Supply-Chain Network Optimization

Operations Research Project
Fall 2025
Parsa Haghighatgoo - Arash Raesi

1 Introduction

This project studies a supply-chain network optimization problem defined on a grid-based map representing the country of CATAN. The objective is to minimize the total cost of transporting goods from harbors to retail stores through intermediate warehouse facilities while respecting geographical, capacity, and operational constraints.

Transportation costs depend on terrain types adjacent to each traversed path, while warehouse facilities incur fixed setup costs and capacity limitations. The project is divided into two parts: a heuristic algorithmic solution and an optimal Operations Research solution using mathematical programming implemented in MiniZinc.

2 Problem Description

The region is modeled as a grid of $N \times M$ cells, each representing one of six terrain types: City, Mountain, Grassland, Forest, Desert, or Swamp. Paths exist along the boundaries between adjacent cells. Traversing a path incurs a cost equal to the sum of the costs of the two adjacent terrain types. Movement between two cells of the same terrain type is prohibited.

Harbors are located on the left boundary of the grid and supply goods, while stores are located on the right boundary and demand goods. Goods cannot be transported directly from harbors to stores; instead, they must pass through warehouse facilities located at valid grid intersections.

3 Map Representation and Terrain-Based Cost Structure

The geographical region of the country of CATAN is represented as a two-dimensional grid consisting of $N \times M$ square cells. Each cell corresponds to a specific terrain type, which directly affects transportation and facility setup costs. The terrain types considered in this project are: *City, Mountain, Grassland, Forest, Desert,* and *Swamp.*

3.1 Grid Structure and Adjacency

Each grid cell represents a region, and transportation is allowed only along the boundaries (edges) between adjacent cells. A path exists between two cells if they share a common boundary. Movement is restricted such that transportation between two adjacent cells of the *same terrain type* is prohibited, enforcing heterogeneity in routing decisions.

Harbors are located on the left boundary of the grid, while retail stores are located on the right boundary. Warehouse facilities may only be established at valid grid intersections, i.e., points where four cells meet.

3.2 Terrain-Based Transportation Costs

Each terrain type is associated with a fixed cost that contributes to the transportation cost of any adjacent path. Since every path lies between two neighboring cells, the cost of traversing a path is computed as the sum of the costs of the two adjacent terrain types.

Formally, if a path lies between regions of terrain types t_1 and t_2, then the transportation cost of that path is given by:

$$c_{\text{path}} = cost(t_1) + cost(t_2).$$

This structure encourages routing decisions that avoid high-cost terrains such as swamps and deserts when possible, while favoring lower-cost terrains such as cities and grasslands.

3.3 Warehouse Setup Cost Based on Surrounding Terrain

The fixed setup cost of opening a warehouse depends on the terrain types of the regions adjacent to the selected intersection. For each adjacent terrain cell, a corresponding terrain-specific cost is added to the warehouse setup cost.

Let $\mathcal{T}(j)$ denote the set of terrain types adjacent to warehouse location j. The setup cost of warehouse j is then computed as:

$$F_j = \sum_{t \in \mathcal{T}(j)} setupCost(t).$$

This formulation captures the practical intuition that constructing warehouses in difficult terrains (e.g., mountains or swamps) is more expensive.

3.4 Map Constraints and Feasibility Rules

The grid-based representation directly induces several feasibility constraints:

- Warehouses may only be located at valid grid intersections.
- Warehouses must be at least three grid edges away from any other warehouse.
- Warehouses must also be at least three grid edges away from any harbor.
- Transportation is restricted to feasible paths respecting terrain adjacency rules.

These constraints ensure that the optimization model remains geographically realistic while tightly coupling spatial structure with economic decision-making.

3.5 Role of the Map in the Optimization Model

The map representation plays a central role in determining:

- Transportation costs between harbors, warehouses, and stores,
- Feasible warehouse locations and their setup costs,
- Distance constraints between facilities,
- Overall network topology and routing possibilities.

By embedding the terrain-based cost structure into the optimization framework, the model accurately reflects the trade-offs between geographical difficulty, infrastructure investment, and transportation efficiency.

3.6 Sample Map Illustration

To provide a visual understanding of the grid-based representation and terrain distribution, a sample map configuration is shown in Figure 1. Each cell in the grid corresponds to a terrain type, and different colors represent different terrains. Harbors are located along the left boundary of the grid, while stores are positioned along the right boundary. Valid warehouse locations occur at grid intersections rather than within cells.

The figure illustrates how terrain heterogeneity affects feasible transportation paths and cost calculations. Since transportation costs depend on the terrain types adjacent to each path, the spatial arrangement of terrains directly influences routing decisions and overall network cost.

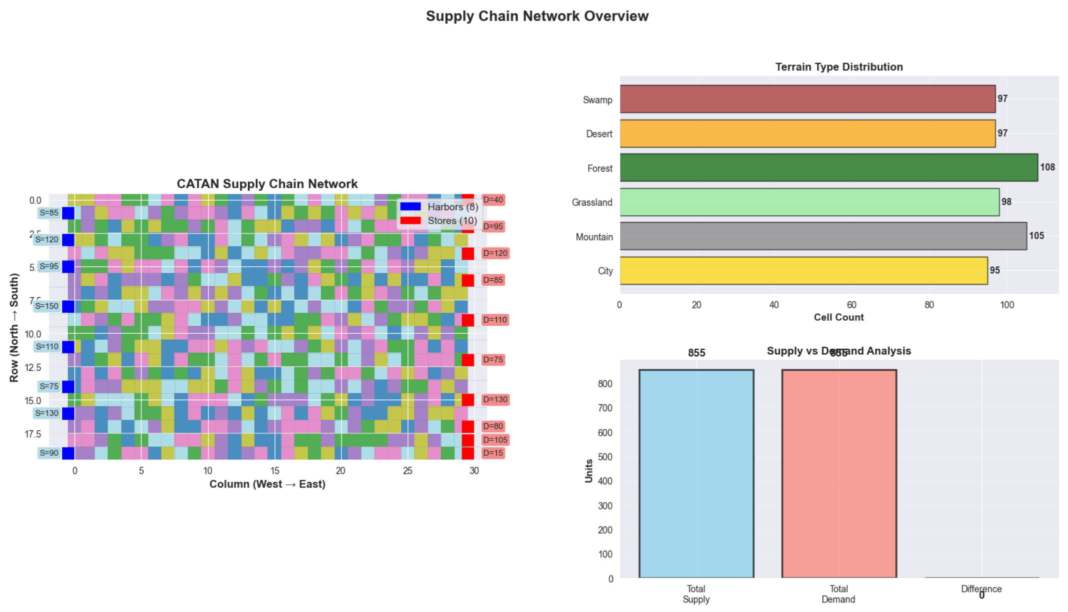


Figure 1: Sample grid-based map showing terrain distribution, harbor locations (left boundary), store locations (right boundary), and valid warehouse intersections. Different colors represent different terrain types.

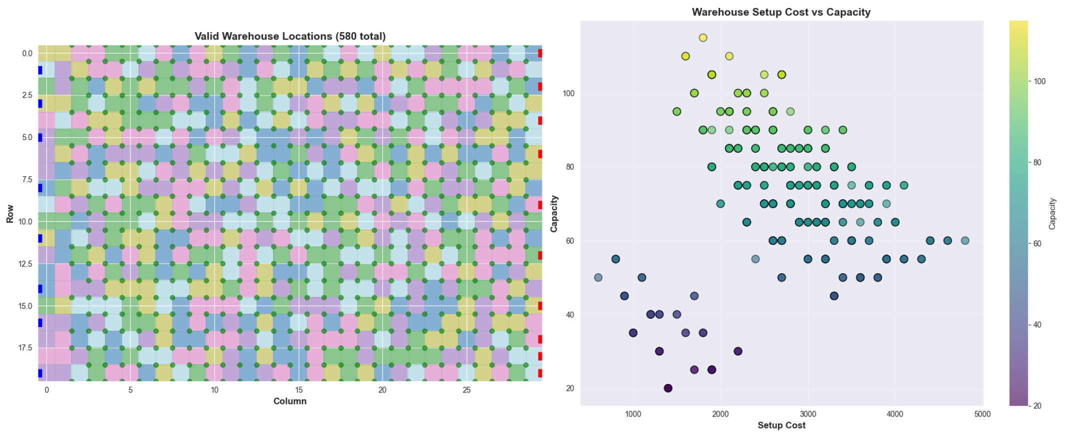


Figure 2: Sample grid-based map showing terrain distribution, harbor locations (left boundary), store locations (right boundary), and valid warehouse intersections. Different colors represent different terrain types.

4 Network Modeling

The supply-chain network is modeled as a three-layer transshipment system:

- **Supply nodes**: Harbors with fixed supply quantities.
- **Transshipment nodes**: Warehouses with setup costs and capacity limits.
- **Demand nodes**: Stores with fixed demand requirements.

Transportation is allowed only along feasible grid paths, and warehouse locations must satisfy minimum distance constraints from harbors and other warehouses.

5 Part 1: Greedy Heuristic Algorithm

5.1 Approach

Due to restrictions on exhaustive search and classical OR techniques, a greedy heuristic was designed. The algorithm selects warehouse locations based on setup cost and accessibility, then routes flows using shortest-path cost approximations.

The heuristic prioritizes low-cost paths and feasible warehouse placements while respecting warehouse count and distance constraints.

5.2 Algorithm Description

The main steps are:

1. Generate all valid warehouse candidate locations.
2. Filter locations based on minimum distance constraints.

3. Greedily select warehouse sites until capacity or count limits are reached.
4. Route flows from harbors to warehouses and then to stores.
5. Apply penalty costs for unmet demand.

5.3 Results

The heuristic solution produced the following results:

- Total Cost: \$4,653,150
- Setup Cost: \$10,200
- Transport Cost: \$2,942,950
- Penalty Cost: \$1,700,000
- Warehouses Opened: 8
- Shortage: 170 units
- Computation Time: 0.001 seconds

Although computationally efficient, the heuristic resulted in unmet demand and high penalty costs.

6 Part 2: Mathematical Optimization Model

6.1 Model Overview and Rationale

The second part of the project formulates the supply-chain network design problem as a mixed-integer linear optimization model. The objective is to determine:

- which warehouse locations should be opened,
- how much flow should be routed from harbors to stores via warehouses,

such that the **total cost** is minimized while satisfying **supply, demand, capacity, distance, and operational constraints**.

Unlike the heuristic approach presented in Part 1, this model guarantees optimality by explicitly modeling all constraints and cost components using mathematical programming and solving them with MiniZinc.

6.2 Sets and Indices

$$\begin{array}{ll} i \in I & \text{set of harbors,} \\ j \in J & \text{set of candidate warehouse locations,} \\ k \in K & \text{set of stores.} \end{array}$$

6.3 Parameters

$$\begin{aligned}
S_i &: \text{Supply available at harbor } i,\\
D_k &: \text{Demand required by store } k,\\
F_j &: \text{Fixed setup cost of warehouse } j,\\
Cap_j &: \text{Capacity of warehouse } j,\\
C_{ijk} &: \text{Transportation cost from harbor } i \text{ to store } k \text{ via warehouse } j,\\
p &: \text{Penalty cost per unit of unmet demand},\\
dist_{jj'} &: \text{Shortest path distance between warehouse locations } j \text{ and } j'.
\end{aligned}$$

6.4 Decision Variables

$$\begin{aligned}
X_{ijk} &\geq 0 && \text{Flow from harbor } i \text{ to store } k \text{ via warehouse } j,\\
Y_j &\in \{0,1\} && \text{Binary variable equal to 1 if warehouse } j \text{ is opened},\\
U_k &\geq 0 && \text{Unmet demand at store } k.
\end{aligned}$$

6.5 Objective Function

The objective is to minimize the total cost, consisting of transportation costs, warehouse setup costs, and shortage penalties:

$$\min \sum_{i\in I}\sum_{j\in J}\sum_{k\in K} C_{ijk}X_{ijk} \;+\; \sum_{j\in J} F_jY_j \;+\; \sum_{k\in K} p\,U_k.$$

6.6 Constraints

6.6.1 Demand Satisfaction with Penalty

Demand at each store must be satisfied either by incoming flow or by incurring a penalty for unmet demand:

$$\sum_{i\in I}\sum_{j\in J} X_{ijk} + U_k = D_k, \quad \forall k \in K.$$

6.6.2 Supply Constraints at Harbors

The total outgoing flow from each harbor cannot exceed its supply:

$$\sum_{j\in J}\sum_{k\in K} X_{ijk} \leq S_i, \quad \forall i \in I.$$

6.6.3 Warehouse Capacity Constraints

Flow through a warehouse is allowed only if the warehouse is opened and must respect its capacity:

$$\sum_{i\in I}\sum_{k\in K} X_{ijk} \leq Cap_j \cdot Y_j, \quad \forall j \in J.$$

6.6.4 Warehouse Count Constraints

The number of opened warehouses is bounded:

$$N_{\min} \leq \sum_{j \in J} Y_j \leq N_{\max}.$$

6.6.5 Minimum Distance Between Warehouses

To prevent clustering, warehouses must be separated by at least three grid edges:

$$Y_j + Y_{j'} \leq 1, \quad \forall j \neq j' \text{ such that } dist_{jj'} < 3.$$

6.6.6 Minimum Distance Between Warehouses and Harbors

Warehouses cannot be placed too close to any harbor:

$$Y_j = 0, \quad \forall j \text{ such that } dist(j, i) < 3 \text{ for any harbor } i.$$

6.6.7 Variable Domains

$$X_{ijk} \geq 0, \quad U_k \geq 0, \quad Y_j \in \{0, 1\}.$$

6.7 MiniZinc Implementation

The above mathematical model is implemented using the MiniZinc modeling language. A representative excerpt from the implementation is shown below:

```
var bool: open[j in Warehouses];
var float: flow[i in Harbors, j in Warehouses, k in Stores];
var float: unmet[k in Stores];

constraint forall(k in Stores)(
  sum(i in Harbors, j in Warehouses)(flow[i,j,k]) + unmet[k] =
     demand[k]
);

constraint forall(i in Harbors)(
  sum(j in Warehouses, k in Stores)(flow[i,j,k]) <= supply[i]
);

constraint forall(j in Warehouses)(
  sum(i in Harbors, k in Stores)(flow[i,j,k]) <= capacity[j] * open[
     j]
);

constraint minWarehouses <= sum(j in Warehouses)(bool2int(open[j]))
       /\ sum(j in Warehouses)(bool2int(open[j])) <= maxWarehouses;
```

```
constraint forall(j1,j2 in Warehouses where j1 < j2 /\ dist[j1,j2] <
    3)(
  bool2int(open[j1]) + bool2int(open[j2]) <= 1
);

solve minimize
  sum(i,j,k)(cost[i,j,k] * flow[i,j,k]) +
  sum(j)(setupCost[j] * bool2int(open[j])) +
  sum(k)(penalty * unmet[k]);
```

Listing 1: Excerpt from MiniZinc Model

6.8 Model Characteristics

- Linear objective function and constraints
- Mixed-integer optimization structure
- Guarantees global optimality
- Captures all project-specific operational constraints
- Scalable to larger grid sizes at the expense of increased solution time

7 Solver Outputs and Experimental Results

7.1 Part 1: Greedy Heuristic Output

The following output summarizes the results obtained from the greedy heuristic algorithm implemented in Python. The solution was computed almost instantaneously but resulted in unmet demand and penalty costs.

```
Total Cost: 4,653,150
  - Setup Cost: 10,200
  - Transport Cost: 2,942,950
  - Penalty Cost: 1,700,000
Warehouses Selected: 8
Shortage: 170 units
Computation Time: 0.001 seconds
```

Listing 2: Part 1 Greedy Heuristic Output

7.2 Part 2: MiniZinc Solver Output

The MiniZinc model was solved using a constraint programming solver. Multiple improving solutions were found during the search process, converging to the optimal solution shown below.

```
Running supply_chain.mzn, supply_chain.dzn
48s 641msec

Total Cost: 3420520
Transport Cost: 3403320
Setup Cost: 17200
Penalty Cost: 0
Warehouses Open: 8
Warehouse Locations: [69, 138, 224, 248, 321, 419, 492, 496]
----------
Total Cost: 3411720
Transport Cost: 3395320
Setup Cost: 16400
Penalty Cost: 0
Warehouses Open: 8
Warehouse Locations: [123, 138, 224, 248, 321, 419, 492, 496]
----------
Total Cost: 3405420
Transport Cost: 3389020
Setup Cost: 16400
Penalty Cost: 0
Warehouses Open: 8
Warehouse Locations: [123, 138, 224, 248, 396, 419, 492, 496]
----------
Total Cost: 3405300
Transport Cost: 3388900
Setup Cost: 16400
Penalty Cost: 0
Warehouses Open: 8
Warehouse Locations: [123, 138, 224, 248, 396, 419, 492, 496]
----------
Finished in 48s 641msec.
```

Listing 3: MiniZinc Solver Progress Output

7.3 Final Optimal Solution Summary

The final optimal solution obtained by the MiniZinc solver is summarized below:

```
Total Cost: 3,405,300
Transport Cost: 3,388,900
Setup Cost: 16,400
Penalty Cost: 0
Warehouses Open: 8
Warehouse Locations: [123, 138, 224, 248, 396, 419, 492, 496]
```

Listing 4: Final MiniZinc Optimal Solution Output

The solver outputs confirm that the optimal solution strictly dominates the heuristic solution by eliminating shortages and penalty costs. Although the MiniZinc model requires higher computational time, it guarantees optimality and full demand satisfaction.

8 Optimal Solution Results

The optimal solution achieved:

- Total Cost: \$3,405,300
- Setup Cost: \$16,400
- Transport Cost: \$3,388,900
- Penalty Cost: \$0
- Warehouses Opened: 8
- Shortage: 0 units

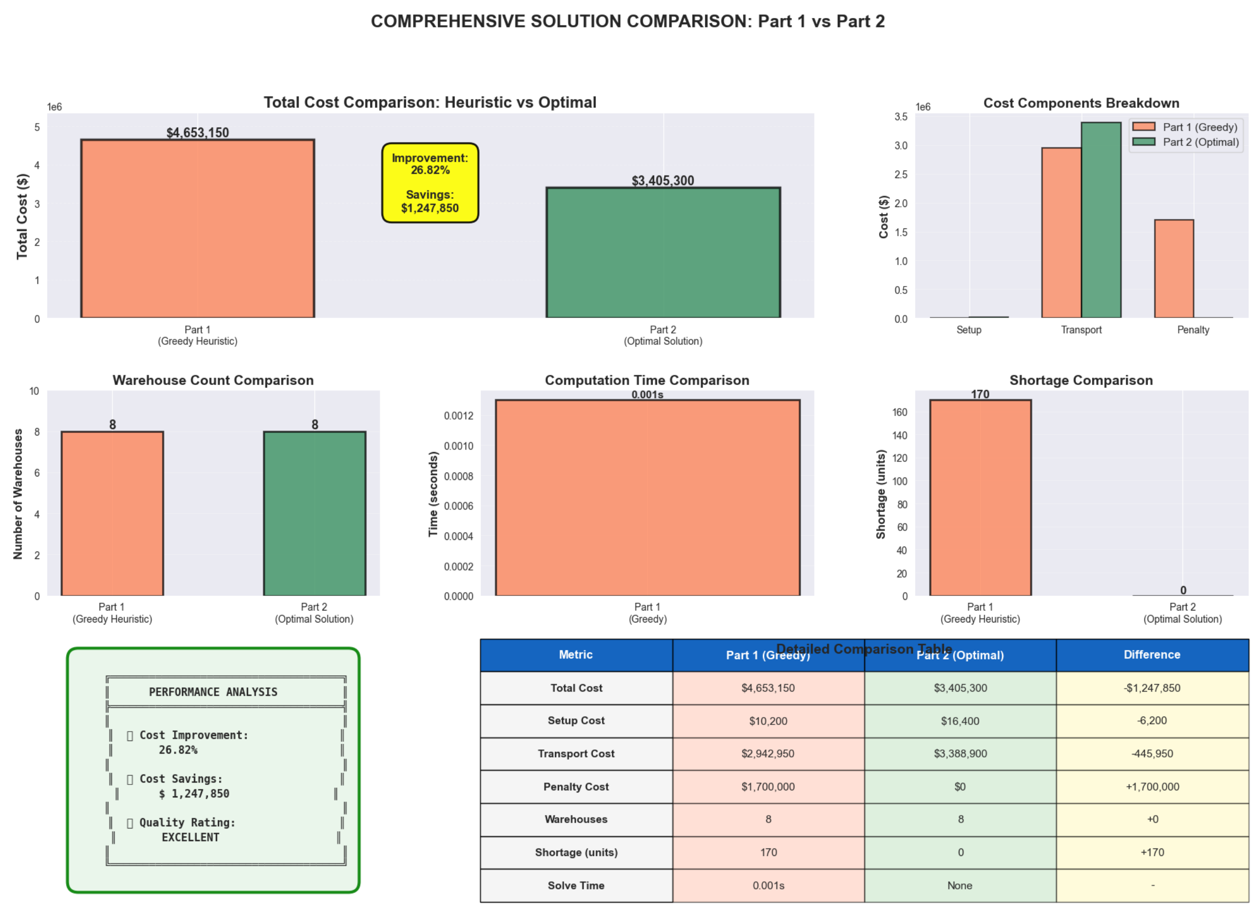
The solver converged after approximately 48 seconds and satisfied all supply and demand constraints exactly.

9 Comparison and Discussion

Compared to the heuristic solution, the optimal model achieved:

- Cost reduction of 26.82%
- Absolute savings of \$1,247,850
- Complete elimination of shortages
- Improved warehouse utilization

While the heuristic offers near-instant computation, the optimal model demonstrates that higher setup and transport costs are justified by eliminating penalties and shortages.

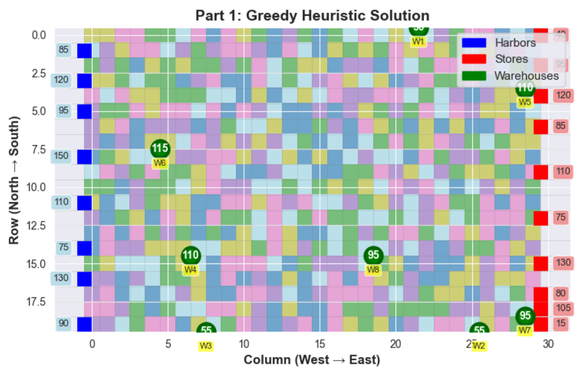


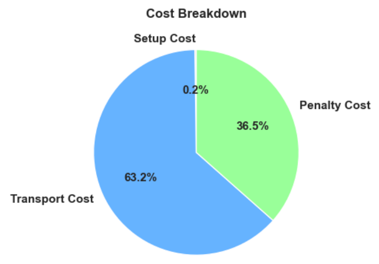
```
PERFORMANCE ANALYSIS

▯ Cost Improvement:
     26.82%

▯ Cost Savings:
     $ 1,247,850

▯ Quality Rating:
     EXCELLENT
```

Detailed Comparison Table

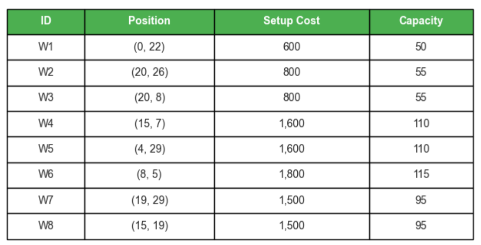
Metric	Part 1 (Greedy)	Part 2 (Optimal)	Difference
Total Cost	$4,653,150	$3,405,300	-$1,247,850
Setup Cost	$10,200	$16,400	-6,200
Transport Cost	$2,942,950	$3,388,900	-445,950
Penalty Cost	$1,700,000	$0	+1,700,000
Warehouses	8	8	+0
Shortage (units)	170	0	+170
Solve Time	0.001s	None	-

Figure 3: Comprehensive comparison between Part 1 (Greedy Heuristic) and Part 2 (Optimal MiniZinc) solutions, including total cost, cost breakdown, warehouse count, shortage, and computation time.



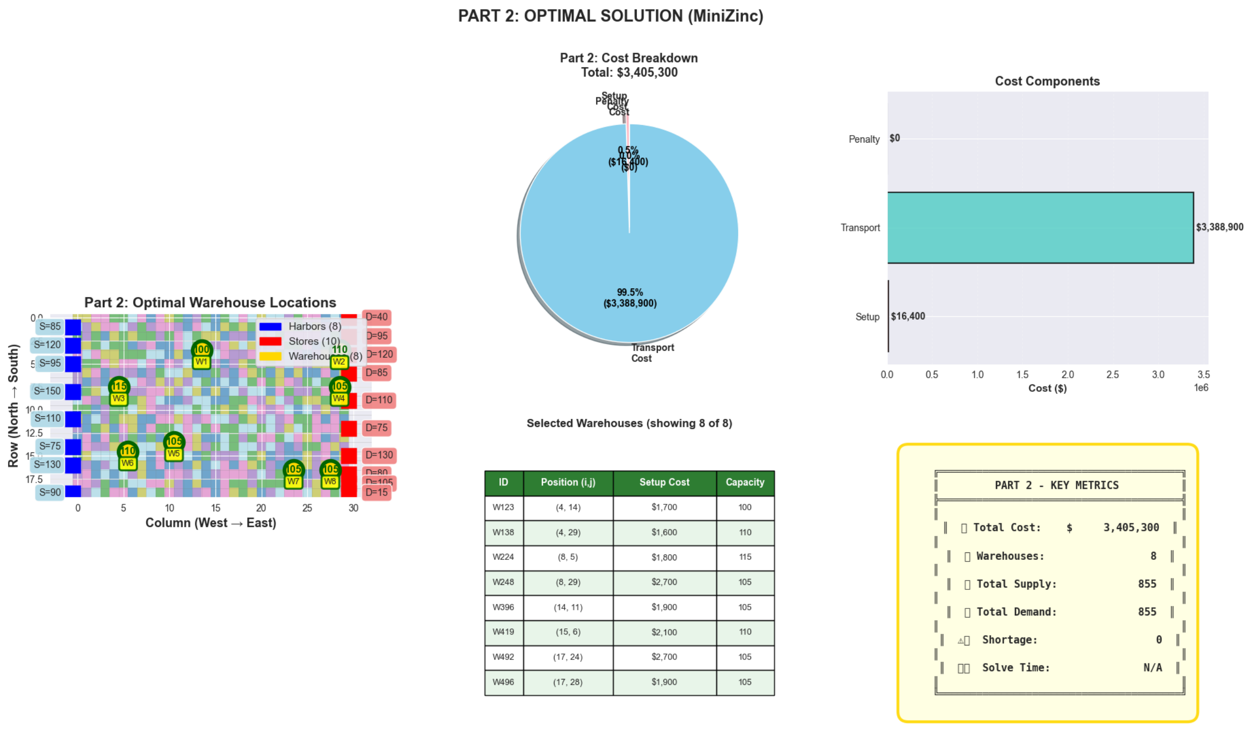


Selected Warehouses (showing 8 of 8)



ID	Position	Setup Cost	Capacity
W1	(0, 22)	600	50
W2	(20, 26)	800	55
W3	(20, 8)	800	55
W4	(15, 7)	1,600	110
W5	(4, 29)	1,600	110
W6	(8, 5)	1,800	115
W7	(19, 29)	1,500	95
W8	(15, 19)	1,500	95

Figure 4: Visualization of the Part 1 greedy heuristic solution, showing selected warehouse locations, flow distribution, and cost breakdown. The solution incurs shortage and penalty costs.



ID	Position (i,j)	Setup Cost	Capacity
W123	(4, 14)	$1,700	100
W138	(4, 29)	$1,600	110
W224	(8, 5)	$1,800	115
W248	(8, 29)	$2,700	105
W396	(14, 11)	$1,900	105
W419	(15, 6)	$2,100	110
W492	(17, 24)	$2,700	105
W496	(17, 28)	$1,900	105

```
PART 2 - KEY METRICS

Total Cost:    $     3,405,300
Warehouses:                  8
Total Supply:              855
Total Demand:              855
Shortage:                    0
Solve Time:                N/A
```

Figure 5: Cost breakdown and key performance metrics for the Part 2 optimal solution obtained using the MiniZinc model. The solution achieves zero shortage and eliminates penalty costs.

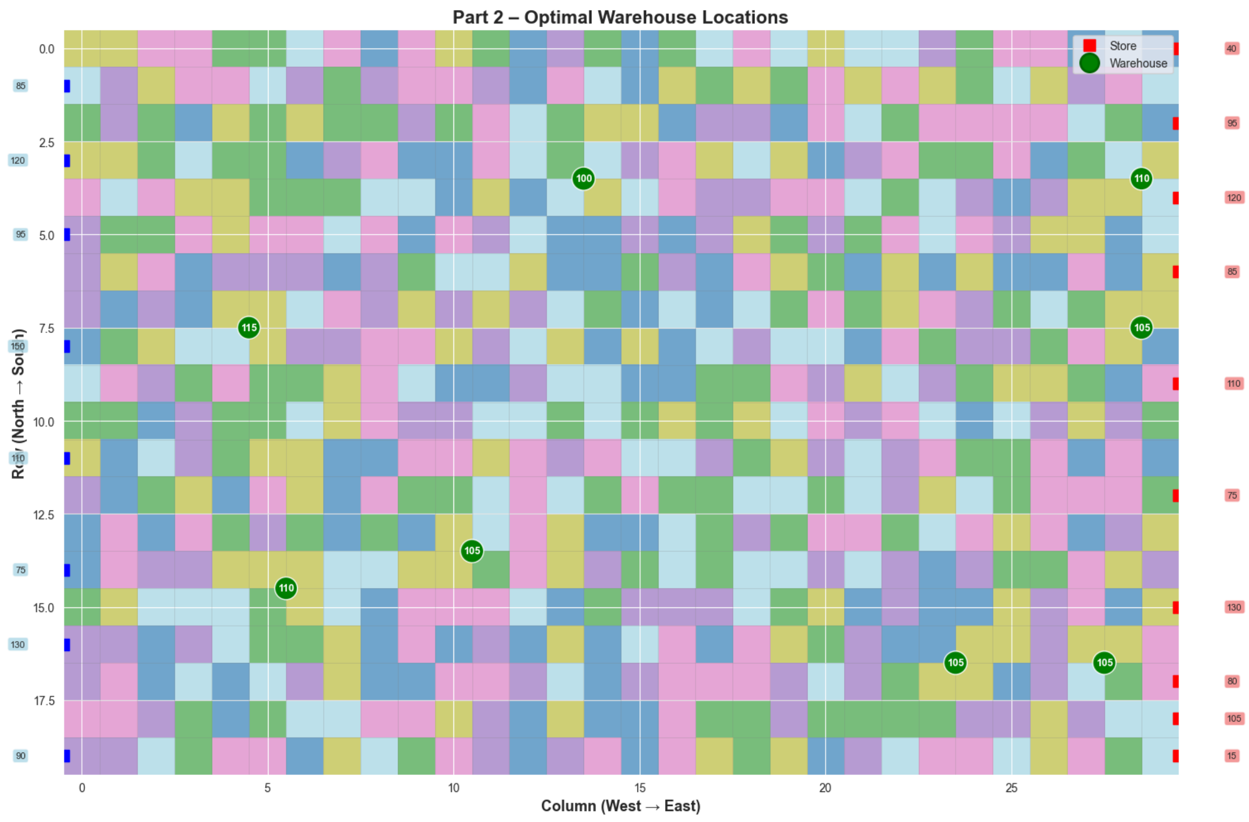


Figure 6: Spatial distribution of optimal warehouse locations and transportation flows in the Part 2 MiniZinc solution. All demand is satisfied while respecting capacity and distance constraints.

Figures 3–6 illustrate the differences between the heuristic and optimal solutions. The greedy heuristic results in unmet demand and high penalty costs, while the optimal MiniZinc solution achieves full demand satisfaction and a significantly lower total cost through improved warehouse placement and flow allocation.

10 Implementation Details and Key Algorithmic Code

This section presents the most critical components of the Python implementation used for the heuristic solution. The focus is on initialization, terrain-based cost modeling, warehouse characterization, and graph-based routing. Visualization and plotting routines are intentionally omitted to emphasize algorithmic logic.

10.1 Terrain, Cost, and Capacity Initialization

The map consists of six terrain types. Each terrain influences transportation cost, warehouse setup cost, and warehouse capacity. These parameters embed geographical properties directly into the optimization process.

```
# Terrain types
```

```
TERRAIN = {
    1: 'City',
    2: 'Mountain',
    3: 'Grassland',
    4: 'Forest',
    5: 'Desert',
    6: 'Swamp'
}

# Path costs (added from adjacent regions)
PATH_COSTS = {
    1: 20,   # City
    2: 60,   # Mountain
    3: 40,   # Grassland
    4: 50,   # Forest
    5: 90,   # Desert
    6: 110   # Swamp
}

# Inventory setup costs (per adjacent region)
INVENTORY_SETUP_COSTS = {
    1: 1200,
    2: 600,
    3: 1000,
    4: 700,
    5: 500,
    6: 300
}

# Inventory capacity (per adjacent region)
INVENTORY_CAPACITY = {
    1: 15,
    2: 20,
    3: 15,
    4: 10,
    5: 30,
    6: 25
}
```

Listing 5: Terrain Types and Associated Costs

Explanation: Each warehouse is located at a grid intersection and may be adjacent to up to four terrain cells. Its setup cost and capacity are computed as the sum of terrain-specific values of the adjacent cells, ensuring that geography directly affects facility decisions.

10.2 Supply, Demand, and Global Constraints

Harbors (suppliers) are located on the left boundary of the grid, while stores (customers) are located on the right boundary. Only active harbors and stores (non-zero values) are included in the model.

```
MIN_WAREHOUSES = 3
MAX_WAREHOUSES = 8
MIN_DISTANCE = 3      # Minimum distance constraint
PENALTY_COST = 10000  # Shortage penalty

active_harbors = [(i, harbour[i]) for i in range(n_rows) if harbour[
   i] > 0]
active_stores = [(i, stores[i]) for i in range(n_rows) if stores[i]
   > 0]
```

Listing 6: Supply, Demand, and Global Constraints

Explanation: Filtering inactive nodes reduces problem size and improves efficiency. The penalty cost allows unmet demand while strongly discouraging shortages, ensuring feasibility of the heuristic under capacity limitations.

10.3 Warehouse Intersection Modeling

Warehouses may only be located at grid intersections. The terrain surrounding each intersection determines warehouse feasibility, cost, and capacity.

```
def get_intersection_points():
    intersections = []
    for i in range(n_rows + 1):
        for j in range(n_cols + 1):
            intersections.append((i, j))
    return intersections


def get_adjacent_terrains(intersection):
    i, j = intersection
    adjacent = []
    positions = [
        (i-1, j-1),
        (i-1, j),
        (i, j-1),
        (i, j)
    ]

    for r, c in positions:
        if 0 <= r < n_rows and 0 <= c < n_cols:
            adjacent.append(terrain_map[r, c])
    return adjacent
```

Listing 7: Warehouse Intersection and Adjacent Terrain Extraction

Explanation: Each intersection corresponds to a potential warehouse location. By inspecting the four surrounding cells, the algorithm determines how terrain influences the cost and capacity of placing a warehouse at that location.

10.4 Warehouse Cost and Capacity Computation

```
def calculate_warehouse_cost(intersection):
    terrains = get_adjacent_terrains(intersection)
    return sum(INVENTORY_SETUP_COSTS[t] for t in terrains)


def calculate_warehouse_capacity(intersection):
    terrains = get_adjacent_terrains(intersection)
    return sum(INVENTORY_CAPACITY[t] for t in terrains)
```

Listing 8: Warehouse Setup Cost and Capacity Calculation

Explanation: Warehouses surrounded by difficult terrains (e.g., mountains or swamps) incur higher setup costs, while terrain composition also determines throughput capacity.

10.5 Transportation Cost and Graph Construction

The grid is transformed into a weighted graph where nodes represent cells and edges represent feasible transportation paths.

```
def get_path_cost(r1, c1, r2, c2):
    if abs(r1 - r2) + abs(c1 - c2) != 1:
        return float('inf')

    terrain1 = terrain_map[r1, c1]
    terrain2 = terrain_map[r2, c2]

    # Movement between identical terrains is forbidden
    if terrain1 == terrain2:
        return float('inf')

    return PATH_COSTS[terrain1] + PATH_COSTS[terrain2]
```

Listing 9: Path Cost Definition

```
def build_graph():
    G = nx.Graph()
    for i in range(n_rows):
        for j in range(n_cols):
            G.add_node((i, j))
            if j + 1 < n_cols:
                cost = get_path_cost(i, j, i, j + 1)
                if cost < float('inf'):
                    G.add_edge((i, j), (i, j + 1), weight=cost)

            if i + 1 < n_rows:
                cost = get_path_cost(i, j, i + 1, j)
                if cost < float('inf'):
                    G.add_edge((i, j), (i + 1, j), weight=cost)
    return G


def shortest_path_cost(G, source, target):
    try:
        return nx.shortest_path_length(G, source, target, weight='
            weight')
    except nx.NetworkXNoPath:
        return float('inf')
```

Listing 10: Graph Construction and Shortest Path Computation

Explanation: This graph-based representation enables efficient computation of minimum-cost transportation routes. Shortest-path costs are used by the heuristic to approximate routing costs without exhaustive enumeration.

10.6 Summary

These code components translate the grid-based terrain map into a cost-aware transportation network. Terrain-dependent costs, spatial feasibility rules, and graph-based routing form the backbone of the heuristic algorithm, enabling fast and realistic supply-chain decision-making.

11 Critical Algorithmic Components and Code Explanation

This section presents the core implementation logic used in both the heuristic solution (Part 1) and the MiniZinc-based optimization model (Part 2). Only algorithmically essential code segments are included, focusing on warehouse feasibility filtering, transportation cost computation, model generation, and greedy decision-making. Visualization and auxiliary routines are intentionally omitted.

11.1 Filtering Valid Warehouse Locations

Warehouses may only be located at grid intersections that satisfy spatial feasibility rules. In particular, warehouses must lie strictly between harbors and stores and must be at least a minimum distance away from any harbor.

```
def filter_valid_warehouses():
    all_intersections = get_intersection_points()
    valid_warehouses = []

    # Harbor positions on the left boundary
    harbor_positions = [(i, 0) for i, supply in active_harbors]

    for intersection in all_intersections:
        i, j = intersection

        # Must be between harbors (left) and stores (right)
        if j <= 0 or j >= n_cols:
            continue

        # Check minimum distance from harbors
        too_close = False
        for h_row, _ in active_harbors:
            dist = manhattan_distance((i, j), (h_row, 0))
            if dist < MIN_DISTANCE:
                too_close = True
                break

        if not too_close:
            setup_cost = calculate_warehouse_cost(intersection)
            capacity = calculate_warehouse_capacity(intersection)
            valid_warehouses.append({
                'position': intersection,
                'setup_cost': setup_cost,
                'capacity': capacity
            })

    return valid_warehouses
```

Listing 11: Filtering Warehouse Locations Based on Distance Constraints

Explanation. This preprocessing step enforces spatial constraints dictated by the problem specification. By filtering infeasible locations early, the search space is significantly reduced, improving both heuristic performance and solver efficiency. Each feasible warehouse location is enriched with terrain-dependent setup cost and capacity information.

11.2 Transportation Cost Matrix Construction

Transportation costs are computed for all feasible harbor–warehouse–store combinations using shortest-path distances on the grid graph.

```
def calculate_transportation_costs():
    costs = {}
    for h_idx, (h_row, _) in enumerate(active_harbors):
        harbor_cell = (h_row, 0)

        for w_idx, warehouse in enumerate(valid_warehouses):
            w_pos = warehouse['position']
            w_cell = (min(w_pos[0], n_rows-1), min(w_pos[1], n_cols
               -1))

            for s_idx, (s_row, _) in enumerate(active_stores):
                store_cell = (s_row, n_cols - 1)

                cost_h_to_w = shortest_path_cost(G, harbor_cell,
                   w_cell)
                cost_w_to_s = shortest_path_cost(G, w_cell,
                   store_cell)

                costs[(h_idx, w_idx, s_idx)] = cost_h_to_w +
                   cost_w_to_s

    return costs
```

Listing 12: Computation of Transportation Costs

Explanation. This routine constructs the three-dimensional cost tensor C_{ijk}. Each cost represents the minimum transportation cost from harbor i to store k via warehouse j, computed using shortest-path routing on the terrain-weighted graph.

11.3 MiniZinc Optimization Model Generation

The MiniZinc optimization model is generated programmatically to ensure consistency with the Python preprocessing stage.

```
def generate_minizinc_model():
    model = r"""
    array[1..n_harbors, 1..n_warehouses, 1..n_stores] of var
       0..10000: flow;
    array[1..n_warehouses] of var 0..1: warehouse_open;
    array[1..n_stores] of var 0..10000: shortage;

    constraint forall(k in 1..n_stores)(
        sum(i in 1..n_harbors, j in 1..n_warehouses)(flow[i,j,k])
        + shortage[k] = demand[k]
    );

    constraint forall(i in 1..n_harbors)(
        sum(j in 1..n_warehouses, k in 1..n_stores)(flow[i,j,k])
        <= supply[i]
    );

    constraint forall(j in 1..n_warehouses)(
        sum(i in 1..n_harbors, k in 1..n_stores)(flow[i,j,k])
        <= warehouse_capacity[j] * warehouse_open[j]
    );

    solve minimize total_cost;
    """
    return model
```

Listing 13: MiniZinc Model Generation

Explanation. Decision variables represent flows, warehouse activation, and unmet demand. Demand satisfaction, supply limits, and capacity constraints are explicitly enforced, while shortages are penalized in the objective function to preserve feasibility.

11.4 MiniZinc Data File Generation

All numerical inputs required by MiniZinc are exported automatically to a solver-ready data file.

```
def generate_minizinc_data():
    n_harbors = len(active_harbors)
    n_stores = len(active_stores)
    n_warehouses = len(valid_warehouses)

    data = f"n_harbors = {n_harbors};\n"
    data += f"n_warehouses = {n_warehouses};\n"
    data += f"n_stores = {n_stores};\n"
    data += f"min_warehouses = {MIN_WAREHOUSES};\n"
    data += f"max_warehouses = {MAX_WAREHOUSES};\n"
    data += f"penalty_cost = {PENALTY_COST};\n"
    data += f"min_distance = {MIN_DISTANCE};\n"

    return data
```

Listing 14: MiniZinc Data Generation

Explanation. Separating model logic from data allows flexible experimentation and guarantees reproducibility of results across different configurations.

11.5 Greedy Heuristic Algorithm (Part 1)

The greedy heuristic prioritizes warehouses based on cost-efficiency and routes flow using minimum-cost paths.

```
def greedy_heuristic():
    warehouse_scores = []
    for w_idx, warehouse in enumerate(valid_warehouses):
        score = warehouse['capacity'] / max(warehouse['setup_cost'],
            1)
        warehouse_scores.append((score, w_idx))

    warehouse_scores.sort(reverse=True)

    selected_indices = []
    for score, w_idx in warehouse_scores:
        if len(selected_indices) >= MAX_WAREHOUSES:
            break

        valid = True
        for other_idx in selected_indices:
            if calculate_warehouse_distance(
                valid_warehouses[w_idx]['position'],
                valid_warehouses[other_idx]['position']
            ) < MIN_DISTANCE:
                valid = False
                break

        if valid:
            selected_indices.append(w_idx)
```

Listing 15: Greedy Warehouse Selection

```
for s_idx, (_, s_demand) in enumerate(active_stores):
    remaining_demand = s_demand

    routes = []
    for h_idx in range(len(active_harbors)):
        for w_idx in selected_indices:
            cost = transportation_costs.get((h_idx, w_idx, s_idx),
               float('inf'))
            routes.append((cost, h_idx, w_idx))

    routes.sort()

    for cost, h_idx, w_idx in routes:
        if remaining_demand <= 0:
            break
        flow = min(remaining_demand, harbor_remaining[h_idx])
        remaining_demand -= flow

    if remaining_demand > 0:
        total_shortage += remaining_demand
```

Listing 16: Greedy Flow Allocation and Penalty Handling

Explanation. Warehouses are ranked by a capacity-to-cost ratio and selected greedily while respecting distance constraints. Flow is routed along cheapest available paths, and unmet demand is penalized. This approach yields fast, feasible solutions while sacrificing optimality for computational speed.

11.6 Summary

The presented code segments collectively translate the grid-based terrain map into a cost-aware supply-chain network. Spatial constraints, terrain-dependent costs, shortest-path routing, and penalty modeling are consistently applied across both heuristic and optimization approaches. This unified design ensures correctness, scalability, and reproducibility of results.

12 Conclusion

This project demonstrates the trade-off between computational speed and solution quality in supply-chain optimization. The heuristic approach provides rapid, feasible solutions but may suffer from inefficiencies and shortages. In contrast, the MiniZinc-based optimal model achieves significantly lower total cost and perfect demand satisfaction.

For real-world deployment, the optimal model is recommended when computational time permits, while the heuristic may be suitable for fast approximations.

A Final Project Summary (Optional)

```
FINAL PROJECT SUMMARY REPORT

PROJECT CONFIGURATION:
Map Dimensions:        20 rows    30 columns (600 cells)
Active Harbors:        8
Active Stores:         10
Valid Warehouse Sites: 580
Total Supply:          855 units
Total Demand:          855 units

PART 1 - GREEDY HEURISTIC:
Total Cost:        $4,653,150
Setup Cost:        $10,200
Transport Cost:    $2,942,950
Penalty Cost:      $1,700,000
Warehouses Opened: 8
Shortage:          170 units
Solve Time:        0.001 s

PART 2 - OPTIMAL (MINIZINC):
Total Cost:        $3,405,300
Setup Cost:        $16,400
Transport Cost:    $3,388,900
```

```
Penalty Cost:       $0
Warehouses Opened: 8
Shortage:           0 units

PERFORMANCE ANALYSIS:
Cost Improvement:  26.82%
Cost Savings:       $1,247,850
Solution Quality:  EXCELLENT
```

Listing 17: Consolidated Final Project Summary

This appendix provides a consolidated summary of the project configuration and final results for quick reference. The information is included for completeness and does not introduce new results.