Jeremiah McDonald

Combinatorics

23 April 2025

Dr. Dunshee

Combinatoric: Final Project

Optimizing Transfers

Project Goals:

- Accept any number of containers with custom capacities and starting volumes.
- Allow user to set the initial configuration.
- Enable users to define a flexible target goal state.
- Support transfer operations between any two containers (partial or full).
- Uses a graph search algorithm to find the shortest sequence.
- Print each step clearly with source, destination, and volume transferred and total cost.
- Create generic, reusable code that can apply to a variety of problems such as:
  - Memory partitioning: Optimally dividing memory blocks across processes or threads.
  - Fuel tank rerouting: Determining efficient transfer of fuel between multiple tanks or vehicles.
  - Warehouse distribution: Balancing or redistributing goods between storage units or delivery routes.
  - Game logic puzzles: Solving fill-the vial or potion mixing puzzles where resources must be balanced.
- Support interactive exploration or allow listing multiple valid solutions.
- Extra features:
  - Weighted costs for each transfer.
  - Support for constraints (min/max amounts, blocked paths, sender rules).
  - Show all valid paths up to a given depth.
  - Heuristic support for faster A* performance.
  - Add a GUI (visual animation) for clarity.

Java Components: Updated

1. Container Class
   a. Represents an individual container (capacity, current volume)
   b. Tracks:
      i. Capacity(fixed)
      ii. Current volume
      iii. Can be extended for per-container constraints or cost rules.

2. State Class
    a. Represents a full system state: volumes of all containers.
    b. Implements:
        i. equals() and hashCode() for tracking visited states
        ii. toString() for logging and visualization.
    c. Central to comparing, caching, and pathfinding logic.
3. TransferAction Class
    a. Records a single transfer step from one container to another:
    b. Stores:
        i. source, destination, and volume
        ii. weight (cost)
    c. Used by A* and output display.
4. MoveResult Class
    a. Bundles:
        i. A new State
        ii. The Transfer that led to it.
    b. Returned by state-generation logic and used by the solver.
5. AStar (Solver) Class
    a. Contains core A* search logic.
    b. Features:
        i. Supports cost-based optimization via transfer weights.
        ii. Accepts optional transfer constraints via TransferConstraint interface.
        iii. Supports any GoalCondition (expression, menu, lambda).
        iv. Can return the shortest path or all valid paths up to a max depth. (As well as all paths)
6. GoalCondition (Interface + Lambda + Parser)
    a. Defines when a state satisfies the goal.
    b. Implemented as:
        i. Predefined goals (ExactMatch, SingleContainer, EvenDistribution)
        ii. Custom lambda expressions
        iii. Advanced math expression parsing (ex: v[0] + v[2] >= 30)
7. TransferConstraint (Interface)
    a. Defines rules to block invalid transfers.
    b. Examples:
        i. Max transfer size
        ii. Forbidden container routes
        iii. No transfers to/from certain containers
    c. Fully composable using lambdas.
8. Main Class
    a. Handles user input.
    b. Features:
        i. Interactive containers setup
        ii. Goal selection (menu + custom input)

iii. Constraint setup (repeat loop + validation)
iv. Solving strategy options (shortest vs. all)
v. Total cost output for all results
vi. Visualization toggle (console vs. GUI)
vii. Option to export to file
viii. Option to restart the session.

9. Visualizer
    a. Formats and displays each transfer step in console.
    b. Tracks cumulative state across steps.
10. TransferGUI (Swing-based GUI)
    a. Visualizes transfer sequence in a scrollable window.
    b. Allows replaying each step with cost and volume states.

This project combines combinatorics, optimization, and state-space search to solve complex transfer problems in both real-world and abstract scenarios. This design ensures reusability and adaptability across domains.

What This System Can Solve (Capability Table)

Note: **Menu Goal = Selected via built-in options in the CLI menu.**

Expression Goal = The user provides a custom formula.

Problem	Menu Goal	Expression Goal	Notes
Exact match for all containers	☑	☑	Standard use case
Single container goal (Volume equals X)	☑	☑	Useful in puzzles
Even distribution	☑	☑	Great for resource fairness
Total volume equals X	☑	☑	Useful for batching, limits
Volume of container[i] >= X	☑	☑	Capacity planning
Sum of v[i] + v[j] equals X	☑	☑	Mixing tasks
Sum of v[i…n]	✗	☑	Only via expression
Exactly N containers non-empty	✗	☑	Advanced condition
Prevent overfilling	☑	☑	Flexible via constraints
Disallow certain paths	☑	☑	Custom logic supported
Minimize total volume moved	☑	☑	Uses transfer weights

Container volumes form a specific ratio	✗	☑	Ex: v[0] == 2 * v[1]

Solves: Water jug puzzles, Memory partitioning, Fuel tank planning, Warehouse loading, Game-based logic challenges, Distributed resource optimization.