Project warehouse
Algoritmen voor beslissingsondersteuning

(B-KUL-JPI30V)

Benjamin Rübenkamp, Arne Vanmarcke

December 20, 2024

Instructor: Manos Thanos, Annemie Vorstermans

1 Introduction

This report describes the structure of the program developed for managing a warehouse. The program utilizes various data structures to improve the efficiency and effectiveness of operations. This report discusses the main components and the data structures used, as well as the benefits these structures offer.

2 Core Architecture and Design Patterns

2.1 Overall Architecture

The system follows a modular architecture organized around several key components:

2.1.1 Core Domain Objects

- **Warehouse:** Central coordinator managing vehicles, requests, and operations
- **Vehicle:** Represents autonomous vehicles with movement and box handling capabilities
- **Stack:** Implements storage locations with LIFO (Last-In-First-Out) behavior
- **Bufferpoint:** Implements storage locations with no ordering
- **Request:** Encapsulates movement operations with source, destination, and box information
- **Graph:** Each stack and bufferpoint is linked to a specific location. This allows for the construction of a fully connected graph in advance. Using the Floyd-Warshall algorithm, the travel times between all stacks and bufferpoint locations can be established beforehand. These calculations are stored so that the travel times can be easily retrieved later without needing to be recalculated each time.

2.1.2 Strategy Pattern Implementation

The system employs the Strategy pattern for scheduling algorithms through the abstract SchedulingStrategy class. This allows for different scheduling implementations:

- **TopBoxSchedulingStrategy:** Handles accessible boxes at stack tops
- **StackToBufferSchedulingStrategy:** Manages movement from stacks to buffer zones
- **BufferToStackSchedulingStrategy:** Controls movement from buffers back to stacks

2.1.3 Data Access Layer

The DataReader class implements a clean separation between data access and business logic

2.2 Key Design Patterns

2.2.1 Factory Method Pattern

Our implementation uses factory-like construction in the DataReader to create complex objects from JSON data. Based on a provided file path, a JSON file is read. This file contains various types of data, such as vehicles, requests, stacks and bufferpoints, which are then converted into objects within the code.

2.2.2 State Pattern

Request processing uses a state pattern through the REQUEST_STATUS enum. This enables clear tracking of request progression through different stages.

2.2.3 Observer-like Pattern

While not a classic Observer pattern, our implementation has a monitoring system through operation logging.

2.3 Design Decisions

2.3.1 Separation of Concerns

The system's clear separation of concerns data structures like Graph and Stack, business logic in SchedulingStrategy, and I/O operations in DataReader ensures each component has a single responsibility, enhancing maintainability and flexibility.

2.3.2 Extensibility

The abstract SchedulingStrategy and the IStorage interface together enable seamless integration of new scheduling algorithms, diverse storage implementations, and expandable vehicle capabilities.

2.3.3 Error Handling

This implementation ensures robust error handling through null checks in critical operations, exception handling for I/O processes, and status tracking of operations.

3 Data Structures Used

Various data structures are used within the program to maximize efficiency:

- **Vehicle and Bufferpoint Storage:** Boxes in the storage of vehicles and Bufferpoints do not need to be retrieved in a specific order, wich makes an ArrayList suitable here.
- **Requests:** All requests that have not yet been completed are kept in a list and are sorted at runtime into a priority queue based on which vehicle should handle them or the height of the box described in the request in the stack.

3.1 Stack

The `Stack` data structure is used for managing the stacks of boxes in the warehouse. A `Stack` follows the LIFO (Last In, First Out) principle, which is ideal for modeling stacks where the top box must be removed first. This is technically efficient because adding and removing elements at the top of the stack takes constant time.

- **Stack Storage:** Boxes in the storage of a stack must be retrieved via LIFO (Last In, First Out), which is why a stack was used for this purpose.

Sorting Requests: When the next request needs to be chosen, the requests that have not yet started are sorted based on the distance to the available vehicle or the height of the box in a stack.
Locking stacks: To indicate that a stack is used we keep track of the time it is used until in a hashmap that maps a stackId to a time. If the current time is bigger then the time in the hashmap the stack is not used.
Keeping track of active relocations: To keep track of active relocations, so that we can temporarily pause a vehicle if 2 vehicles are relocation to each other in a loop. If that is the case we need to indicate that the paused vehicle has to wait for the other vehicle to finish it's request. We do this by using a hashmap that maps a requestID to a vehicleID. (waitForRequestFinish)

4 Scheduling Strategies and Request Distribution

4.1 Overview of Scheduling Strategies

The system implements a multi-phase scheduling approach with three main strategies:

4.1.1 TopBoxSchedulingStrategy

The system prioritizes easy moves, handling directly accessible boxes first, eliminating intermediate relocations. This approach is highly efficient in terms of vehicle movements but is constrained by box accessibility.

4.1.2 StackToBufferSchedulingStrategy

Handles blocked boxes that need to go from stack to buffer.

4.1.3 BufferToStackSchedulingStrategy

The system processes boxes stored in buffers. Because this happens last, no relocations of these boxes happen. We first make enough space for all boxes that need to go to that stack and then bring the boxes there one by one.

4.2 Request Distribution System

Requests get distributed in two manners:

- **Stack Sorting by Request Count**: The system sorts stacks based on the number of pending requests that need that stack in descending order.
- **Load Balancing**: Stacks are then distributed over all vehicles so the load is balanced.

By doing this (assigning stacks to one vehicle) we avoid vehicle conflicts as much as possible. This is further improved by first trying to relocate to a vehicles own stacks when needed.

5 Request handling

To better manage the progress of requests and allow for step-by-step completion, the request class has been given a Status field. This field is an enum value (INITIAL, SRC, SRC_RELOC, DEST, DEST_PU, DEST_RELOC).

In the **INITIAL** phase, it checks whether 1) it is at a stack and can free its vehicle so that it has its full capacity to proceed (status remains INITIAL), 2) the request's place location is a stack and is full (then first free up the destination) (status becomes DEST_PU because it is at the destination but picks up a box instead of dropping it off), or 3) the destination is a buffer or a stack with free space, then it can go to the source and pick up the top box. After this, the status changes to SRC.

In the **SRC** phase, it checks whether the current vehicle is holding the required box. If not, and the vehicle's capacity is not full, it picks up another box (status remains SRC). If it does not have the desired box and its capacity is full, it searches for storage to relocate these unnecessary boxes, goes there, and places the first box. (Status becomes INITIAL, where it again tries to unload as many boxes as possible at its current location before returning to the source). If it does have the desired box, it moves the vehicle to the destination and places the box there. (Status becomes DEST, indicating that the request is completed.)

In the **DEST_PU** phase, the vehicle simply searches for storage for the unnecessary box, moves there, and places the box. (Status returns to INITIAL, from which it will proceed to the source).

The phases SRC_RELOC and DEST_RELOC are used in the processing of a request solely for logging whether an unnecessary box has been moved for a relocation.

5.1 Finding temporary storage:

Our findNstorage() function finds stacks with free space in 3 tiers to avoid conflict as much as possible:

- **Tier 1:** Check vehicle's own assigned stacks
- **Tier 2:** Find available stacks (no vehicle assignments)
- **Tier 3:** Find stacks assigned to other vehicles

6 Conclusion

By implementing efficient data structures and managing boxes strategically, the algorithm achieves both speed and efficiency, minimizing the number of moves required.

Datafile	**Moves**	**Time**	**Computational Time (ms)**
I3_3_1	8	880	0
I3_3_1_5	8	180	0
I10_10_1	20	3040	0
I15_16_1_3	32	2670	0
I20_20_2_2_8b2	90	5980	0
I30_100_1_1_10	288	60740	6
I30_100_3_3_10	344	17055	1
I30_200_3_3_10	400	15030	2
I100_50_2_2_8b2	172	33385	6
I100_120_2_2_8b2	442	84530	260
I100_500_3_1_20b2	2130	333586	114
I100_500_3_5_20	2146	178094	24
I100_800_1_1_20b2	2698	1348906	61
I100_800_3_1_20b2	2702	459028	41

Table 1: Performance of our algorithm

A link to our project on github: `https://github.com/Benjamin4-23/warehouse`