SAXION HOGESCHOOL

Documentation

2.1 Complex Data Structures

Sandra Krevová 516044

DHI1V.So

Table of Contents

Introduction

This document outlines my implementation of various data structures, algorithms, and techniques for an CDS assignment. Each section, from linked lists to visual representations, details the specific approaches and tools I employed, along with the rationale behind my choices.

Linked List

For the implementation of the Linked List, I drew inspiration from a previous SDP exercise, where I already build my own Linked list in week 5. The code for this implementation can be found in the "lists" package, where I have defined a LinkedList interface that includes essential functions.

Doubly Linked List

My DoublyLinkedList implements the LinkedList interface. It features both head and tail nodes within its Node objects and keeps track of the current size of the list. I use my Doubly Linked List in my App.java program for storing the stations and tracks. In readStationsFromCSV(), readTracksFromCSV() I am simply adding stations and tracks in an unsorted manner. I also use it in my HashMap.

Linear search

It's designed to search for a specific item in a linked list by iterating through each element in the list one by one. Accessing the elements in the linked list is O(n) because, in the worst case, we might have to traverse the entire list to find the desired element.

My linearSearch() method is located in utils > ListUtils.java. This method will receive as an argument two parameters: head as a starting point of the list and condition. Condition is a predicate, essentially a function that takes an element of type T and returns a boolean value.

The method starts by setting a current node to the head of the list. It then enters a while loop, which continues as long as there are more elements in the list to search (i.e., current is not null). Inside the loop, it checks if the condition specified by the condition predicate is true for the data stored in the current node. If it's true, that means the desired element has been found and returned. If the loop completes without finding the element it simply returns null.

The core idea of this linear search is inspired by the provided pseudo-code from week 1. However, I adapted it to work with generic nodes instead of just integers. To handle the condition check within the current node, I turned to ChatGPT, which recommended using the condition.test(current.getData()) approach.

Implementation

```
Enter your choice: 1
Insert name to search: Deventer
Found station: Station Deventer with code dv (52.257499694824, 6.1605553627014)
```

Binary search

Used in the App.java module, known for its O(log n) time complexity, which is significantly more efficient than the O(n) complexity of linear search.

I integrated the binary search method into the SortedList class, making use of a comparator to order stations by their names. The process involves comparing the target station (key) with the middle element of the current search range. I use the comparator to determine whether the search should

continue in the upper or lower half of the range, effectively halving the search interval with each iteration.

In cases where the middle element matches the search key, the method immediately returns the found element. If no match is found, the method adjusts the search range based on the comparator's outcome, either narrowing it down to the upper or lower half. This process repeats until the element is found or the range is fully explored.

The core idea of this linear search is inspired by the provided pseudo-code from week 1 + prompt in ChatGPT to include the Comparator.

Implementation

```
Enter your choice: 3
Insert name to search: Enschede
Found station: Station Enschede with code es (52.222499847412, 6.8899998664856)
```

Sort

Both sorting algorithms can be found in sort -> Sort.java. Implementation can be found in App.java where I sort an ArrayList of connections by length using a comparator based on the track's distance. From the menu in App.java, option 4 and 5 are implementation of these sorting algorithms.

Merge sort

Logic: Merge sort recursively divides the list into halves until each sub list has one element, then merges sub lists in a sorted order. It involves splitting the list, sorting each half, and then merging them back together, ensuring overall sort order.

Big O Complexity: Time complexity is O(n log n) due to the recursive division and merging of the list, and space complexity is O(n) as it requires additional space for the merged lists.

Selection sort

Logic: Selection sort iteratively selects the smallest (or largest, depending on the order) element from the unsorted portion of the list and swaps it with the first unsorted element. It continuously moves the boundary of the sorted and unsorted parts of the list, selecting the next smallest element each time.

Big O Complexity: Time complexity is O(n^2) as it requires comparing each element with every other element, and space complexity is O(1) since it sorts the list in place without requiring additional storage.

Hash Map

A Hash map of stations with station code as key. MyHashMap is located in maps > MyHashMap.java. When I read stations in App.java I add the station code as a key and the station as a value to the map. In my custom implementation of MyHashMap, I specifically chose to use an array of DoublyLinkedList<Entry<K, V>> to effectively handle collisions using the chaining method. The choice of a doubly linked list allows more flexible traversal (both forward and backward) through the entries in each bucket, which is particularly advantageous when removing entries.

Implementation

User to enter a station code, retrieves the corresponding station from stationMyMap using the entered code, and then prints the station's details if found. It is not case-sensitive.

```
Enter your choice: 2
Insert stations code: dv
dv
Station Deventer with code dv (52.257499694824, 6.1605553627014)
```

Binary Tree

This class is defined in trees > BinarySearchTree.java. Every node in this tree can have up to two children, adhering to the binary search tree property: each node's left child holds a lesser value, and the right child holds a greater value.

AVL tree

In an AVL tree, the height difference (balance factor) between the left and right subtrees of any node is kept to a maximum of one. This ensures that the tree remains balanced, providing O(log n) time complexity for insertions, deletions, and lookups, where n is the number of nodes in the tree.

To be honest, with the AVL tree, I had a lot of help from ChatGPT because it didn't work at all from the beginning. Especially the rotations. The balance method is full of comments because I didn't understand from the code when its right or left heavy situation and what rotation is needed. I left all the comments there because without it I'm lost.

Implementation

```
Enter your choice: 9
diGraph AVLTree {
    gsb -> eahs;
    gsb -> mg;
    eahs -> bnn;
    eahs -> fie;
    bnn -> asdl;
    bnn -> bspd;
    asdl -> amr;
    asdl -> bf;
    amr -> albert;
    amr -> apd;
    albert -> ahpr;
    albert -> almo;
    ahpr -> ahbf;
    ahpr -> aixtgv;
```

From the menu, its possible to print avl tree with all the station codes.

Heap

The MinHeap constructor uses Array.newInstance for creating a dynamically typed array, addressing Java's generic type erasure limitations, I believe. I stole it from class actually.

Then, there are two important methods – percolate up and down. Both I managed to do with help from material (with the theory) and ChatGPT.
The percolateUp method ensures that the heap's structure and ordering properties are maintained when a new element is added. It repeatedly compares and potentially swaps the newly added element with its parent, moving it up the heap, until the element is correctly positioned according to the min-heap rule

The percolateDown method was more complicated. I first calculate the indices of the left and right children of the node at the given index. If either child is smaller than the node, that child's index becomes the new 'smallestIndex'. If the smallest index is not the original node's index, I swap the node with its smaller child. This process repeats recursively down the heap to ensure the parent node is always smaller than its children.

Graph

All classes can be found in “graphs” package.

The first class, AbstractGraph<V>, is a basic graph structure. It holds an array of vertices and provides fundamental operations such as initialization with a variable number of vertices, getting vertices, graph size, the index of a particular vertex, and checking if the graph is empty. This class serves as the foundation for more complex graph types.

Next is the MatrixGraph<V> class, which extends AbstractGraph<V>. This class represents a graph using a matrix to store connections between vertices and can handle both directed and undirected graphs. The connections are stored in a 2D boolean array, where a true value indicates a connection between vertices. The class offers functionalities to connect two vertices, check if two vertices are connected, and perform breadth-first and depth-first traversals.

Finally, the WeightedMatrixGraph<V> class, also extending AbstractGraph<V>, introduces edge weights, making it suitable for representing weighted graphs. In addition to storing connections, it uses a 2D double array to store the weights of the edges. This class allows adding edges with weights, getting the weight of an edge, and checking connections. It also includes a method to generate a string representation of the graph in GraphViz format, which is helpful for visualization purposes.

BFS

Can be found in MatrixGraph.java

My Breadth-First Search (BFS) implementation effectively traverses graph structures by exploring neighbors of each node before moving to the next level. Starting from a given node, it systematically visits all its immediate neighbors, then each of their neighbors, and so on. The process uses a queue to keep track of nodes to visit next and a set to record visited nodes, ensuring each node is processed only once. This method is particularly efficient for finding the shortest path in unweighted graphs, as it naturally explores nodes in increasing distance from the start node.

For the BFS, I used the pseudo-code from the material and ChatGPT:
breadthFirst(start_node)
create Set visited // visited nodes
create Queue queue // nodes to visit

enqueue start_node in queue
while queue is not empty
next = dequeue from queue
if next not in visited
process next // print or whatever you want to do with it
add next to visited
for every neighbor of next as neighbor
if neighbor not in visited
enqueue neighbor in queue

I asked if ChatGPT could explain to me how it exactly works and then I adapted it to my project with the help provided.

DFS

Can be found in MatrixGraph.java

My Depth-First Search (DFS) implementation navigates through graph structures by delving as deep as possible along each branch before backtracking. This method starts from a specified node, exploring as far as possible along each branch using a stack to remember the path. As each node is visited, it's marked, and its unvisited neighbours are added to the stack. The process continues by popping the stack to backtrack when no further exploration is possible, then progressing along alternate routes.

Dijkstra algorithm

It is located in utils > GraphUtils.java.

When the method is invoked, it initializes several structures: a map for distances, keeping track of the shortest known distance from the start station to each other station; a map for previous stations, noting the last station on the shortest path to a given station; a set for visited stations; and a priority queue.

Initially, every station is assigned an infinite distance, except for the start station, which is set to zero. This start station is then added to the queue. The algorithm then enters its main loop, where it repeatedly processes the station with the shortest distance from the queue. For the current station, if it's the goal, the path is traced back from the previous map and returned. If it's not the goal, the algorithm examines each of its unvisited neighbors. If a shorter path to a neighbor is found via the current station, the neighbor's distance is updated, it's placed in the queue, and its previous station is recorded.

Implementation

In App.java, specifically in case 7, this implementation offers users a reliable way to find the shortest path between any two stations within the network.

```
Enter your choice: 7
Enter the code of the first station: DV
Enter the code of the second station: ES
Shortest path (Dijkstra): [dv, dvc, hon, rsn, wdn, aml, amri, bn, hgl, esk, es]
```

A* algorithm

It is located in utils > GraphUtils.java. Initially, the heuristic function I implemented was not functioning as intended — it frequently chose suboptimal routes, leading the algorithm away from the most efficient path between stations. Then, I ask for guidance from ChatGPT and it provided me with the new heuristic function with geographical coordinates to calculate the straight-line distance between stations. I still have no idea if it is correct implementation, but the tests passed.

The method initializes with a closed set (for visited nodes), a priority queue (open set) sorted by the estimated total cost (f-score), a map for reconstructing the path (cameFrom), and maps for the actual cost from start (gScore) and estimated total cost (fScore). The heuristic function, crucial to the A* algorithm, estimates the cost from a station to the goal, helping prioritize stations closer to the goal. This estimation uses geographical coordinates to calculate a straight-line distance, which is efficient for real-world navigation.

The algorithm iteratively explores the most promising paths first, as determined by the f-score, a sum of the known cost from the start (gScore) and the heuristic estimate to the goal. When the goal station is reached, the method reconstructs the path from the cameFrom map, revealing the most efficient route.

Implementation

```
Enter your choice: 6
Enter the code of the first station: DV
Enter the code of the second station: ASD
Shortest path (A*): [dv, twl, apdo, apd, hvl, amf, brn, hvs, hvsm, bsmz, ndb, wp, dmn, assp, asdm, asd]
```

MCST algorithm – Prim

It is located in utils > GraphUtils.java. I didn't have problem with this code at all. I understood how it works early from the theory so later on, also with help of ChatGPT, I managed to implement it.

The algorithm starts by initializing a set to track vertices included in the MST and a priority queue to select the minimum weight edge at each step. The choice of a priority queue is strategic, as it efficiently manages the edges, always providing the least weight edge available for processing.

The process begins with an arbitrary start vertex, typically the first in the graph. This inclusion of the start vertex in the MST set and adding all its edges to the priority queue is a systematic way to begin expanding the MST.

The algorithm then iteratively selects the smallest edge from the priority queue. If the edge connects to a vertex not already in the MST, it is included. Continually adding the new vertex's edges to the queue and updating the MST set ensures that all vertices are eventually included in the MST.

Implementation

```
Enter your choice:
Enter the latitude of the first corner: 52.504722595215
Enter the longitude of the first corner:
6.0919442176819
Enter the latitude of the opposite corner: 51.843612670898
Enter the longitude of the opposite corner: 5.8522224426269
Minimum Track Connections:
Edge from ah to ahp with weight 1.0
Edge from ahp to ahpr with weight 2.0
Edge from ahpr to vp with weight 3.0
Edge from ahp to wtv with weight 4.0
Edge from vp to rh with weight 4.0
Edge from wtv to dvn with weight 4.0
Edge from dvn to zv with weight 5.0
Total Track Length: 23.0
```

Regular Expressions

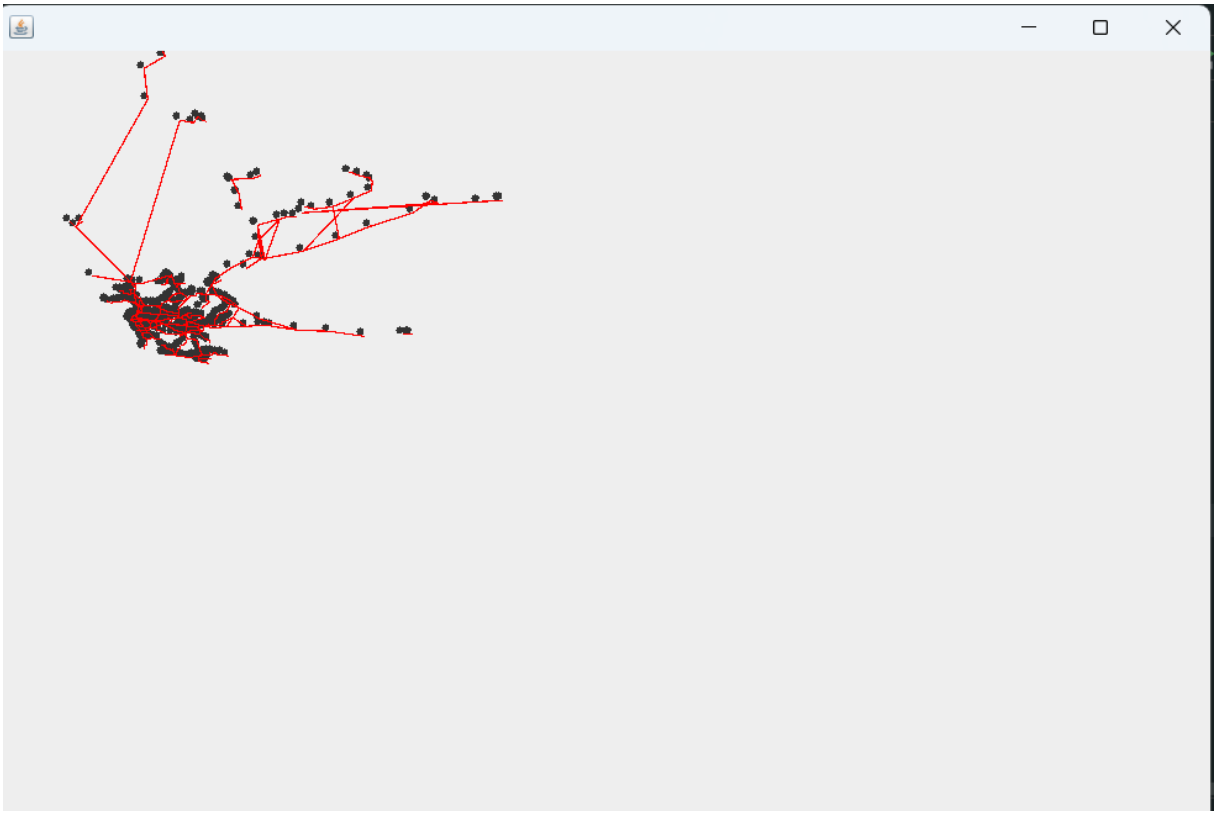
I used regular expression 2x in my project:

1. Reading stations from CSV (in App.java)
    a. ^(\\d+): This matches the beginning of the string (^) and then one or more digits (\\d+). The + means "one or more", and \\d is a digit (0-9).
    b. ([A-Z]+): This matches one or more uppercase letters.
    c. ([\\d]+): Similar to the first part, this matches a sequence of digits.
    d. ([^,]+): This is a bit more general. It matches any sequence of characters except a comma. The [^,] means "any character that is not a comma", and the + again means "one or more".
    e. The next two ([^,]+) are the same as the fourth part, used for additional fields with similar characteristics.
    f. ([\\w-]+): This matches a sequence of word characters (which includes letters, digits, and underscores) and hyphens.
    g. ([A-Z]+): Another sequence of one or more uppercase letters, similar to the second part.
    h. ([\\w-]+): Similar to the sixth part, for another field with alphanumeric characters and hyphens.
    i. ([\\-\\d.]+): This is used for matching decimal numbers, including possible negative values (hence the \\-). It matches digits, the minus sign, and the decimal point.
    j. ([\\-\\d.]+): Same as the ninth part, for another decimal number.
    k. $: This marks the end of the line.
2. Reading tracks from CSV (in App.java)
    a. ^: This asserts the start of a line. It ensures that the pattern matching starts from the beginning of each line in your CSV file.
    b. [a-z]+: This matches a sequence of lowercase letters (from 'a' to 'z'). The + means "one or more", so this part of the expression matches one or more consecutive lowercase letters.

c. ,: This is a literal comma. In CSV files, commas are typically used to separate fields. This part of the expression denotes the end of one field and the start of the next.
d. The second [a-z]+ is similar to the first one and matches another field consisting of one or more lowercase letters.
e. \\d+: This matches a sequence of digits (0-9). The \\d represents a digit, and the + means "one or more".
f. Two more occurrences of \\d+ follow, each separated by a comma. These are similar to the previous \\d+ and are used to match additional numeric fields. Each occurrence is intended for a separate field in the CSV.
g. $: This asserts the end of a line. It ensures that the pattern matching considers the entire line from start to finish.

With the help of ChatGPT and https://regexr.com/ I managed to check if every line is according to the pattern.

Visual Representation



This is my very simple visual representation of stations and connections in NL, probably the scaling is not done correctly but I didn't have more time to make it better looking. Also, some stations are missing.

The Drawing class in the assignment package extends JPanel and utilizes HashMap to store information about stations and their connections, both of which are populated from CSV files using a CsvReader.

Upon initialization, the constructor reads station data, including codes and geographic coordinates, converting them into screen coordinates for display. It also reads track data, detailing connections between stations and distances, populating the connections map.

The paintComponent method, an override from JPanel, handles the graphical rendering. It calculates a vertical offset for the map based on the minimum Y-coordinate of all stations, ensuring the map is properly positioned on the screen. Stations are drawn as filled circles, and connections as red lines between these points.

Conclusion

In this module, I deepened my Java proficiency by exploring algorithms and data structures, essential for efficient data storage, search, and modification. A key focus was on understanding the Big O notation, enabling me to evaluate algorithmic complexity and make informed decisions about choosing the most suitable data structures for specific problems.