷ Compute minimum-edge lower bound
16:    $lower\_bound \leftarrow current\_distance + c_{\min} \times (m - k + 1)$
17:    ▷ Pruning test
18:    if $lower\_bound \geq best\_solution$ then
19:        return ▷ Prune this branch
20:    end if
21:    for each warehouse $w \in \{1, 2, \ldots, m\}$ do
22:        if $w$ is not yet visited then
23:            Mark $w$ as visited
24:            $previous \leftarrow$ last warehouse in route (or depot if $k = 0$)
25:            $new\_distance \leftarrow current\_distance + d[previous][w]$
26:            Add $w$ to current route
27:            Update current stocks
28:            if $new\_distance < best\_solution$ then
29:                BranchAndBound($depth + 1, new\_distance$)
30:            end if
31:            ▷ Backtrack
32:            Restore previous stocks
33:            Remove $w$ from route
34:            Mark $w$ as unvisited
35:        end if
36:    end for
37: end function
38:
39: Initialization:
40: Compute $c_{\min} \leftarrow \min_{i,j:d_{ij}>0} d_{ij}$
41: $(best\_route, best\_solution) \leftarrow$ Greedy
42: Reset all warehouses as unvisited
43: Reset all stock levels
44: BranchAndBound(1, 0)
45: Output: $best\_route$ and $best\_solution$
```

Algorithm 3 Branch and Bound with MST-Based Lower Bound

```
 1: function BranchAndBound($depth, current\_distance$)
 2:     $k \leftarrow depth - 1$ ▷ Number of warehouses already visited
 3:     if all product demands are satisfied then
 4:         $total \leftarrow current\_distance + d[\text{last warehouse}][0]$
 5:         if $total < best\_solution$ then
 6:             $best\_solution \leftarrow total$
 7:             $best\_route \leftarrow$ current route
 8:         end if
 9:         return
10:     end if
11:     if $k > m$ then
12:         return
13:     end if
14:     ▷ Construct set of unvisited warehouses
15:     $\mathcal{U} \leftarrow \{\, w \in \{1, \ldots, m\} \mid w \text{ is not visited} \,\}$
16:     $u \leftarrow$ last warehouse in route (or 0 if $k = 0$)
17:     ▷ Compute MST-based lower bound
18:     $LB_{\text{MST}} \leftarrow current\_distance + \text{MST}(\{u\} \cup \mathcal{U} \cup \{0\})$
19:     if $lower\_bound \geq best\_solution$ then
20:         return ▷ Prune this branch
21:     end if
22:     ▷ Explore candidates in increasing distance order
23:     for each warehouse $w \in \mathcal{U}$ sorted by $d[u][w]$ do
24:         Mark $w$ as visited
25:         $new\_distance \leftarrow current\_distance + d[u][w]$
26:         Add $w$ to current route
27:         if $w$ contributes to unsatisfied demands then
28:             Update current stocks
29:             if $new\_distance < best\_solution$ then
30:                 BranchAndBound($depth + 1, new\_distance$)
31:             end if
32:             Restore previous stocks
33:         end if
34:         Remove $w$ from route
35:         Mark $w$ as unvisited
36:     end for
37: end function
38:
39: Initialization:
40: Compute $c_{\min} \leftarrow \min_{i,j:d_{ij}>0} d_{ij}$
41: $(best\_route, best\_solution) \leftarrow$ Greedy
42: Reset visited flags and stock levels
43: BranchAndBound(1, 0)
44: Output: $best\_route$ and $best\_solution$
```

Algorithm Analysis:

- **Optimality:** Branch and Bound guarantees finding the optimal solution because:
  - All nodes are either explored or proven to not contain better solutions
  - The lower bound is admissible (never overestimates)
  - Pruning only eliminates provably suboptimal branches
- **Time Complexity:**
  - Worst case: $O(m! \cdot m \cdot n)$ (same as backtracking if no pruning occurs)
  - Average case: Significantly better due to pruning
  - Depends heavily on:
    * Quality of initial upper bound (greedy solution)
    * Tightness of lower bound
    * Problem structure and instance characteristics
- **Space Complexity:** $O(m + n)$ for state storage plus $O(m)$ recursion stack.
- **Practical Performance:** Branch and Bound typically explores only 1-10% of nodes compared to pure backtracking on realistic instances.

Advantages over Backtracking:

- **Intelligent pruning:** Eliminates unpromising branches using mathematical bounds
- **Faster convergence:** Good initial bound accelerates search
- **Scalability:** Can handle instances with $m \leq 15 - 20$ warehouses (vs. $m \leq 10$ for backtracking)
- **Still optimal:** Guarantees finding best solution unlike heuristics

Limitations:

- **Exponential worst case:** Still impractical for large instances ($m > 20$)
- **Bound quality:** Simple lower bound may not prune enough branches
- **Problem-dependent:** Performance varies significantly based on instance structure
- **Memory:** Deep recursion for large search trees

When to Use Branch and Bound: Branch and Bound is most effective when:

- Optimal solutions are required (vs. good-enough solutions)
- Problem size is moderate ($10 \leq m \leq 20$)
- A good initial solution can be obtained quickly
- Tight lower bounds are computable efficiently

For larger instances, more sophisticated methods like Branch and Cut or metaheuristics become necessary.

3.3 Integer Programming

Integer Programming (IP) formulates the warehouse retrieval problem as a rigorous mathematical optimization model. By encoding the selection and sequencing logic into linear inequalities, we can leverage state-of-the-art Mixed-Integer Programming (MIP) solvers. These solvers utilize a combination of symbolic preprocessing, cutting planes, and LP-based branch-and-bound trees to explore the solution space systematically.

Implementation Strategy:

Our model is implemented using the **OR-Tools Linear Solver** interface, specifically employing the **SCIP** (Solving Constraint Integer Programs) backend. SCIP is particularly effective here as it integrates constraint programming techniques within a mathematical programming framework.

Algorithm 4 Integer Programming Solution using SCIP

```
1: function SolveWithMIP($N, M, Q, d, q$)
2:     Initialize SCIP solver backend via OR-Tools
3:     Variables:
4:         $x_{ij} \in \{0,1\}$, $y_j \in \{0,1\}$, $u_j \in [1, M]$ ▷ Flow, Selection, and MTZ rank
5:     Constraints:
6:         $\sum_{j=1}^{M} Q_{ij} y_j \geq q_i, \quad \forall i \in \{1,\dots,N\}$ ▷ Demand satisfaction
7:         $\sum_{i=0}^{M} x_{ij} = y_j, \quad \sum_{j=0}^{M} x_{ij} = y_i, \quad \forall i,j \in \{1,\dots,M\}$ ▷ Flow conservation
8:         $u_i - u_j + M \cdot x_{ij} \leq M - 1, \quad \forall i,j \in \{1,\dots,M\}, i \neq j$ ▷ MTZ subtour elimination
9:     Objective:
10:        Minimize $\sum_{i=0}^{M} \sum_{j=0}^{M} d_{ij} x_{ij}$
11:    $status \leftarrow$ Solve(solver)
12:    if $status$ = OPTIMAL or $status$ = FEASIBLE then
13:        Extract edges where $x_{ij} \approx 1$
14:        Perform sequential traversal from depot (node 0) to reconstruct route
15:        return sequence and total distance
16:    end if
17: end function
```

3.3.1 Advantages and Computational Characteristics

The IP approach offers several distinct advantages for small to medium-scale warehouse instances:

- **Mathematical Optimality Certificates:** Unlike heuristic methods, IP solvers provide a "Dual Bound." This allows us to quantify the optimality Gap, providing a guarantee on how close the current feasible solution is to the absolute theoretical minimum.
- **Advanced Preprocessing (Presolve):** Modern solvers like SCIP automatically detect redundant constraints and tighten variable bounds before the search begins. In this problem, it can quickly eliminate shelves with zero stock for requested products.
- **Cutting Planes:** The solver generates "cuts" (valid inequalities) that prune the feasible region of the Linear Programming relaxation without removing any integer

solutions, significantly accelerating the search process compared to basic Backtracking.

- **Versatility:** The model is highly extensible. Adding constraints such as shelf weight limits, multi-trip requirements, or time windows only requires adding linear rows to the matrix without changing the underlying algorithm.

Usage Context: While IP is highly robust, the MTZ subtour elimination constraints grow at $O(M^2)$, which can lead to memory overhead for $M > 100$. For such large-scale instances, the Constraint Programming approach or Meta-heuristics are generally preferred for faster convergence.

3.4 Constraint Programming

While the Integer Programming formulation provides a rigorous mathematical framework, the use of Miller-Tucker-Zemlin (MTZ) constraints for subtour elimination often leads to a weak linear relaxation, resulting in slow convergence for large M. To address this, we implement a Constraint Programming (CP) model using the Google OR-Tools CP-SAT solver.

Algorithm 5 Constraint Programming Solution using CP-SAT

```
1: function SolveWithCPSAT($N, M, Q, d, q$)
2:     Initialize CP-SAT model and Solver from OR-Tools
3:     Define decision variables:
4:         $y_j \in \{0,1\}$ (Select shelf $j$), $x_{ij} \in \{0,1\}$ (Travel from $i$ to $j$)
5:     Add Logic Constraints:
6:         $\sum_{j=1}^{M}(Q_{ij} \cdot y_j) \geq q_i, \quad \forall i \in \{1,\dots,N\}$ ▷ Demand satisfaction
7:         $x_{jj} = 1 - y_j, \quad \forall j \in \{1,\dots,M\}$ ▷ Self-loop if shelf is not visited
8:         $x_{00} = 0$ ▷ Depot must be exited
9:     Degree Constraints:
10:        $\sum_{j=0}^{M} x_{ij} = 1, \quad \sum_{i=0}^{M} x_{ij} = 1, \quad \forall i,j \in \{0,\dots,M\}$
11:    Add Global Constraint:
12:        AddCircuit(all arcs $x_{ij}$) ▷ Enforce a single Hamiltonian circuit
13:    Set Objective: Minimize $\sum_{i=0}^{M}\sum_{j=0,j\neq i}^{M}(d_{ij} \cdot x_{ij})$
14:    $status \leftarrow$ Solve(model)
15:    if $status$ = OPTIMAL or $status$ = FEASIBLE then
16:        Reconstruct route by following $x_{ij} = 1$ edges starting from depot
17:        return extracted route and objective value
18:    end if
19: end function
```

3.4.1 Improvements and Advantages

The CP approach introduces several key advantages over the traditional Integer Programming model:

- **Efficient Subtour Elimination:** The **Circuit Constraint** is a global constraint that ensures the set of selected edges forms a single Hamiltonian circuit. Unlike MTZ constraints which add $O(M^2)$ linear inequalities, the CP-SAT solver uses advanced graph-based algorithms to prune the search space, preventing subtours much more efficiently.

- **Self-Loop Mechanism:** To handle the optional nature of shelves (since not all M shelves need to be visited), we introduce **self-loops** ($x_{jj} = 1$ if $y_j = 0$). This allows the `AddCircuit` constraint to remain valid over a fixed number of nodes while effectively "skipping" unselected shelves with zero contribution to the objective distance.
- **Domain Pruning and Boolean Propagation:** When a shelf is selected ($y_j = 1$), the solver immediately propagates this to the flow constraints, narrowing the possible values for x_{ij} faster than standard branch-and-bound techniques.
- **Implicit Connectivity:** In MIP, the connectivity between the depot and the chosen shelves is maintained through auxiliary variables u_j. In CP, the `AddCircuit` constraint ensures connectivity natively, reducing the total number of variables and improving numerical stability.

3.5 Greedy

3.5.1 Greedy by distance

This method is a variation of the "Nearest Neighbor" strategy, enhanced with a filtering mechanism. At each step, starting from the current position, the warehouse staff searches for the next shelf that satisfies two conditions:

1. The shelf must be **useful** (contains at least one product type required by the current order).
2. The shelf has the **minimum distance** from the current position among all useful shelves.

This process repeats until the required quantity for all products is collected.

Pseudocode

Algorithm 6 Greedy by Distance (Smart Nearest Neighbor)

```
Require: $N$ products, $M$ shelves, Distance matrix $d$, Stock matrix $Q$, Demand vector $q$
 1: $current\_pos \leftarrow 0$ ▷ Start at the warehouse door
 2: $collected \leftarrow \{0, \ldots, 0\}$ ▷ Vector of collected items so far
 3: $visited \leftarrow \emptyset$
 4: $path \leftarrow \emptyset$
    ▷ Continue until all required items are collected
 5: while $\exists i \in \{1 \ldots N\}$ such that $collected[i] < q[i]$ do
 6:     $next\_shelf \leftarrow -1$
 7:     $min\_dist \leftarrow \infty$
 8:     for $shelf \leftarrow 1$ to $M$ do
 9:         if $shelf \in visited$ then
10:             continue
11:         end if
    ▷ Check if the shelf contains any item we still need
12:         $is\_useful \leftarrow$ false
13:         for $i \leftarrow 1$ to $N$ do
14:             if $collected[i] < q[i]$ and $Q[i][shelf] > 0$ then
15:                 $is\_useful \leftarrow$ true
16:                 break
17:             end if
18:         end for
19:         if $is\_useful$ and $d(current\_pos, shelf) < min\_dist$ then
20:             $min\_dist \leftarrow d(current\_pos, shelf)$
21:             $next\_shelf \leftarrow shelf$
22:         end if
23:     end for
24:     if $next\_shelf = -1$ then
25:         break ▷ No more reachable useful shelves
26:     end if
27:     Move to $next\_shelf$
28:     $visited \leftarrow visited \cup \{next\_shelf\}$
29:     $path \leftarrow path \cup \{next\_shelf\}$
30:     $current\_pos \leftarrow next\_shelf$
    ▷ Update collected items based on the shelf's inventory
31:     for $i \leftarrow 1$ to $N$ do
32:         $collected[i] \leftarrow collected[i] + Q[i][next\_shelf]$
33:     end for
34: end while
35: return $path$
```

Complexity

- **Time Complexity:** $O(M^2N)$
  - The main loop runs at most M times (visiting shelves).
  - In each iteration, the algorithm scans up to M unvisited shelves.
  - For each shelf, the `is_useful` check iterates through N products.

- **Space Complexity:** $O(M^2 + MN)$
  - Dominated by the distance matrix (M^2) and the stock matrix (MN).

3.5.2 Greedy by score

This approach introduces a heuristic function that balances the benefit (amount of goods collected) and the cost (travel distance).

At each step, instead of simply choosing the nearest shelf, the algorithm evaluates all unvisited shelves and assigns a "score" to each based on the following ratio:

$$Score(v) = \frac{\text{Total Useful Items at } v}{d(\text{current\_pos}, v)}$$

The algorithm then selects the shelf with the highest score. This ensures that the agent prefers shelves that offer a high volume of necessary items relative to the distance required to reach them.

Pseudocode

Algorithm 7 Greedy by Score

```
Require: $N$ products, $M$ shelves, Stock matrix $Q$, Distance matrix $d$, Demand vector $q$
 1: $current\_pos \leftarrow 0$ ▷ Start at warehouse door
 2: $collected \leftarrow \{0, \dots, 0\}$ ▷ Initially, no items collected
 3: $visited \leftarrow \emptyset$
 4: $path \leftarrow \emptyset$
    ▷ Loop until all required quantities are fulfilled
 5: while $\exists i \in \{1 \dots N\}$ such that $collected[i] < q[i]$ do
 6:     $best\_shelf \leftarrow -1$
 7:     $max\_score \leftarrow -\infty$
 8:     for $shelf \leftarrow 1$ to $M$ do
 9:         if $shelf \in visited$ then continue
10:         end if
    ▷ 1. Calculate the useful value of the shelf (items we need)
11:         $value \leftarrow 0$
12:         for $j \leftarrow 1$ to $N$ do
13:             $remaining\_need \leftarrow \max(0, q[j] - collected[j])$
14:             $value \leftarrow value + \min(Q[j][shelf], remaining\_need)$
15:         end for
    ▷ 2. Calculate score based on Value/Distance ratio
16:         $dist \leftarrow d(current\_pos, shelf)$
17:         if $dist > 0$ then
18:             $score \leftarrow value/dist$
19:         else
20:             $score \leftarrow \infty$
21:         end if
22:         if $score > max\_score$ then
23:             $max\_score \leftarrow score$
24:             $best\_shelf \leftarrow shelf$
25:         end if
26:     end for
27:     if $best\_shelf = -1$ then break
28:     end if
    ▷ 3. Move to the best shelf and update inventory
29:     Move to $best\_shelf$
30:     $visited \leftarrow visited \cup \{best\_shelf\}$
31:     $path \leftarrow path \cup \{best\_shelf\}$
32:     $current\_pos \leftarrow best\_shelf$
33:     for $j \leftarrow 1$ to $N$ do
34:         $collected[j] \leftarrow collected[j] + Q[j][best\_shelf]$
35:     end for
36: end while
37: Return to door (accumulate final distance)
38: return $path$
```

Complexity

- **Time Complexity:** $O(M^2N)$

  - The main loop runs at most M times.
  - In each iteration, it scans up to M unvisited shelves.
  - Calculating the *score* (value/distance) requires iterating through N products to sum the useful quantities.

- **Space Complexity:** $O(M^2 + MN)$
  - Dominated by the distance matrix (M^2) and the stock matrix (MN).

Remarks:
Advantages:

- Simple to implement, very fast runtime $O(m^2 \times n)$.
- Provides a good initial solution for other algorithms such as Branch and Bound, Local Search, and Beam Search.

Disadvantages:

- Does not guarantee an optimal solution.
- Local decisions may lead to suboptimal results.

Additional Comments:

- The greedy by score strategy performs better when warehouses are unevenly distributed, where a few warehouses store a large amount of goods while others store very little.
- The greedy by distance strategy better reflects practical scenarios in which warehouses are relatively evenly distributed in terms of inventory, without extreme imbalance.

3.6 Beam Search

Beam Search is a heuristic tree search algorithm that explores multiple partial solutions simultaneously while restricting the search width to a fixed number of promising candidates. Instead of committing to a single locally optimal decision, Beam Search maintains several competing partial solutions at each depth level.

Compared to greedy search, which constructs only one route, Beam Search significantly reduces the risk of early suboptimal decisions while keeping the computational cost manageable.

Key Idea: Starting from the depot, the algorithm incrementally builds routes by expanding each partial solution with several nearest unvisited warehouses. At each depth, only the best B partial solutions (beam width) with the smallest accumulated travel cost are retained. A greedy solution is first computed to provide an initial upper bound, which enables effective pruning of inferior states.

Beam Expansion Criterion: At a given state with current location c, candidate warehouses are selected as:

$$\mathcal{C}(c) = \text{TopB}_{w \in W_{\text{unvisited}}} d[c][w]$$

Each warehouse $w \in \mathcal{C}(c)$ generates a new state by extending the route, updating the accumulated cost, and collecting additional products.

State Representation: Each state stores the following information:

- Current route (path)
- Collected quantity for each product
- Visited warehouse indicators
- Last visited location
- Accumulated travel cost

Algorithm Workflow: The Beam Search procedure ís summarized in Algorithm 8, consists of the following phases:

1. **Initialization:** Initialize the beam with a single empty state at the depot
2. **Expansion:** Expand each state in the beam by visiting up to B nearest unvisited warehouses
3. **Pruning:** Discard states whose cost exceeds the current best solution
4. **Selection:** Retain only the best B states for the next depth
5. **Termination:** When all demands are satisfied, return to the depot and update the best solution

Algorithm 8 Beam Search

```
function BeamSearch($B, D_{\max}$)
    Initialize beam with one empty state at depot
    Compute greedy solution to obtain initial upper bound $best\_ans$
    for $depth = 1$ to $D_{\max}$ do
        $next\_beam \leftarrow \emptyset$
        for each state $s$ in beam do
            if all demands are satisfied in $s$ then
                Update $best\_ans \leftarrow \min(best\_ans, s.cost + d[s.last][0])$
                continue
            end if
            Select up to $B$ nearest unvisited warehouses from $s.last$
            for each selected warehouse $w$ do
                Create new state $s'$
                Update route, cost, stock, and visited flags
                if $s'.cost < best\_ans$ then
                    Add $s'$ to $next\_beam$
                end if
            end for
        end for
        if $next\_beam$ is empty then
            break
        end if
        Keep the best $B$ states in $next\_beam$ (minimum cost)
        $beam \leftarrow next\_beam$
    end for
end function
```

Theoretical Properties and Practical Analysis:

- **Optimality:** Beam Search does *not* guarantee finding the optimal solution.
- **Time Complexity:**
  - Worst case: $O(D \cdot B \cdot m \log m)$
  - In practice: Much lower due to aggressive pruning using the greedy upper bound
  - Runtime is primarily influenced by:
    - Beam width B
    - Maximum search depth D
    - Effectiveness of pruning via the current best solution
- **Space Complexity:** Beam Search requires storing up to B states per depth, resulting in:

$$O(B \cdot (m + n))$$

memory usage, which is significantly lower than full breadth-first or best-first search.

- **Sensitivity to Beam Width:** The beam width B controls the trade-off between exploration and efficiency:
  - Small B: Fast execution but higher risk of suboptimal solutions
  - Large B: Better solution quality at increased computational cost
  - Empirically, small values ($B = 3$ to 5) already yield substantial improvements over greedy search

3.7 Local Search

Local search techniques iteratively refine a feasible solution by exploring its neighborhood—small, local modifications that may reduce the objective. Starting from a greedy tour that satisfies shelf-demand constraints, we repeatedly probe neighborhoods until no improving move exists.

3.7.1 Neighborhood Operators

We define four neighborhood operators that generate neighboring solutions by making small modifications to the current tour. Let the current solution be represented as a sequence $\pi = (s_1, s_2, \ldots, s_k)$ of k selected shelves, with the tour being $0 \to s_1 \to s_2 \to \cdots \to s_k \to 0$.

1. **2-opt**: Reverses a segment of the tour between positions i and j (where $1 \leq i < j \leq k$).
   - Transform $(s_1, \ldots, s_{i-1}, s_i, \ldots, s_j, s_{j+1}, \ldots, s_k)$
     to $(s_1, \ldots, s_{i-1}, s_j, \ldots, s_i, s_{j+1}, \ldots, s_k)$.
   - Removes edges (s_{i-1}, s_i), (s_j, s_{j+1}); adds (s_{i-1}, s_j), (s_i, s_{j+1}).
   - Neighborhood size: $O(k^2)$
2. **Swap**: Exchanges two shelves at positions i and j (where $1 \leq i < j \leq k$).
   - Transform $(s_1, \ldots, s_i, \ldots, s_j, \ldots, s_k)$ to $(s_1, \ldots, s_j, \ldots, s_i, \ldots, s_k)$.
   - For adjacent positions ($j = i + 1$): affects 3 edges; otherwise: 4 edges.
   - Neighborhood size: $O(k^2)$
3. **Remove**: Removes a shelf at position i if feasibility remains satisfied.
   - Transform $(s_1, \ldots, s_{i-1}, s_i, s_{i+1}, \ldots, s_k)$ to $(s_1, \ldots, s_{i-1}, s_{i+1}, \ldots, s_k)$.
   - Check: $\forall p \in \{1, \ldots, n\}$: $\sum_{j \neq i} Q_p(s_j) \geq q_p$.
   - Neighborhood size: $O(k)$
4. **Add**: Inserts an unvisited shelf s' at position i in the tour.
   - Transform $(s_1, \ldots, s_{i-1}, s_i, \ldots, s_k)$ to $(s_1, \ldots, s_{i-1}, s', s_i, \ldots, s_k)$.
   - Removes edge (s_{i-1}, s_i); adds (s_{i-1}, s'), (s', s_i).
   - Neighborhood size: $O(k \times (m - k))$

For each operator, the objective change Δf is computed incrementally in $O(1)$ for 2-opt, swap, remove; and $O(n)$ for add (due to stock updates).

3.7.2 Hill Climbing

Ideas: Hill climbing starts from the greedy solution and applies neighborhood operators in a fixed order (2-opt, swap, remove, add) until no improvement is found. It accepts only better solutions, stopping at the first local optimum.

Implementation:

Algorithm 9 Hill Climbing Local Search

```
Initialize current solution from greedy()
improved ← true
while improved do
    improved ← false
    if try_2opt() improves solution then
        improved ← true
        continue
    end if
    if try_swap_shelves() improves solution then
        improved ← true
        continue
    end if
    if try_remove_shelf() improves solution then
        improved ← true
        continue
    end if
    if try_add_shelf() improves solution then
        improved ← true
        continue
    end if
end while
```

Each `try_*` function exhaustively checks all possible moves of that type and applies the first improving one found.

Advantages/Disadvantages: Simple to implement and computationally efficient for small to medium-sized problems. However, it typically converges to the nearest local optimum without any mechanism to escape, potentially missing better solutions elsewhere in the search space.

Complexity: $O(\text{iterations} \times |\text{neighborhood}| \times O(\text{move}))$, where $|\text{neighborhood}|$ is $O(k^2)$ for 2-opt and swap operations (k is the tour length), and $O(m)$ for remove/add operations (m is the number of shelves). Typically efficient for $m \leq 1000$.

3.7.3 Simulated Annealing

Ideas: Simulated annealing allows temporarily worse moves with probability $\exp(-\Delta/T)$, where Δ is the cost increase and T is the temperature parameter. Temperature decreases over time (e.g., geometrically), enabling exploration and escape from local optima early in the search while converging to better solutions as temperature cools.

Implementation:

Algorithm 10 Simulated Annealing

```
1: Initialize current solution from greedy()
2: $T_{\text{init}} \leftarrow 100.0$, $T_{\min} \leftarrow 0.01$
3: best ← current
4: while time < limit do
5:     progress ← elapsed/limit
6:     $T \leftarrow T_{\text{init}} \times (T_{\min}/T_{\text{init}})^{\text{progress}}$
7:     Randomly select move type (weighted: 60% 2-opt, 15% swap, 15% remove, 10%
   add)
8:     Compute new_cost for random move
9:     if accept_worse(current, new_cost, $T$) then
10:         Apply move
11:     end if
12:     if new_cost < best then
13:         Update best
14:     end if
15:     if acceptance rate too low then
16:         Reset to best solution, restart with higher temperature
17:     end if
18: end while
19: return best
```

The acceptance function is: `accept_worse`$(c, n, T) = \begin{cases} \text{true} & \text{if } n \leq c \\ \text{true with prob. } e^{-(n-c)/T} & \text{otherwise} \end{cases}$

Uses random sampling for moves instead of exhaustive enumeration.

Advantages/Disadvantages: Can discover better global optima than hill climbing by accepting worse moves, allowing escape from local optima. However, requires careful parameter tuning (initial temperature, cooling schedule, time limit) and runs slower due to probabilistic acceptance and periodic restarts.

Complexity: $O(\text{iterations} \times O(\text{move}))$, with iterations typically ranging from 10^5 to 10^6. Move cost is similar to hill climbing but with higher constant factors from randomization and state management. Time-bounded rather than iteration-bounded.

3.7.4 Tabu Search

Ideas: Tabu search maintains a tabu list (short-term memory) of recently applied moves to prevent cycling back to previously visited solutions. It allows *aspiration criteria* to override the tabu status if a move leads to a solution better than the current global best. When stagnation occurs (no improvement after threshold iterations), a *diversification* strategy perturbs the solution to explore new regions.

Implementation:

Algorithm 11 Tabu Search

```
1: Initialize current solution from greedy()
2: tabu_set ← $\emptyset$, best ← current
3: no_improve_cnt ← 0
4: while time < limit do
5:     Select move type randomly (weighted distribution)
6:     best_move ← null
7:     for each sampled move $m$ in neighborhood do
8:         Compute new_cost($m$)
9:         if $m \notin$ tabu_set or new_cost($m$) < global_best then
10:            if new_cost($m$) < new_cost(best_move) then
11:                best_move ← $m$
12:            end if
13:        end if
14:    end for
15:    if best_move $\neq$ null then
16:        Apply best_move
17:        Add best_move to tabu_set and queue
18:        if queue size > max_tabu_size then
19:            Remove oldest move from tabu_set
20:        end if
21:        if new_cost < best then
22:            Update best, no_improve_cnt ← 0
23:        else
24:            no_improve_cnt ← no_improve_cnt + 1
25:        end if
26:    end if
27:    if no_improve_cnt $\geq$ threshold then
28:        Diversify: apply random swaps, clear tabu_set
29:        no_improve_cnt ← 0
30:    end if
31:    if periodic reset condition then
32:        Reset current to best solution
33:    end if
34: end while
35: return best
```

Tabu list size is dynamic (e.g., $\max(10, m/2)$), using a FIFO queue for eviction. Samples approximately 50,000 moves per iteration rather than exhaustive search.

Advantages/Disadvantages: Effectively avoids local optima and cycling through explicit memory of recent moves. Diversification mechanism helps explore different regions when stuck. However, requires careful management of tabu lists, aspiration criteria, and diversification strategies, making implementation more complex. Memory overhead for maintaining tabu structures.

Complexity: $O(\text{iterations} \times \text{samples} \times O(\text{move}))$, with approximately 50,000 samples per move type per iteration. Tabu checks are $O(1)$ using hash-based sets. Similar asymptotic complexity to simulated annealing but with additional constant factors for memory management and sampling.

3.8 Genetic Algorithm

This algorithm is based on the Biased Random-Key Genetic Algorithms (BRKGA) framework proposed by Gonçalves and Resende.

In this algorithm, each individual in the population is represented by a chromosome, which is a vector of random real numbers (random keys) in the interval $[0, 1]$. The size of this vector corresponds to the number of shelves M in the warehouse.

The mapping process from a chromosome to a valid route (Decoder) operates on the following principles:

- **Priority Definition:** The value of each gene (key) determines the priority of the corresponding shelf. Shelves with smaller key values are prioritized for consideration.
- **Route Construction:** The algorithm sorts the shelves based on their priority. It iteratively considers each shelf; if a shelf contains required products that satisfy the remaining demand, it is added to the route. The insertion position is determined using the *Cheapest Insertion Heuristic* (inserting at the position that minimizes the increase in total distance).
- **Pruning:** Once a feasible route satisfying all product demands is constructed, a pruning step is performed to remove redundant shelves (provided that their removal does not violate product constraints) to reduce the total cost.
- **Local Search (Intensification):** To enhance solution quality, local search techniques such as *Relocate*, *Swap*, and *2-Opt* are applied. Note that due to high computational costs, this intensification step is selectively applied only in the final generation and is restricted to the top percentage of elite individuals (e.g., top 5%).

The evolutionary mechanism of BRKGA maintains diversity by partitioning the population into an *Elite* set (high-fitness individuals) and a *Non-elite* set. In the Crossover process, the first parent is invariably selected from the Elite set, while the second parent is chosen randomly from the **entire population**, with the offspring inheriting genes from the Elite parent with a higher probability (e.g., bias $\rho_e = 0.7$) to ensure that favorable characteristics are propagated. This strategy effectively permits both Elite-NonElite and Elite-Elite matings. Selecting the second parent from the entire population (rather than restricting selection solely to the non-elite set) accelerates convergence. Importantly, by maintaining a relatively low elite fraction (approximately 20%), the algorithm ensures sufficient genetic diversity, as the vast majority of crossover operations ($\approx 80\%$) still effectively draw the second parent from the non-elite pool. This hybrid strategy allows for rapid exploitation while avoiding the premature convergence and local optima traps associated with exclusive Elite-Elite mating.

Pseudo-code

Algorithm 12 Decoder: Mapping Chromosome to Valid Route

```
Require: Chromosome $v$ (vector of random keys), Demand vector $q$, Distance matrix $d$,
    Shelves inventory $Q$, $do\_local\_search\_flag$
Ensure: A feasible route and its total distance
 1: Define set of all shelves $S = \{1, \ldots, M\}$
 2: Sort $S$ based on values in $v$ in ascending order (priority queue)
 3: $Route \leftarrow \emptyset$, $Collected \leftarrow \vec{0}$
 4: for each shelf $s \in S$ in sorted order do
 5:     if $Collected$ satisfies demand $q$ then
 6:         break
 7:     end if
 8:     Find insertion position $pos$ in $Route$ that minimizes distance increase (Cheapest
    Insertion)
 9:     Insert $s$ into $Route$ at $pos$
10:     Update $Collected$ with products from shelf $s$
11: end for
12: $Route \leftarrow$ PRUNE($Route$) ▷ Remove redundant shelves cost-aware
13: if $do\_local\_search\_flag$ is True then
14:     $Route \leftarrow$ RELOCATELOCALSEARCH($Route$)
15:     $Route \leftarrow$ SWAPLOCALSEARCH($Route$)
16:     if distance matrix $d$ is symmetric then
17:         $Route \leftarrow$ TWOOPT($Route$)
18:     end if
19:     $Route \leftarrow$ PRUNE($Route$)
20: end if
21: return CALCULATEDISTANCE($Route$), $Route$
```

Algorithm 13 BRKGA Optimization Algorithm

```
Require: Parameters: Population size $p$, Elite fraction $\rho_e$, Mutant fraction $\rho_m$, Crossover
    bias $\rho_b$, Max generations $G_{max}$
 1: Initialize population $\mathcal{P}$ of size $p$ where each individual consists of random keys drawn
    uniformly from $[0, 1]$
 2: Evaluate $\mathcal{P}$ using DECODER($\mathcal{P}[i]$, $do\_local\_search\_flag$ = False)
 3: $BestSol \leftarrow$ Individual with min cost in $\mathcal{P}$
 4: for $g \leftarrow 1$ to $G_{max}$ do
 5:     Sort $\mathcal{P}$ by fitness (cost) in ascending order
 6:     Update $BestSol$ if $\mathcal{P}[0]$ is better
 7:     $\mathcal{P}_{next} \leftarrow \emptyset$
 8:     Elite Step: Copy top $p_e = \lfloor p \times \rho_e \rfloor$ individuals from $\mathcal{P}$ to $\mathcal{P}_{next}$
 9:     Crossover Step:
10:     for $k \leftarrow 1$ to $(p - p_e - p_m)$ do
11:         Select parent $A$ from Elite set $\{\mathcal{P}[0], \ldots, \mathcal{P}[p_e - 1]\}$
12:         Select parent $B$ randomly from entire population $\mathcal{P}$
13:         Add $Child$ to $\mathcal{P}_{next}$
14:     end for
15:     Mutation Step: Add $p_m$ individuals with random keys to $\mathcal{P}_{next}$
16:     $\mathcal{P} \leftarrow \mathcal{P}_{next}$
17:     Evaluate $\mathcal{P}$ using DECODER($\mathcal{P}[i]$, $do\_local\_search\_flag$ = False)
18:     if $g = G_{max}$ then
19:         $\triangleright$ Intensification on Elites in the final generation
20:         for $i \leftarrow 0$ to Top $k$ Elites do
21:             Re-evaluate $\mathcal{P}[i]$ with DECODER($\mathcal{P}[i]$, $do\_local\_search\_flag$ = True)
22:         end for
23:         Update $BestSol$
24:         break
25:     end if
26: end for
27: return $BestSol$
```

Algorithm 14 Relocate Local Search

```
function RelocateLocalSearch($Route, d, max\_moves$)
    $moves \leftarrow 0$
    $improved \leftarrow$ true
    while $improved$ and $moves < max\_moves$ do
        $improved \leftarrow$ false
        $n \leftarrow |Route|$
        for $i \leftarrow 0$ to $n - 1$ do
            $b \leftarrow Route[i]$
            $a \leftarrow (i = 0?0 : Route[i - 1])$
            $c \leftarrow (i = n - 1?0 : Route[i + 1])$
            $\Delta_{remove} \leftarrow d[a][c] - d[a][b] - d[b][c]$
            $R' \leftarrow Route \setminus \{b\}$ ▷ Temporary route without node $b$
            $best\_j \leftarrow -1$
            $best\_j\_change \leftarrow 0$
            for $j \leftarrow 0$ to $|R'|$ do
                $p \leftarrow (j = 0?0 : R'[j - 1])$
                $q \leftarrow (j = |R'|?0 : R'[j])$
                $\Delta_{insert} \leftarrow d[p][b] + d[b][q] - d[p][q]$
                if $\Delta_{remove} + \Delta_{insert} < best\_j\_change$ then
                    $best\_j\_change \leftarrow \Delta_{remove} + \Delta_{insert}$
                    $best\_j \leftarrow j$
                end if
            end for
            if $best\_j \neq -1$ then
                Remove($Route, i$)
                Insert($Route, best\_j, b$)
                $moves \leftarrow moves + 1$
                $improved \leftarrow$ true
                break ▷ Restart scanning from current state
            end if
        end for
    end while
    return $Route$
end function
```

Algorithm 15 Swap Local Search

```
function SwapLocalSearch($Route, d, max\_swaps$)
    $n \leftarrow |Route|$
    if $n < 2$ then return $Route$
    end if
    $swaps \leftarrow 0$
    while $swaps < max\_swaps$ do
        $best\_change \leftarrow 0$
        $best\_i \leftarrow -1, best\_j \leftarrow -1$
        for $i \leftarrow 0$ to $n - 2$ do
            for $j \leftarrow i + 1$ to $n - 1$ do
                ▷ Calculate cost difference $\Delta$ for swapping $Route[i]$ and $Route[j]$
                $\Delta \leftarrow$ CalculateSwapDelta($Route, i, j, d$)
                if $\Delta < best\_change$ then
                    $best\_change \leftarrow \Delta$
                    $best\_i \leftarrow i, best\_j \leftarrow j$
                end if
            end for
        end for
        if $best\_i < 0$ then break
        end if
        Swap($Route[best\_i], Route[best\_j]$)
        $swaps \leftarrow swaps + 1$
    end while
    return $Route$
end function
```

Algorithm 16 2-opt Local Search

```
function TwoOpt($Route, d, max\_swaps$)
    $n \leftarrow |Route|$
    if $n < 4$ then return $Route$
    end if
    $improved \leftarrow$ true, $swaps\_done \leftarrow 0$
    while $improved$ and $swaps\_done < max\_swaps$ do
        $improved \leftarrow$ false
        for $i \leftarrow 0$ to $n - 2$ do
            $a \leftarrow (i = 0 ? 0 : Route[i - 1])$
            $b \leftarrow Route[i]$
            for $k \leftarrow i + 1$ to $n - 1$ do
                $c \leftarrow Route[k]$
                $d\_next \leftarrow (k = n - 1 ? 0 : Route[k + 1])$
                $before \leftarrow d[a][b] + d[c][d\_next]$
                $after \leftarrow d[a][c] + d[b][d\_next]$
                if $after < before$ then
                    Reverse($Route, i, k$)
                    $improved \leftarrow$ true, $swaps\_done \leftarrow swaps\_done + 1$
                    if $swaps\_done \geq max\_swaps$ then
                        return $Route$
                    end if
                end if
            end for
        end for
    end while
    return $Route$
end function
```

Complexity of BRKGA

- **Time Complexity** $\mathcal{O}(G \cdot P \cdot M^2 \cdot N)$ where: N: Number of product types, M: Number of shelves (nodes), P: Population size, G: Number of generations.

    1. `decode` **Function (Individual Decoding)** – $\mathcal{O}(M^2 \cdot N)$

       This is the dominant component as it repeats for every individual in every generation.

       – *Route Construction:* Inserts up to M nodes into the route. Each insertion scans for the optimal position in the current route ($\mathcal{O}(M)$) and updates product states ($\mathcal{O}(N)$).
         $\Rightarrow$ Total: $\mathcal{O}(M(M + N))$.

       – *Pruning:* In the worst case, iterates through M nodes to attempt removal, checking the `required` constraint on N products each time.
         $\Rightarrow$ Total: $\mathcal{O}(M^2 \cdot N)$.

    2. **Evolution Loop** – $\mathcal{O}(G \cdot P)$

       – The algorithm performs decoding (`decode`) for P individuals in each generation, repeated G times.

    – Total cost: $\mathcal{O}(G \cdot P \times \text{decode cost})$.

3. **Local Search (Intensification)** – $\mathcal{O}(L \cdot M^2)$
    – Functions `two_opt`, `swap`, and `relocate` have a complexity of $\mathcal{O}(L \cdot M^2)$, where L is the iteration limit.
    – However, these are applied only to a small number of individuals (Elites) or in the final generation, thus they do not alter the overall asymptotic order of the process.

Remarks:

- **Advantages:**
    – The evolutionary mechanism of Genetic Algorithms (GA), combined with the introduction of random mutants, enables the algorithm to escape local optima more effectively than single-point search methods.
    – The integration of Local Search in the final stages or within the decoder allows for the refinement of coarse solutions into high-quality results.
    – The decoding process for each individual in the population is mathematically independent, making the algorithm highly suitable for multi-threaded implementation to accelerate computation.
- **Disadvantages:**
    – As an approximate method, the algorithm does not guarantee finding the global optimum (exact solution) like mathematical programming methods.
    – The complex decoding process, repeated thousands of times across generations, results in significantly longer execution times compared to simple greedy algorithms, making it less suitable for real-time applications.
    – The performance of the algorithm is sensitive to input parameter settings (e.g., population size, elite fraction), often requiring a time-consuming tuning process to achieve optimal results.

4 Results and Evaluation

4.1 Test Data Generation Strategy

To objectively evaluate the performance of the proposed algorithms, we implemented a robust test generation strategy. Instead of generating purely random distance matrices, which often violate the triangle inequality and fail to reflect reality, we constructed a dataset based on a simulated 2D warehouse environment.

4.1.1 Spatial Modeling

The warehouse is modeled as a 2D Cartesian plane with dimensions 1000×1000. The spatial configuration is defined as follows:

- The **Depot (Door)** is fixed at coordinates $(0, 0)$.
- Each **Shelf** j $(j = 1, \ldots, M)$ is assigned a random coordinate (x_j, y_j) uniformly distributed within the plane.
- The travel distance $d(i, j)$ between any two points is calculated using the Euclidean distance, rounded to the nearest integer:

$$d(i,j) = \text{round}\left(\sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}\right) \tag{9}$$

This approach ensures that the distance matrix satisfies the *triangle inequality* $(d(i,j) \leq d(i,k) + d(k,j))$, which is critical for the correct performance of spatial heuristics and local search algorithms.

4.1.2 Inventory and Feasibility Assurance

A common issue in random test generation is the creation of infeasible instances where demand exceeds supply. To prevent this and to mimic real-world inventory sparsity, the following logic was applied:

- **Sparse Inventory Matrix (Q):** Real-world shelves do not stock every product type. We introduced a sparsity factor where approximately 40% of entries in the inventory matrix $Q[i][j]$ are zero. This increases the complexity for Greedy algorithms, as the nearest neighbor may not contain the required item.
- **Feasible Demand (q):** The demand for each product i is derived from the total available supply in the warehouse, denoted as $S_i = \sum_{j=1}^{M} Q[i][j]$. The demand q_i is generated as a random integer in the range:

  $$q_i \in [1, \alpha \times S_i] \tag{10}$$

  where α (e.g., 0.6) represents the supply-demand tightness ratio. This constraint guarantees that a valid solution always exists.

4.2 Dataset Classification

To systematically evaluate the performance and scalability of the algorithms, the experiments were conducted on three distinct dataset categories based on problem size (N) and constraints (M):

1. **Small Scale** ($N \leq 8, M \leq 18$): Designed to verify the correctness of exact algorithms (Backtracking, Branch-and-Bound). In these scenarios, the global optimum is achievable within reasonable time limits, serving as a baseline to validate the convergence of heuristic methods.
2. **Medium Scale** ($N \approx 10, M \approx 47 - 100$): Represents the transition phase where exact methods (Backtracking, BnB) begin to fail (Time Limit Exceeded - TLE). These tests evaluate the efficiency of Constraint Programming (CP) and the initial stability of metaheuristics.
3. **Large Scale** ($N \geq 15, M \geq 400$): Designed to stress-test metaheuristics. In this range, exact solvers and CP cannot find a solution within the time limit. The primary metric shifts to the ability of algorithms (GA, Tabu Search, SA) to escape local optima and minimize the objective function.

4.3 Experimental Results

The following tables present the performance comparison across the three scales defined above. The time limit was set to a fixed threshold ($\approx 300s$), denoted as TLE (Time Limit Exceeded) if surpassed.

Table 1: Performance on Small Scale ($N = 5, M = 9$). All algorithms found valid solutions.

Algorithm	**Total distance**	**Runtime (s)**
Backtracking	**2865**	0.65
Greedy	3002	0.44
Branch And Bound	**2865**	0.53
Beam Search	3002	0.42
Integer Programming (SCIP)	**2865**	0.23
Constraint Programming	**2865**	0.62
Hill Climbing	**2865**	0.42
Simulated Annealing	**2865**	2.00
Tabu Search	**2865**	2.00
GA	**2865**	0.67

Table 2: Performance on Medium Scale ($N = 7, M = 12$). Backtracking runtime increases significantly (11.85s), indicating the start of exponential growth, while metaheuristics remain stable.

Algorithm	**Total distance**	**Runtime (s)**
Backtracking	**2263**	11.85
Greedy	2916	0.59
Branch And Bound	**2263**	0.51
Beam Search	2916	0.60
Integer Programming (SCIP)	**2263**	0.80
Constraint Programming	**2263**	0.64
Hill Climbing	2315	0.53
Simulated Annealing	**2263**	2.00
Tabu Search	**2263**	2.00
GA	**2263**	0.67

Table 3: Performance on Medium Scale ($N = 7, M = 15$). Backtracking fails to converge (TLE). Branch and Bound and GA continue to find the global optimum efficiently ($< 1s$).

Algorithm	**Total distance**	**Runtime (s)**
Backtracking	-	TLE
Greedy	2900	0.53
Branch And Bound	**2188**	0.51
Beam Search	2261	0.50
Integer Programming (SCIP)	**2188**	0.88
Constraint Programming	**2188**	0.65
Hill Climbing	2346	0.54
Simulated Annealing	**2188**	2.53
Tabu Search	**2188**	2.57
GA	**2188**	0.85

Table 4: Performance on Medium-Large Transition ($N = 8, M = 18$). Exact methods show drastic performance degradation (SCIP: 26.8s, BnB: 10.33s). CP and GA demonstrate superior scalability.

Algorithm	**Total distance**	**Runtime (s)**
Backtracking	-	TLE
Greedy	3615	0.65
Branch And Bound	**2611**	10.33
Beam Search	3095	0.82
Integer Programming (SCIP)	**2611**	26.80
Constraint Programming	**2611**	1.05
Hill Climbing	2792	0.52
Simulated Annealing	**2611**	2.43
Tabu Search	**2611**	2.54
GA	**2611**	0.90

Table 5: Performance on Medium Scale ($N = 7, M = 47$). Exact methods encounter TLE; CP remains viable.

Algorithm	**Total distance**	**Runtime (s)**
Backtracking	-	TLE
Greedy	5512	0.71
Branch And Bound	-	TLE
Beam Search	5340	0.57
Integer Programming (SCIP)	-	TLE
Constraint Programming	**3521**	17.09
Hill Climbing	4553	0.54
Simulated Annealing	3840	10.56
Tabu Search	**3521**	10.69
GA	**3521**	2.73

Table 6: Performance on Medium-Large Scale ($N = 10, M = 100$). Constraint Programming (CP) fails to converge (TLE). GA achieves the global optimum significantly faster (10.45s) than other metaheuristics.

Algorithm	**Total distance**	**Runtime (s)**
Backtracking	-	TLE
Greedy	7222	0.64
Branch And Bound	-	TLE
Beam Search	6271	0.50
Integer Programming (SCIP)	-	TLE
Constraint Programming	-	TLE
Hill Climbing	5350	0.54
Simulated Annealing	4097	40.51
Tabu Search	3741	40.48
GA	**3725**	10.45

Table 7: Performance on Large Scale ($N = 15, M = 424$). Simple heuristics yield poor quality solutions (¿10,000). GA demonstrates the best trade-off, finding the shortest path (8244) with the lowest runtime among metaheuristics.

Algorithm	**Total distance**	**Runtime (s)**
Backtracking	-	TLE
Greedy	12719	1.05
Branch And Bound	-	TLE
Beam Search	12015	0.42
Integer Programming (SCIP)	-	TLE
Constraint Programming	-	TLE
Hill Climbing	10206	0.96
Simulated Annealing	8820	80.61
Tabu Search	8640	80.67
GA	**8244**	60.84

Table 8: Performance on Large Scale ($N = 20, M = 1000$). Exact methods fail to converge (TLE). Genetic Algorithm (GA) outperforms all other metaheuristics, achieving the lowest total distance (15291).

Algorithm	**Total distance**	**Runtime (s)**
Backtracking	-	TLE
Greedy	21375	0.67
Branch And Bound	-	TLE
Beam Search	21375	0.77
Integer Programming (SCIP)	-	TLE
Constraint Programming	-	TLE
Hill Climbing	17663	31.64
Simulated Annealing	16940	121.25
Tabu Search	16596	122.80
GA	**15291**	124.94

4.4 Analysis

Based on the experimental results and solution visualizations, several critical observations can be made regarding the behavior of the algorithms:

- **Scalability Limits of Exact Methods:** Exact algorithms (Backtracking, Branch-and-Bound) are strictly limited to small instances ($N \leq 8$). The exponential growth in runtime (e.g., from 0.65s at $N = 5$ to TLE at $N = 7, M = 15$) confirms their inability to handle the combinatorial explosion inherent in the Warehouse Routing problem.

- **The "Greedy Trap" vs. Global Optimization:** Simple heuristics like Greedy and Hill Climbing exhibit extremely low latency ($< 1s$ even for $N = 20$) but suffer from poor solution quality. As observed in the visual analysis (Figure 1), these algorithms frequently get trapped in local optima, resulting in paths that are significantly longer (up to 40% worse) than those found by metaheuristics.

- **Constraint Programming (CP) Viability:** CP serves as a powerful bridge for medium-scale problems ($N = 7, M = 47$), outperforming standard exact methods. However, it lacks the scalability required for large instances ($M \geq 100$), where the constraint propagation overhead becomes prohibitive.

- **Superiority of Genetic Algorithm (GA):** Among metaheuristics, GA demonstrates the best robustness. While Simulated Annealing and Tabu Search require similar runtime budgets (120s for largest test), GA achieved the lowest total distance (**15291**). This suggests that the population-based approach of GA is more effective at maintaining diversity and exploring the search space than the single-solution trajectory methods (Tabu/SA) in highly complex environments.

4.4.1 Visual Analysis of Solution Quality

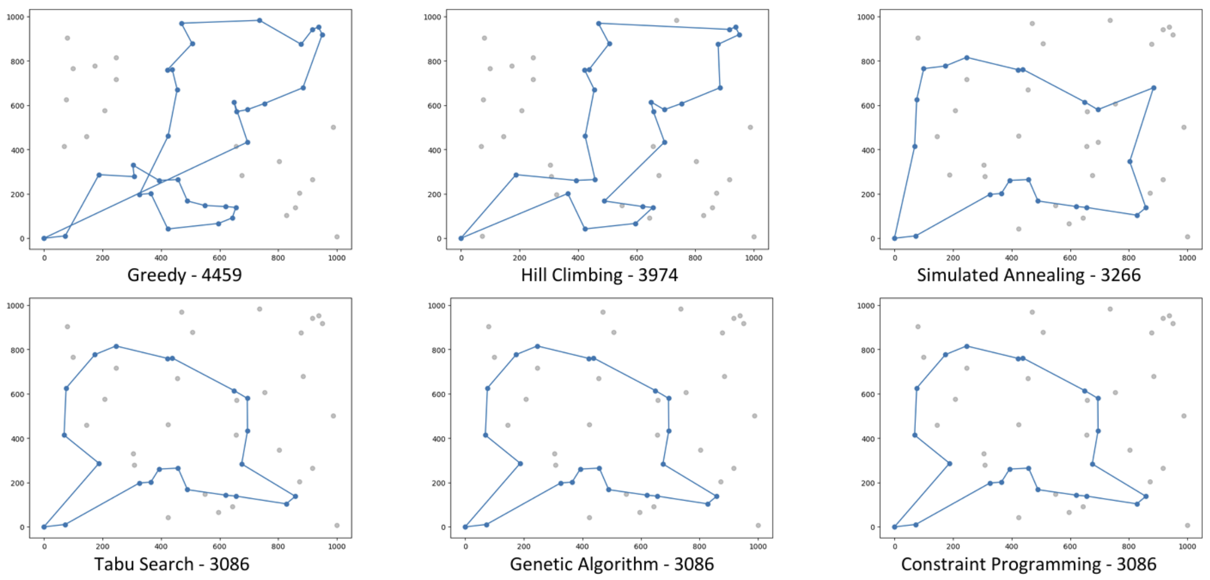

Figure 1: Visualization of solution paths for a Medium Scale instance ($N = 10, M = 50$). Comparison between heuristics (Greedy, Hill Climbing) and metaheuristics/CP.

Figure 1 illustrates the routing paths generated by different algorithms for a representative Medium Scale test case ($M = 50$). Visual inspection reveals significant differences in solution structure:

- **Local Optima Traps:** Simple heuristics like **Greedy** (Cost: 4459) and **Hill Climbing** (Cost: 3974) exhibit chaotic routing with frequent unnecessary crossings. This visually confirms their tendency to get trapped in poor local optima early in the search process.
- **Transition: Simulated Annealing** (Cost: 3266) shows a marked improvement over Hill Climbing, demonstrating its ability to escape some local optima, though it has not yet reached the global minimum.
- **Convergence of Advanced Methods: Tabu Search**, **Genetic Algorithm (GA)**, and **Constraint Programming (CP)** all converged to the identical best-known solution (Cost: 3086). The visualizations for these algorithms show a highly optimized structure with minimal path overlap, verifying that for medium-sized datasets, metaheuristics can achieve solution quality comparable to exact constraint-based methods.

5 Conclusion

This research presented a comprehensive analysis and experimental comparison of various algorithms for the Warehouse Order Picking Routing problem. By evaluating approaches ranging from exact methods to metaheuristic optimization across three distinct dataset scales, we draw the following critical conclusions:

- **Scalability Limitations of Exact Methods:** Exact algorithms (Backtracking, Branch and Bound) and Constraint Programming proved effective only for small-scale instances ($N \leq 7, M \leq 15$), where they guarantee global optimality. However, due to exponential time complexity, these methods consistently failed to converge within the time limit (TLE) for medium and large-scale datasets, rendering them impractical for real-world warehouse scenarios with hundreds of items.
- **Trade-off in Greedy Strategy:** The Greedy algorithm offers superior computational speed, solving all instances in under 1 second. However, this comes at a significant cost to solution quality. In complex scenarios ($N = 20, M = 1000$), the Greedy approach yielded paths that were approximately **40% longer** than the optimal solutions found by metaheuristics, highlighting its susceptibility to local optima.
- **Robustness of Metaheuristics:** Local Search-based methods (Simulated Annealing, Tabu Search) successfully bridged the gap between speed and quality, significantly outperforming Greedy and Hill Climbing. Specifically, Tabu Search demonstrated high stability in medium-scale tests, often matching the best-known solutions.
- **Superiority of the Genetic Algorithm (GA):** The experimental results conclusively identify the Genetic Algorithm (specifically BRKGA) as the most effective method for this problem domain. GA not only matched the performance of other metaheuristics in medium tests but emerged as the clear winner in Large Scale instances ($M \geq 500$). It achieved the shortest travel distance (**15,291**) in the most complex test case, demonstrating an exceptional ability to escape local optima and explore the solution space effectively.

Future Work: Based on these findings, future research could focus on hybridizing the Genetic Algorithm with local search operator to further refine convergence speed, or exploring parallel computing techniques to reduce the execution time for ultra-large-scale warehouse configurations.