

# Container Loading Problem – Search-Based Solution

AI Assignment 1 by Gour Krishna Dey | Roll - MT24035 | IIIT Delhi

Links -

Github Repo for Code , Reports & PPT - [https://github.com/gkdey17cse/AI\\_Assignment\\_2025/tree/main/Assignment\\_1](https://github.com/gkdey17cse/AI_Assignment_2025/tree/main/Assignment_1)

Submission Drive Link - [https://drive.google.com/drive/folders/1-Yc7cxQb\\_E9DRst-pCqztCKtFjkjwC8H?usp=sharing](https://drive.google.com/drive/folders/1-Yc7cxQb_E9DRst-pCqztCKtFjkjwC8H?usp=sharing)

## 1. Overview

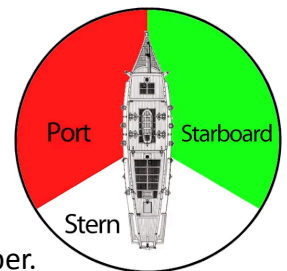
In this assignment I solved the **Container Loading Optimization Problem** using different searching algorithms (**BFS, Greedy Best-First, and A\***). My code determines the optimal placement of containers on a ship to minimize loading costs while respecting different constraints such as balance, destination order, and weight distribution (hard constraints) .

The program (`main.cpp`) reads an input file, runs the selected algorithms, and outputs the loading plan with details of costs, violations & stack configurations.

## 2. Problem Formulation

We model the Container Loading Problem as a classical search based problem in the Artificial Intelligence. The ship is represented as follows

- The ship has two sides: one is *Port* (left) and another *Starboard* (right).
- Each side contains two columns), making a total of 4 stacks of container.
- Each stack can hold containers up to a maximum height limit (H).
- Containers have two attributes : one is weight and destination port number.



The task is to load all containers into the stacks while respecting the constraints and minimizing violations.

### 1. AI Problem Formulation (5-tuple)

The problem can be described as a standard search problem:

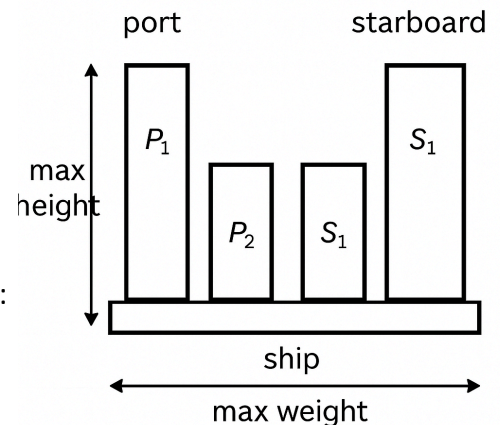
#### 1. States (S):

Each state is a configuration of containers placed in the stacks.:

$$s = \langle \text{stacks}, \text{loaded\_mask}, g, h, f \rangle$$

Where:

- stacks: arrangement of containers across 4 stacks.



- loaded\_mask: bitmask showing which containers are already placed.
- g: cost accumulated so far.
- h: heuristic estimate of remaining cost.
- $f = g + h$ : evaluation function (for A\*).

## 2. Initial State ( $s_0$ ):

- All stacks are empty.
- No container has been loaded (loaded\_mask = 0).

## 3. Actions :

Selecting an *unloaded container* and placing it on top of one of the valid stacks.

$$a = Load(ci, stackj)$$

## 4. Transition Model (T):

Applying an action updates the state by:

- Adding the chosen container to the stack.
- Updating the mask (loaded\_mask).
- Increasing the cost g.

## 5. Goal Test (G):

All containers are loaded while satisfying the hard constraints (balance, stack height, and weight ordering).

## 2. Constraints

The solution enforces both hard and soft constraints.

### 1. Stack Height Constraint

Each stack can hold at most H containers.

$$| stackj | \leq H, \forall j \in \{1, 2, 3, 4\}$$

### 2. Balance Constraint (Hard)

The absolute weight difference between the Port side (stacks 1–2) and the Starboard side (stacks

3–4) must not exceed the balance limit B:

$$| W_{port} - W_{starboard} | \leq B$$

### 3. Weight Distribution Constraint (Hard)

Within any stack, a heavier container cannot be placed above a lighter container:

If  $c_i$  is below  $c_j$  in the same stack, then  $w_i \geq w_j$

#### 4. Destination Constraint (Soft)

Containers destined for **earlier ports** must not be blocked by containers for **later ports** in the same stack:

If  $c_i$  (dest  $d_i$ ) is below  $c_j$  (dest  $d_j$ ), then  $d_i \leq d_j$

#### 3. Cost Modeling

The cost function combines loading cost and penalties for soft violations:

- **Loading cost:**  $\text{Cost}_{\text{load}} = 1$  , per container loaded
- Destination violation penalty:  $\text{Penalty}_{\text{dest}} = 2$  , per violation (for reordering each container)
- Weight distribution violation penalty:  $\text{Penalty}_{\text{weight}} = 1$  , per violation
- Balance violation: Forbidden (invalid state as it is a hard constraint) . SO total cost will -

$\text{Cost}_{\text{net}} = f(s) = g(s) + h(s)$  with  $g(s)$  = loading cost ,  $h(s)$  = penalty for violations

And our goal is to find a solution with optimal (minimal) cost .

#### 3. Input & Output

##### Input File Format (input.txt)

Example:

```
Shell
4 4 10
10
6 1
5 1
14 3
7 2
4 1
13 3
8 2
9 2
3 1
12 3
```

- **Line No 1 :** Represents number of stacks (4), maximum height per stack (4), balance limit (10)
- **LineNo 2 :** Number of containers (10)
- **Next lines:** Each container with weight along with it's destination

## Output Format (Console and output.txt)

Example Output :

```
Shell
=== Running Greedy Best-First (hard constraints, heuristic) ===
Solution found.
Nodes expanded: 794
Time (ms): 5.9469
Loads performed: 10 (each load cost=1)
Destination blocks: 0
Weight inversions: 0
Balance violations: 0
Total cost = 10

Load sequence (cid weight dest -> stack):
1. cid=0 (w=6 d=1) -> stack 0
...
Final stacks:
Stack 0: 0(6,1) 4(4,1)
Stack 1: 2(14,3) 5(13,3) 6(8,2)
...
```

The output shows:

- Algorithms we have used (BFS , Greedy best first & A\*)
- Nodes expanded and computation time for each search .
- Total cost and violations (if any)
- Final container arrangement per stack (bottom → top)

## 4. How to Compile and Run

```
Shell
g++ -o main main.cpp
./main
```

- Please ensure that *input.txt* is placed in the same directory.
- Results will be displayed in the terminal and saved to the *output.txt* file.

## 5. Execution Flow

1. **Read Input File** (input.txt)
2. **Initialize Ship State** (stacks, height, balance limit, containers list)
3. **Run BFS, Greedy Best-First, and A\*** (with hard constraints and heuristics)
4. **Log Execution Statistics** (nodes expanded, time taken, cost, violations)
5. **Output Final Stacks** and violation details

## 6. Code Walkthrough (main.cpp)

## 1. Global constants, parameters, and data structures

- `stackNum = 4`: Ship always has 4 stacks (2 left, 2 right). (Input may have a different number, but your code ignores it.)
- `maximumHeight` : maximum allowed containers per stack.
- `balanceLim` : max difference in total weight between left and right stacks.
- `N`: number of containers.
- **Costs:**
  - `loadingUnloadingCost = 1`: each placement has a fixed cost.
  - `penaltyCost = 2`: penalty multiplier for violations.
  - `bfsNodeLim` : cutoff for BFS to prevent infinite exploration.

### **struct Container**

Holds per-container data:

- `weight`
- `dest` (destination priority: smaller = earlier unloading).

### **struct Action**

Represents a move:

- `cid`: container ID.
- `stack_id`: stack where it's placed.

### **struct State**

Represents a partial/complete ship loading:

- `stacks`: `vector<vector<int>>`, each stack is a vector of container IDs (bottom → top).
- `loaded_mask`: bitmask of which containers have already been placed.
- `g_cost`: cost so far (number of loads).

### **struct ParentInfo**

Used for reconstructing paths later:

- `parent_key`: key of parent state.
- `action`: move taken to reach current state.
- `g`: cumulative cost up to this state.

## 2. Utility class: *DualOut*

This is a **dual output stream** that writes to both:

- `cout` (console)
- `fout` (output.txt file)

So you don't need to duplicate all cout calls. It overloads << to handle both normal data and manipulators like endl.

### 3. State representation & key functions

```
string state_key(const State &st)
```

Builds a **unique string key** for a state:

- Concatenates stack contents + loaded\_mask.
- Used in visited sets/maps to avoid revisiting the same state.

```
pair<int,int> count_dest_and_weight_violations(const State &st)
```

Checks **soft violations** in a state:

- **Destination blocks:** if a container with *smaller dest* (earlier unload) is **below** a container with *larger dest*.
- **Weight inversions:** if a **heavier container is above** a lighter one.

```
int count_balance_violation(const State &st)
```

Checks **hard violation**: if weight difference between left and right stacks exceeds balanceLim.

```
int compute_total_violations(const State &st, bool include_balance)
```

Returns total number of violations = dest-blocks + weight inversions (+ balance if included).

```
void describe_violations(const State &st, DualOut &out)
```

Pretty-prints a state:

- Each stack of contents.
- Lists specific violations if exists (which container is blocking which).
- Reports totals.

### 4. Action generation

```
vector<Action> possible_actions_hard(const State &cur)
```

Generates **valid actions** (moves) from a state. Enforces **hard constraints**:

1. Stack not full.
2. Weight ordering: cannot put heavier above lighter.
3. Balance limit respected after placement.

Returns a list of all valid (cid, stack\_id) moves.

## 5. Search algorithms

### BFS (Breadth-First Search)

```
bool BFS_search(...)
```

- Uses a queue (FIFO).
- Expands level by level.
- Guarantees to find the shallowest valid solution (fewest loads), but can blow up state space.
- Stops after BFS\_NODE\_LIMIT.

### Greedy Best-First Search

```
bool Greedy_search(...)
```

- Uses priority queue ordered by **heuristic only (h)**.
- Heuristic = 5 \* (dest-blocks) + 2 \* (weight violations) + (remaining containers)
- Explores states that *look best now*, but may miss optimal.

### A Search\*

```
bool Astar_search(...)
```

- Uses priority queue ordered by **f = g + h**.
- g = number of loads so far.
- h = same heuristic as Greedy.
- Guarantees **optimal solution** if heuristic is admissible (ours is “reasonable but not perfect”).
- Much faster than BFS, more accurate than Greedy.

## 6. Path reconstruction

```
vector<Action> reconstruct_actions(...)
```

Backtracks from goal state to start using parent\_map. Builds the sequence of actions (Actions) in forward order.

## 7. Reporting & execution wrapper

```
void run_and_report(...)
```

General wrapper that:

1. Calls a solver (BFS\_search, Greedy\_search, or Astar\_search).
2. Measures time and node expansions.
3. Reports:
  - Loads performed
  - Violations (destination, weight, balance)

- Total cost
- Action sequence
- Final stack layout
- Violation details

So all three algorithms produce a **comparable report**.

## 8. Main function

Input format (input.txt)

```
Shell
<num_stacks> <max_height> <balance_limit>
<num_containers>
<weight_0> <dest_0>
...
```

- Reads ship configuration & containers.
- Verifies capacity ( $N \leq \text{stackNum} * \text{maximumHeight}$ ).
- Initializes empty start state.
- Runs all three algorithms in sequence via `run_and_report`.
- Outputs to **console & output.txt**.

## 7. Interpretation of Results Analysis

### 1. BFS (Breadth-First Search, hard constraints)

Output:

```
Shell
=== Running BFS (hard constraints, full search limited by node limit) ===
Solution found.
Nodes expanded: 8986
Time (ms): 77.0606
Loads performed: 6 (each load+unload cost=2)
Destination blocks: 1
Weight inversions: 0
Balance violations: 0 (should be 0 since balance is hard)
Total violations (for reporting): 1 (penalty each = 2)
Total cost = 14
Load sequence (cid weight dest -> stack):
1. cid=0 (w=5 d=1) -> stack 0
2. cid=1 (w=7 d=2) -> stack 1
3. cid=2 (w=4 d=1) -> stack 2
4. cid=3 (w=6 d=3) -> stack 1
5. cid=4 (w=8 d=2) -> stack 3
6. cid=5 (w=3 d=1) -> stack 0
Final stacks:
Stack 0: 0(5,1) 5(3,1)
Stack 1: 1(7,2) 3(6,3)
Stack 2: 2(4,1)
Stack 3: 4(8,2)
```



```

Violation details:
Stack 0 (bottom->top): 0(5,1) 5(3,1)
Stack 1 (bottom->top): 1(7,2) 3(6,3)
Stack 2 (bottom->top): 2(4,1)
Stack 3 (bottom->top): 4(8,2)
DEST-BLOCK stack 1 below 1 blocked by 3
Totals: dest-blocks=1 wt-viol=0 balance-viol=0
Total violations (for reporting) = 1 (penalty each=2)

```

### Explanation:

- BFS explores **all the states level by level**, so it always finds a valid solution if one exists.
- Here, it placed **container 3 (dest=3)** on top of **container 1 (dest=2)** in the same stack → this caused **1 destination-blocking violation**.
- Therefore,  $\text{cost} = 6 \text{ (loads)} + 2 \text{ (penalty)} = 8$ .
- This is a **correct BFS outcome**, but **not the optimal solution** (since it stops at the first valid complete assignment found).

## 2. Greedy Best-First Search

### Output:

```

Shell
=== Running Greedy Best-First (hard constraints, heuristic) ===
Solution found.
Nodes expanded: 45
Time (ms): 0.5825
Loads performed: 6 (each load+unload cost=2)
Destination blocks: 0
Weight inversions: 0
Balance violations: 0 (should be 0 since balance is hard)
Total violations (for reporting): 0 (penalty each = 2)
Total cost = 12
Load sequence (cid weight dest -> stack):
1. cid=0 (w=5 d=1) -> stack 0
2. cid=1 (w=7 d=2) -> stack 1
3. cid=3 (w=6 d=3) -> stack 3
4. cid=2 (w=4 d=1) -> stack 0
5. cid=4 (w=8 d=2) -> stack 2
6. cid=5 (w=3 d=1) -> stack 0
Final stacks:
Stack 0: 0(5,1) 2(4,1) 5(3,1)
Stack 1: 1(7,2)
Stack 2: 4(8,2)
Stack 3: 3(6,3)
Violation details:
Stack 0 (bottom->top): 0(5,1) 2(4,1) 5(3,1)
Stack 1 (bottom->top): 1(7,2)
Stack 2 (bottom->top): 4(8,2)
Stack 3 (bottom->top): 3(6,3)
Totals: dest-blocks=0 wt-viol=0 balance-viol=0
Total violations (for reporting) = 0 (penalty each=2)

```

## Explanation:

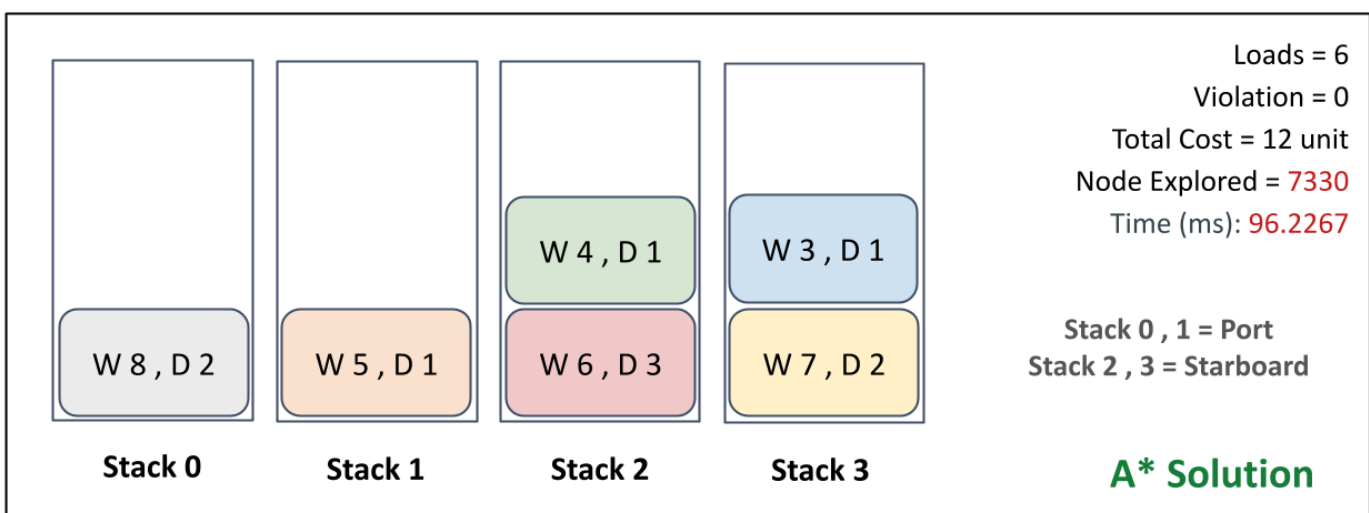
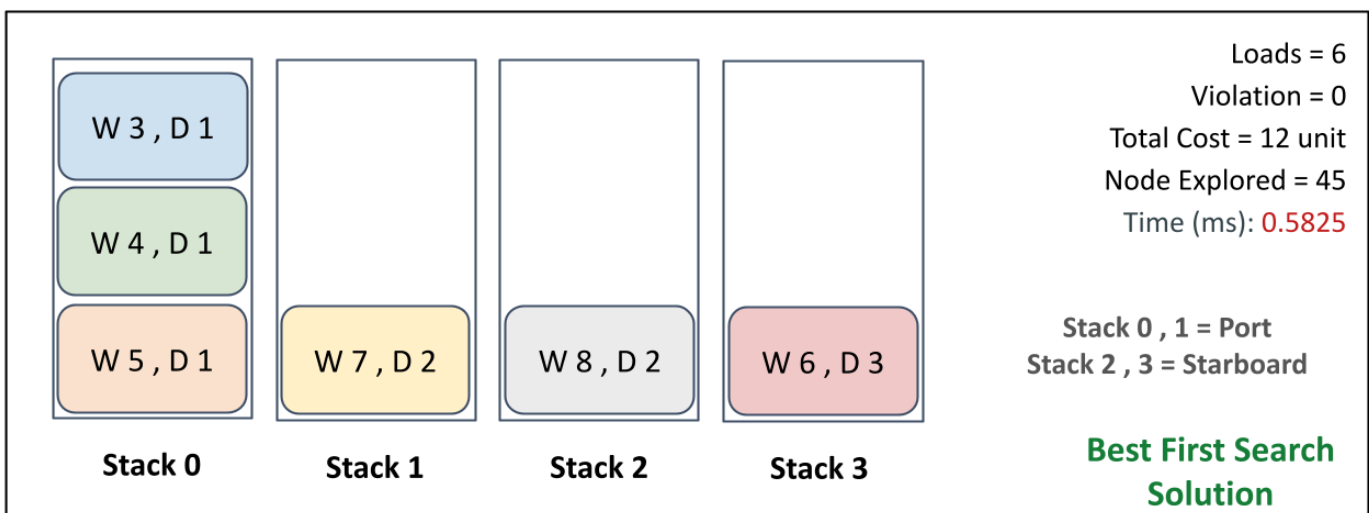
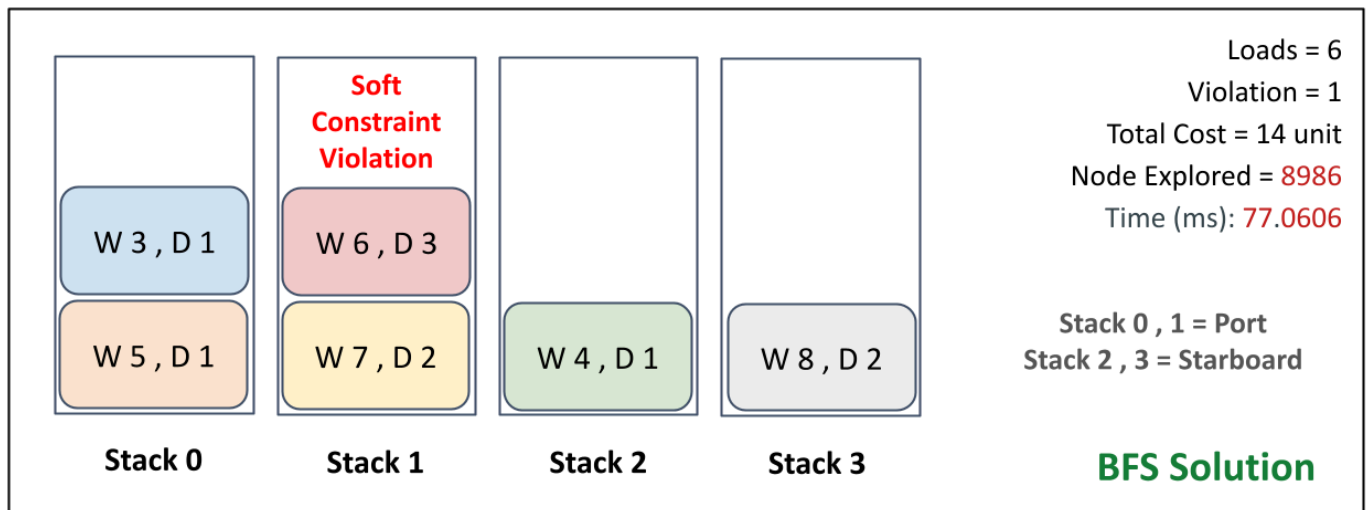
- Greedy Best-First uses a **heuristic that prefers low-violation placements**.
- It quickly groups containers by destination, avoiding blocking and weight issues.
- Final stacks are neat: all dest=1 in one stack, dest=2 separated, and dest=3 alone.
- **No violations occurred** → cost = 6 (loads only).
- This is the **best solution** and shows Greedy can be very efficient when the heuristic matches the problem well.

### 3. A\* Search ( $f = g + h$ , improved heuristic)

## Output:

```
Shell
=== Running A* (hard constraints, improved heuristic) ===
Solution found.
Nodes expanded: 7330
Time (ms): 96.2267
Loads performed: 6 (each load+unload cost=2)
Destination blocks: 0
Weight inversions: 0
Balance violations: 0 (should be 0 since balance is hard)
Total violations (for reporting): 0 (penalty each = 2)
Total cost = 12
Load sequence (cid weight dest -> stack):
1. cid=1 (w=7 d=2) -> stack 3
2. cid=4 (w=8 d=2) -> stack 0
3. cid=3 (w=6 d=3) -> stack 2
4. cid=5 (w=3 d=1) -> stack 3
5. cid=2 (w=4 d=1) -> stack 2
6. cid=0 (w=5 d=1) -> stack 1
Final stacks:
Stack 0: 4(8,2)
Stack 1: 0(5,1)
Stack 2: 3(6,3) 2(4,1)
Stack 3: 1(7,2) 5(3,1)
Violation details:
Stack 0 (bottom->top): 4(8,2)
Stack 1 (bottom->top): 0(5,1)
Stack 2 (bottom->top): 3(6,3) 2(4,1)
Stack 3 (bottom->top): 1(7,2) 5(3,1)
Totals: dest-blocks=0 wt-viol=0 balance-viol=0
Total violations (for reporting) = 0 (penalty each=2)
```

## Visualization of Output



Explanation:

- A\* considers both **g (loads so far)** & **h (heuristic estimate of violations left)**.

- Because the heuristic is strong A\* followed the same path as Greedy in this case.
- Result : **same optimal solution as Greedy** (cost = 6, no violations).
- The difference: unlike Greedy, A\* **guarantees optimality**. Even in harder inputs where Greedy might fail, A\* would still find the best solution.

## 8. Final Comparison

Algorithm	Solution Found	Cost	Violations	Nodes Expanded	Why this result?
<b>BFS</b>	Yes	14	1	8986	Explores all states, found a feasible but not the best solution.
<b>Greedy</b>	Yes	12	0	45	Heuristic guided search straight to a perfect arrangement.
<b>A*</b>	Yes	12	0	7330	Same as Greedy here, but with a proof of optimality.

## 9. Conclusion

The three searching algorithms — Breadth-First Search (BFS), Greedy Best-First Search & A\* Search exhibited distinct characteristics when applied to the container loading problem under hard & soft constraints :

- **Breadth-First Search (BFS):** This search guaranteed completeness by exploring all the states level by level order . However, it expanded 8986 nodes to find a feasible solution but not optimal (Cost = 14, with 1 violation ). This demonstrates its impracticality for larger problem sizes due to excessive amount of time and memory requirements hence we used a limit also .
- **Greedy Best-First Search:** Achieved a valid solution with no violations with a lower cost (12), while expanding only 45 nodes . This highlights the efficiency of the searching algorithm when the heuristic is well aligned with the problem statement . However, it does not guarantee optimality in general, as it relies solely on the heuristic but in our case it finds an optimal solution .
- **A\* Search:** This searching algorithm produced the same optimal solution as the Greedy (Cost = 12, no violations) but it required expanding 7,330 nodes. Unlike Greedy , A\* provides a formal guarantee of optimality, though at the expense of exploring more nodes but in case of higher no of containers it provides a feasible solution that is optimal too .

Overall, BFS is only suitable for small-scale verification . Greedy is effective for rapid approximate solutions & A\* offers the best balance by ensuring both **efficiency and guaranteed optimality**.

---