SnackNow: Using Spatial Trees to optimize Food Costs for Lower Wage Families

Kannan Vishal[1], Prannaya Gupta[1], Quek Yu Pin[1], and Vikram Ramanathan[1]

[1]NUS High School of Math and Science

Contents

1 Background

Lower income families are plagued by food expenses nowadays, with it making up a significant proportion of their monthly expenditures. Frequently, one's choice of grocery-outlet is largely determined by convenience, e.g. going to the nearest *7-Eleven*. On the other hand, walking a bit further to the closest *FairPrice* is also a valid option, as compared to other nearby sources that tend to mark up their products. But what if such families enter a neighbourhood they are unfamiliar with? How can they be sure that they can get meals for affordable prices? In this report, we design a platform to present and compare such options in a relatively concise interface, that allows selection based on **both price and distance**. This selection is done via the use of Spatial R-Trees and QuadTrees. Of course, this is not limited to food; this platform can be extended to work with any type of commodity. In this way, users can make more informed decisions regarding their expenditures.

2 User Documentation

*You are to provide a brief user documentation with **screenshots** on how to use the program and its functionalities.*

Below is an image of the planned wireframes, which we have largely adapted into the actual application.

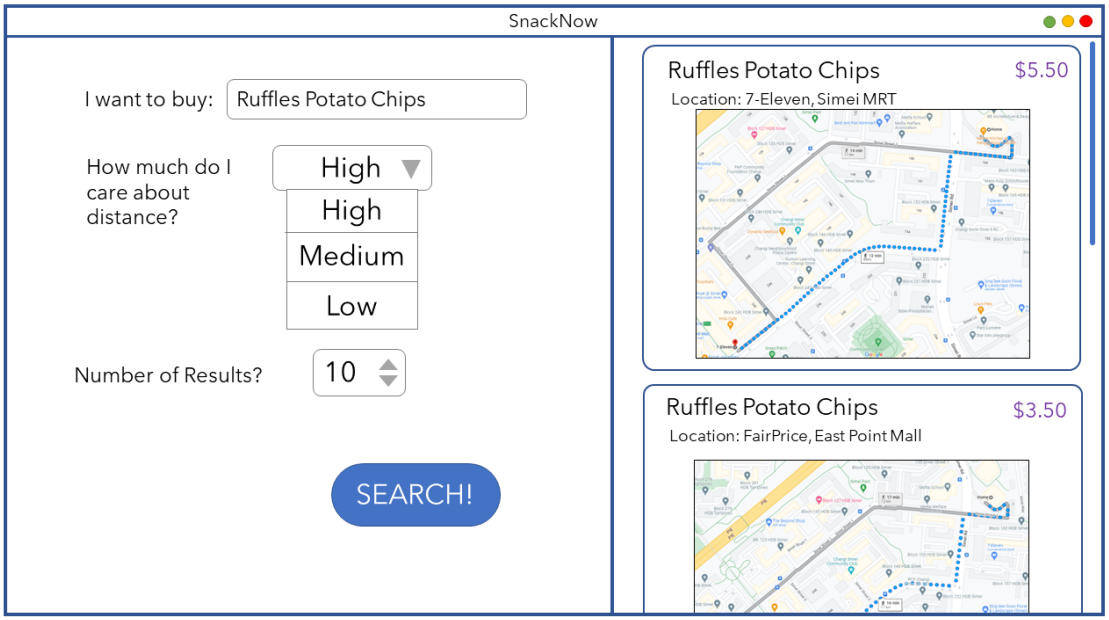

Figure 1: Wireframes initially used to design the User Interface

The app is a desktop app that can be run as an EXE or a JAR. The idea is to add a specific item that you wish to buy to the first textbox, then indicate how much you prioritize the distance, and also scroll up and down for the number of results. Upon pressing search, the app queries the areas based on the distance setting and produces results in the form of cards. This card contains the following:

- Name of product at the relevant store
- Address of the relevant store
- Image of the location of the store
- Price of the item

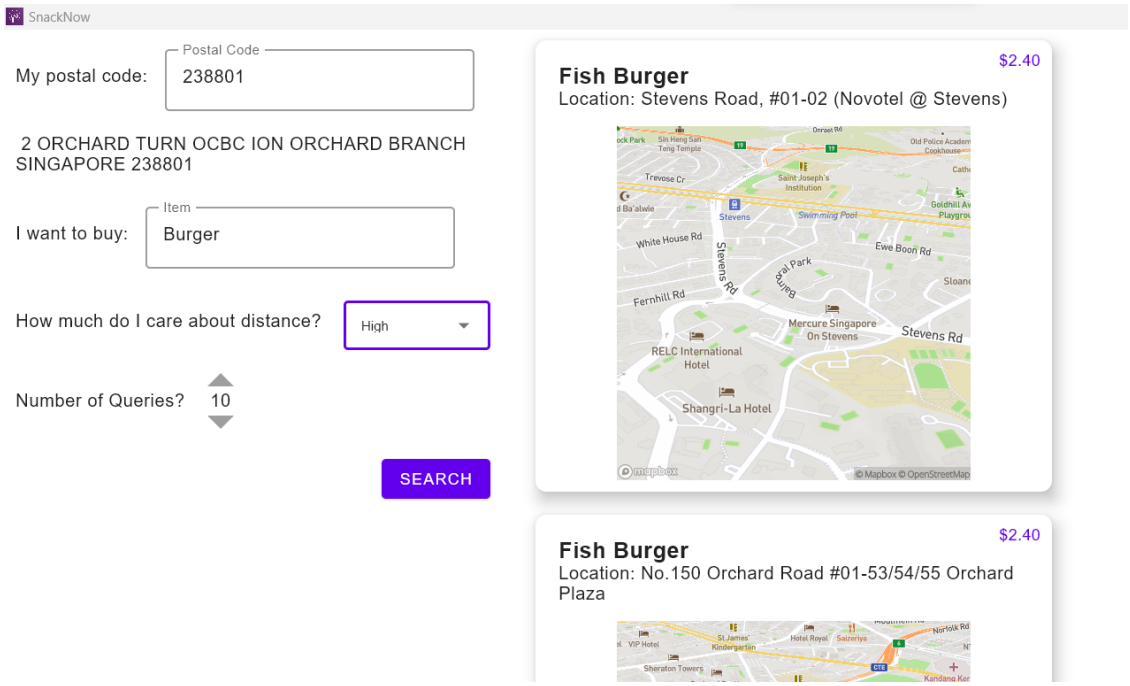

Figure 2: The final application - As you may notice, we query from ION Orchard due to the limitations of the database

In order to indicate your location, you have to put in your postal code (due to the current permission-related limitations of Jetpack Compose). The app then queries a database to identify your latitude and longitude for querying.

This app has some limitations though. For one, only a small area was actually queried as the locations of the 7-Elevens (which we used as our underlying database) were only limited to a specific region of Singapore, as shown below:

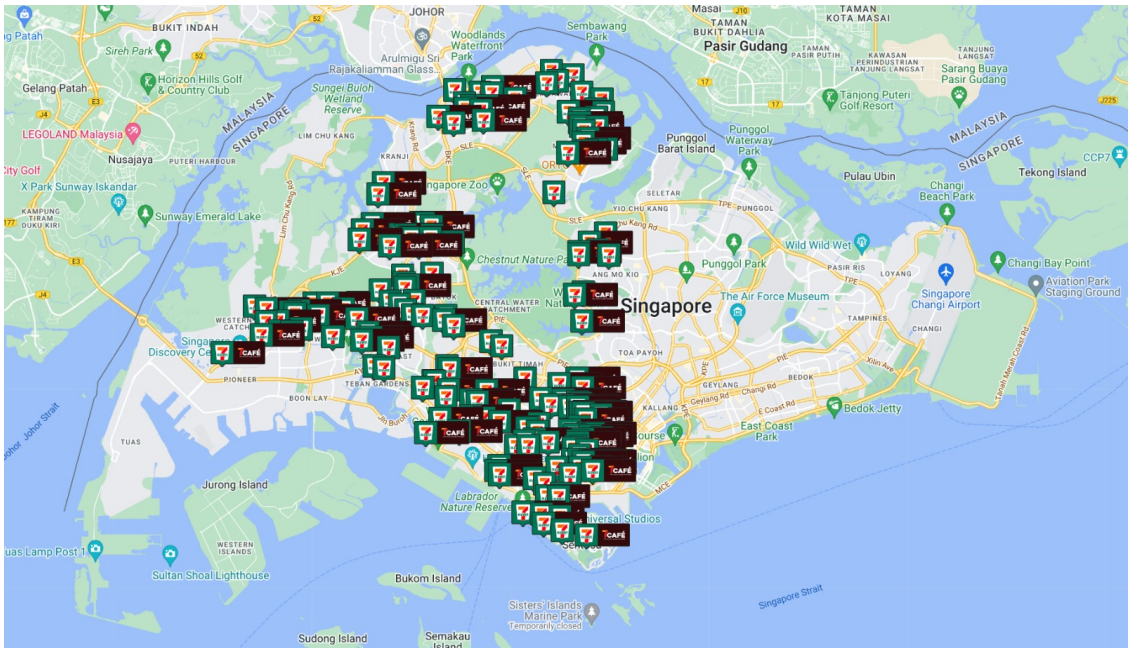

Figure 3: The search limitation on Google Maps lead to only a certain set of readings, mainly barring the eastside of Singapore

This leads to the following screen being rendered, which obviously isn't correct:

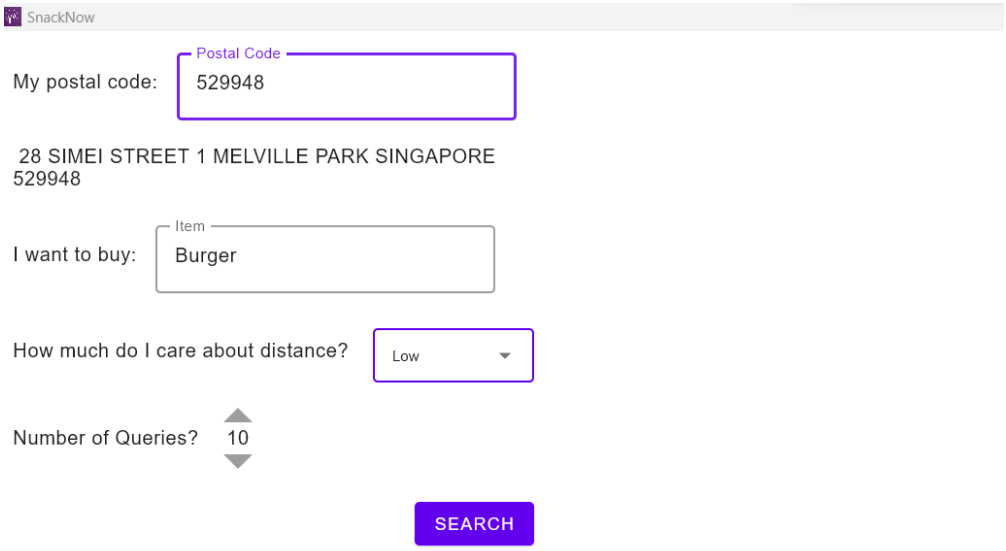

Figure 4: The limited set causes no results in Simei, an eastern estate in Singapore

On the other hand, another thing implemented to ensure user understand what is going on is a dummy list generated on start-up.

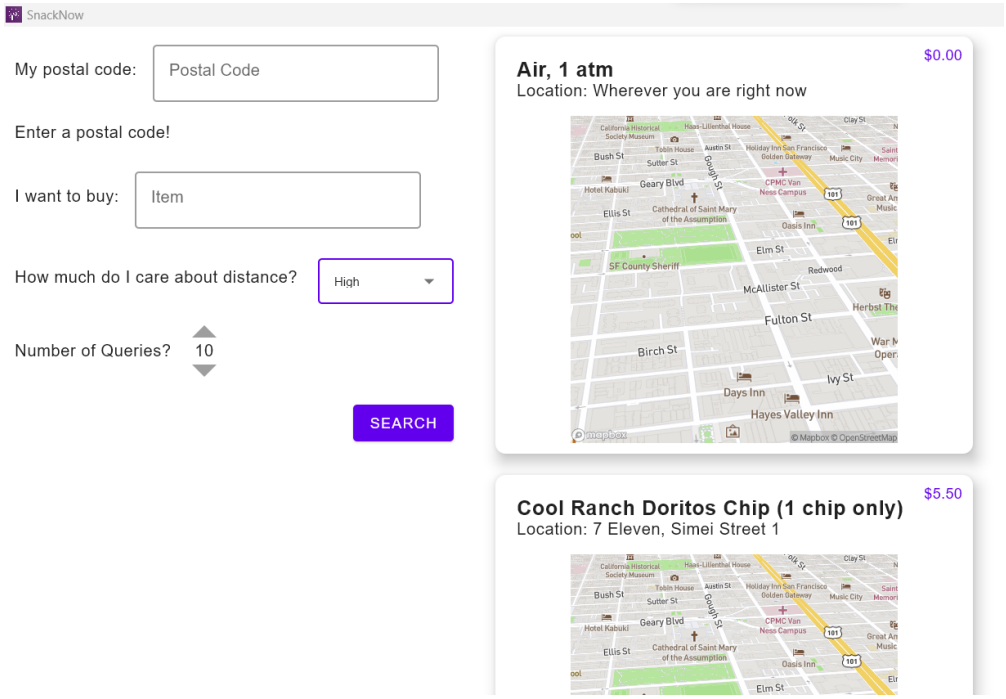

Figure 5: We swear, this is a *feature*, not a *bug*!

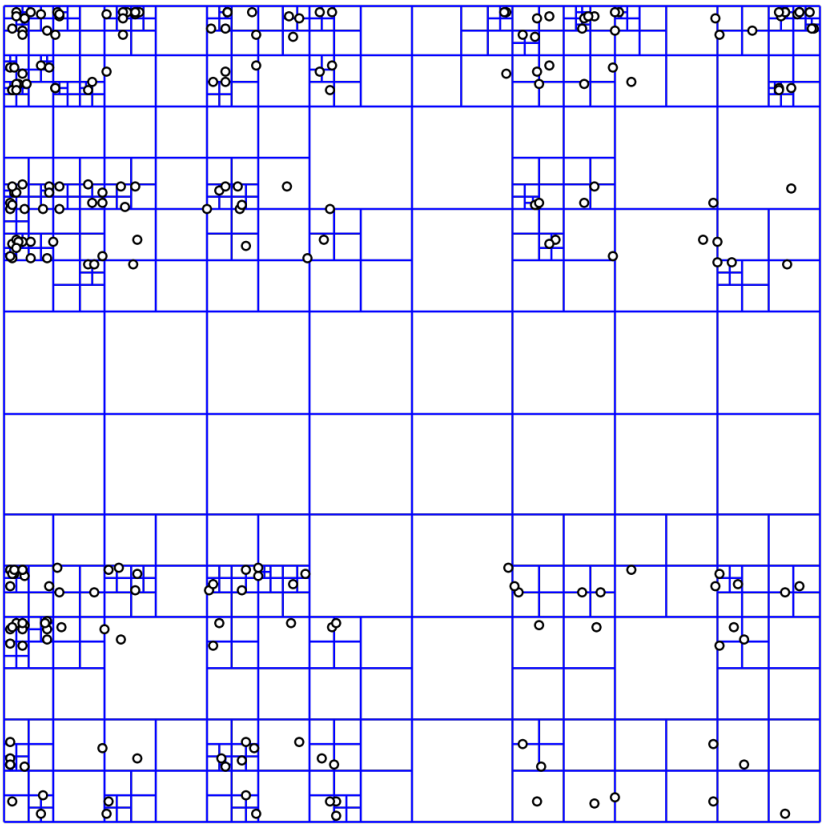

Figure 6: Quad-Tree visualization

3 Analysis and Explanation of Implemented Tree Methods

Provide a brief explanation, in your own words, and rigorous mathematical worst-case analysis derivation and time complexity of the key methods of your tree structure (retrieval, insertion, deletion, balance (if any))

3.1 Spatial Quad-Trees

Quad-Trees are commonly used in analysing the distribution of a collection in 2D space, as the size and precision of the data structure scales with the distribution of the spatial data we are working with. As we map out store branches over Singapore, we may find that the locations are not evenly distributed. If we try to split up the map of Singapore into a grid, and check if each grid is occupied by a store, then we may have a lot of overlap of stores in shopping districts, and wasted space in more residential areas.

3.1.1 Why Quad-Trees?

So, given the above issues, why did we choose Quad-Trees? Quad-Trees address this by adopting a recursive subdivision approach to grid-propagation [1]. That is, a quadrant is subdivided only during when there is a child node to be inserted into the quadrant. The depth of the tree is thus proportional to the log of the local density of points, which results in many speedups. Empty areas have very low depth, and dense areas have fewer "collisions".

This is particularly relevant in our case, because stores selling goods are very much not uniformly distributed throughout space. Many parts of Singapore, such as housing areas, will obviously have a relatively lower density

of food-sellers than a large shopping mall. There is a hierarchy to be found here, too. While quads obviously do not perfectly represent buildings, a resolution of 50 meters ensures that a building is likely to take up a subtree of depth 1-2 at the most. The implicit ordering follows upwards: A few of these subtrees form a series of blocks, these blocks form an area, and so on and so forth. While this is more a reflection of the choice of 50 meters than anything else, the arrangement is still noteworthy. It also, being a relatively small number, ensures that a quad will not have too many stores in it.

3.1.2 Algorithm

In terms of implementation, Quad-Trees are a generalisation of binary search trees to two dimensions. In order to facilitate this, internal nodes in the Quad-Trees have four children rather than two. Naturally however, one has to ask: how are we ordering objects in two dimensions? It is common knowledge that extending from the reals to the complex numbers gives up natural ordering, and the same applies here. Well, I'm so glad you asked, dear reader. Think about it: what do we actually use ordering for in a binary search tree? We use it to figure out if an element goes left or right. And here? Well, if we're using ordering to partition, why not just partition without ordering? And so we do, recursively subdividing quads into 4 smaller quads when necessary.

Insertion begins with a little preamble. If the object (the object in question is termed a `SuperStore`) to be inserted has a position that is outside of the bounding box of the Quad-Tree's root node, an exception is immediately thrown. Otherwise, the appropriate QuadNode child of the current node is chosen, created if it is currently null, and recursed on if it has an item (even if that item is null). When we reach a node that already has an item in it, we simply subdivide that node, and then reallocate both the current item and the previously stored item. Essentially, what this does is ensure that the depth of the tree is both **Dynamic** and **Local**. Naturally, you may wonder, "What if two `SuperStore`s have the same position"? Well, for that, we have a minimum Quad size. This minimum Quad size is about 0.00045 degrees (latitude and longitude), which translates to a resolution of roughly 50 meters.

Naturally, however, this raise another problem: How do we store multiple `SuperStore`s in a single QuadNode? After all, as per the specifications of this assignment, QuadNode must extend Node, and Node only holds a single object of type T. Must we store an `ArrayList` of `SuperStore`s? The answer is, kind of. You see, each `SuperStore` holds an `ArrayList` of items. And now you see why it's called a `SuperStore`; when two `SuperStore`s would take up the same minimum-sized node, they instead merge their catalogues together. For this to work nicely, items must store their location so that what store they actually belong to can be ascertained, and indeed they do. While this is not the most space-efficient approach, the space required to store this information is not huge in the grand scheme of things.

That more or less covers insertion. As there is no deletion operation (it doesn't make sense in this context), that leaves only window queries.

The process for doing window queries is rather intuitive. We start at the root, where we iterate over all children. If a given child quad has overlap with the query area, we recurse on it, passing down a shared `ArrayList` of items. If not, we skip the child. Whenever we reach a child that has an object inside of it, we append all of its items (that are inside of the query area) to the `ArrayList`. Repeat, until all quads overlapping the query have been traversed. This `ArrayList` is finally returned when the recursion terminates.

3.1.3 Time Complexity

3.1.3.1 Insertion: The worst-case time complexity of the insertion operation for Quad-Trees is O(n), where n is the size of the element to be inserted.

First, let us observe that no matter the structure of the tree, at most a finite number of recursive calls will be made to propagate down it. Due to the minimum size of the lowermost QuadNode, once we reach a recursion depth of around 10 (because Singapore is roughly 50km across), we will hit the minimum size. However, since this is a finite distance, this only contributes a constant term to the asymptotic time complexity of the algorithm. Indeed, 10 is low enough that we can call it constant without feeling bad about it. More relevant here is the merge operation.

Because we merge the the two catalogues together, we invariably invoke java's System.arrayCopy method. In researching the time complexity of this method, I have come to the conclusion that the only explanation here is "It's complicated". Given this, we will make the safe assumption that it is still O(n), where n is the number of elements to be moved from one array to another.
The time taken to run `System.arraycopy` is then roughly of the form:

$$T_i(n) = an + 10b + c$$

Where a is the constant factor hidden behind O(n), b is the time taken to run one recursive call, and c is whatever other overhead we have. Since this only needs to be run once per insertion, and does not matter on the current state of the tree, merely the element being inserted, then the asymptotic worst-case time complexity of insertion is:

$$O(T_i(n)) = O(an + 10b + c) = O(n)$$

3.1.3.2 Range Query: The worst-case time complexity of the range query operation for Quad-Trees is O(n).

At each recursion step, we must iterate over each of the four child nodes of the current node and add the number of elements n_k inside the node to the shared `ArrayList`. Since by greedy arguments, the worst case must when we query the entire search space, the worst-case time complexity of a range query is equivalent to the worst-case time complexity of traversing the entire tree and outputting all the items contained within it.

The former has a worst case time complexity of $T(n) = \max(4n, 4^{10}) + O(1)$. This is because at each insertion step, the tree will subdivide into four child nodes at most once, increasing the number of vertices and edges by 4. However, since we have artificially limited the depth of the tree by imposing real-world constraints, the time complexity of traversal is upper bounded by 4^{10}, that is the branching factor raised to the power of the max depth. This is due to the fact that the number of vertices can be expressed as a recurrence relation:

$$\begin{aligned} a_0 &= 1 \\ a_d &= 4a_{d-1} \\ &= 4(4a_{d-2}) \\ &\dots \\ &= 4^d a_0 \\ &= 4^d \end{aligned}$$

As for the time complexity of outputting all the items, although the number of elements contained in each node varies, since there are at most n items, the time complexity is only $O(n)$.

Thus, the asymptotic worst-case time complexity is $O(n + 4n + 4^{10})$, equivalent to $O(n)$

3.1.3.3 Why use Quad-Trees From our discussion of the worst-case analysis of the time complexities of Quad-Trees, we may notice that it is no better than a simple array. This highlights a shortfall of asymptotic worst-analysis, that is for certain use-cases where we are working with non-adversarial data with real-world constraints on the size, it is worth exploring algorithms focus on improving the time-complexity of the average-case. For instance, we may expect that stores would not all be concentrated in one quadrant, which would result in a severely unbalanced quad-tree, nor would the intended clientele be interested in finding stores beyond walking distance. In these cases, we can expect the quad tree to be roughly balanced, and be faster than an array by efficiently skipping nodes which are not interesting in, just as a binary search tree has $\log(n)$ search and insert on average.

4 Testing Strategy

Logs of the sample input and output can be found at https://github.com/ThePyProgrammer/SnackNow/tree/main/src/test/kotlin/model/algorithm/quadtree/testdata. Sample input files end with `'.csv'`, while sample output files end with `'.out'`.

4.1 `grid.csv`

To stress-test our algorithm, we generated a 20x20 grid of evenly spaced points and randomized queries. This dataset should resemble real world data in density and distribution.

This test cases tests queries that are very thin and do not enclose any points (8,10), queries which barely enclose points (2,4,6,7,9), queries which barely graze points (6 (notice the third point falls just outside the bounding box)), large queries (4,7), and queries close to the boundary (3). As expected, we may verify that the algorithm is robust in handling large instances, does not suffer from numerical errors in picking out stores close to a boundary, and works consistently. This should cover most typical cases.

```
OUTPUT:
1) Queried (0.00365, 0.00864) to (0.00762, 0.00762) and received 16 result(s):
... full output suppressed
2) Queried (0.00263, 0.00806) to (0.00806, 0.00548) and received 50 result(s):
... full output suppressed
3) Queried (0.00590, 0.00990) to (0.00990, 0.00753) and received 28 result(s):
... full output suppressed
4) Queried (0.00050, 0.00815) to (0.00815, 0.00094) and received 210 result(s):
... full output suppressed
5) Queried (0.00606, 0.00873) to (0.00652, 0.00652) and received 4 result(s):
[(0.00632, 0.00684), (0.00632, 0.00737), (0.00632, 0.00789), (0.00632, 0.00842)]
6) Queried (0.00886, 0.00344) to (0.00999, 0.00310) and received 2 result(s):
[(0.00895, 0.00316), (0.00947, 0.00316)]
7) Queried (0.00312, 0.00728) to (0.00728, 0.00106) and received 88 result(s):
... full output suppressed
8) Queried (0.00898, 0.00893) to (0.00898, 0.00033) and received 0 result(s):
[]
9) Queried (0.00074, 0.00723) to (0.00723, 0.00624) and received 24 result(s):
... full output suppressed
10) Queried (0.00045, 0.00891) to (0.00891, 0.00888) and received 0 result(s):
[]
```

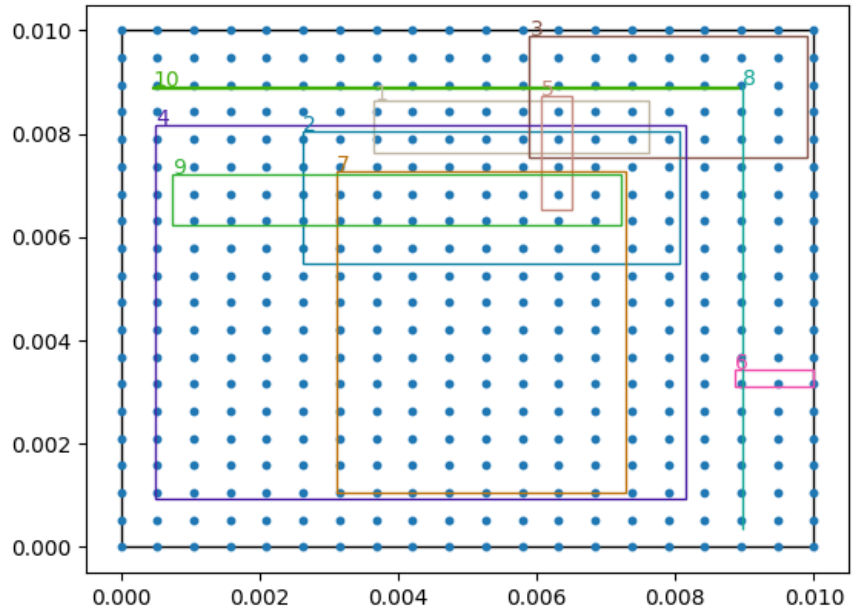

Figure 7: Visualization of the sample input of `grid.csv`

4.2 `edges.csv`

The program should throw an error when trying to insert points outside the bounding box to alert the user of malformed data.

```
OUTPUT:
Exception in thread "main" java.lang.IndexOutOfBoundsException:
Point outside of bounding box of QuadTree!
```

4.3 `single.csv`

We verify that the QuadTree works even when it does not need to subdivide.

```
OUTPUT:
1) Queried (-0.01000, 0.01000) to (0.01000, -0.01000) and received 1 result(s):
[(0.00000, 0.00000)]
```

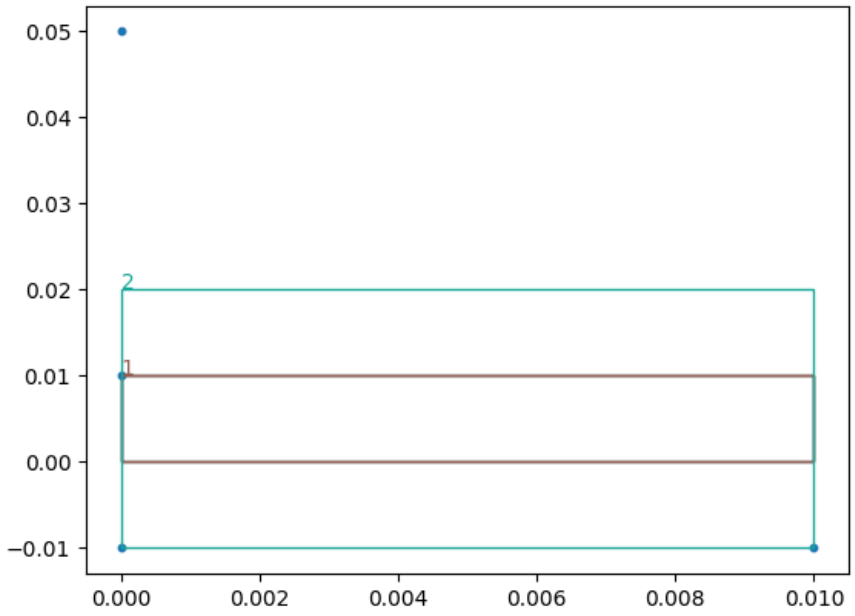

Figure 8: `edges.csv` - Bounding box overlaps exactly with the brown query 1 window.

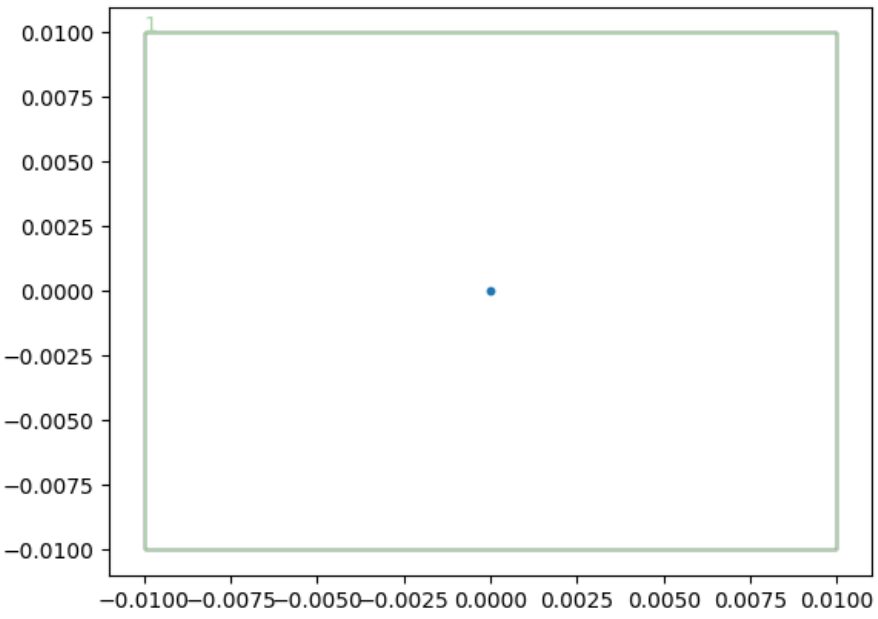

Figure 9: Sample data visualization from `single.csv`

5 GitHub Information

You can find our repository at `https://github.com/ThePyProgrammer/SnackNow`.

5.1 User Directory

GitHub Username	Real Name
`delargement`	Kannan Vishal
`ThePyProgrammer`	Prannaya Gupta
`h1810126`	Quek Yu Pin
`VikramRamanathan`	Vikram Ramanathan

5.2 Repo Report

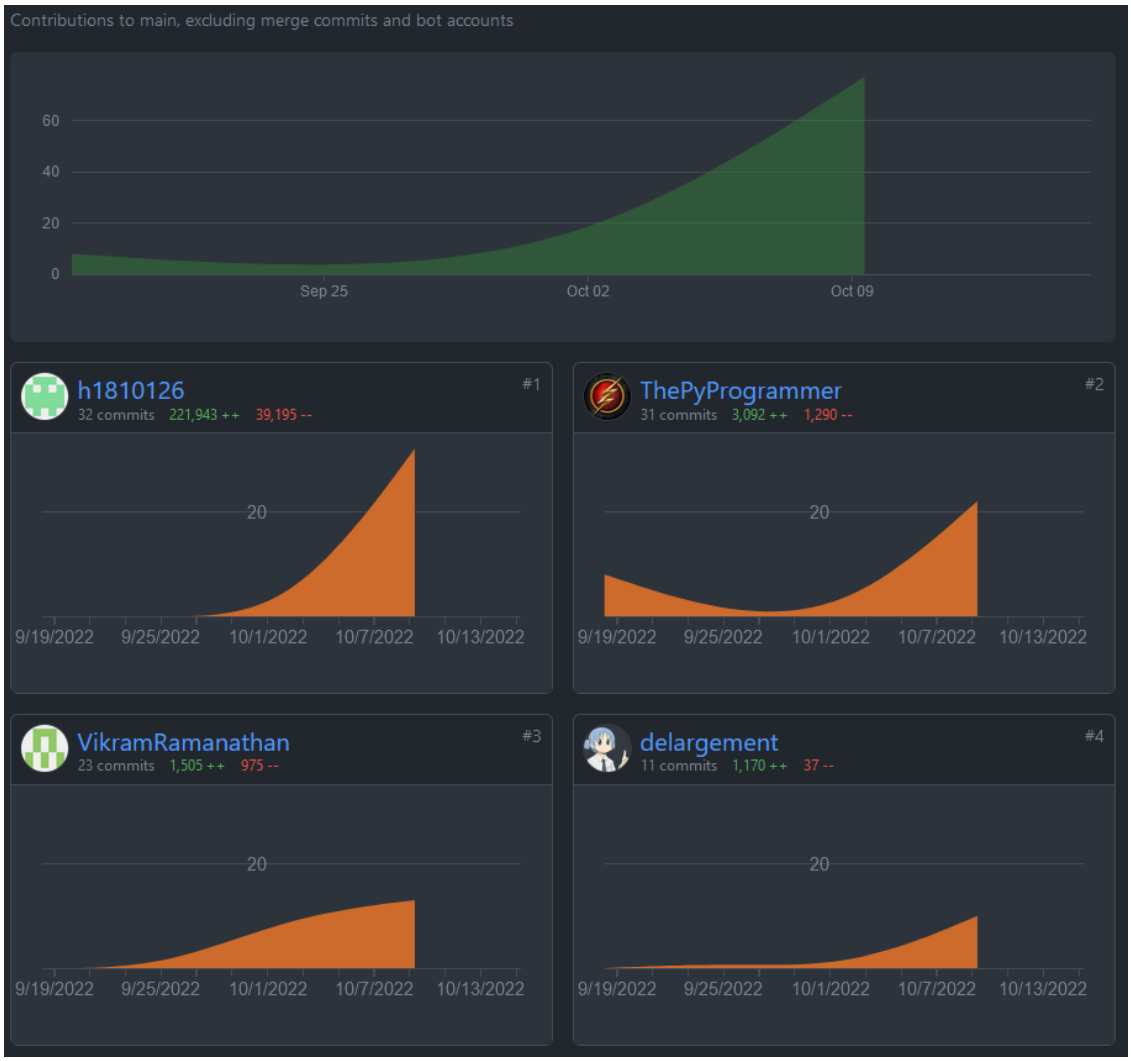

Figure 10: Our Commit Graph

6 Reflection

6.1 Vishal

By helping out a here and there in each component of the project, and reviewing our project while writing the report, I have gained a better understanding of the software development lifecycle, from the brainstorming stage, UI prototyping stage, data collection and documentation. By abstracting each component of the project, such as the model (RTree and Quadtree implementation) and GUI, and distributing the workload, we were able to be more productive, and it was enjoyable to see each of the pieces to connect together to give a minimum viable product.

6.2 Prannaya

In my humble opinion, with the help of this project, I have been able to understand the structures and methods associated with trees a lot better, compared to when I simply attempted Task I on its own. Getting to create a user interface to adapt trees showed me how such data structures can be used adaptively for different problems which I never thought of, and has broadened my perspective on using Data Structures like Queues and Stacks beyond IO-like problems. In general, it's been fun to see the apps come together, and I am happy to see that it works (somewhat).

6.3 Yu Pin

Doing this project made me understand how trees that don't exist in real life are used in real life, and made me have the experience of creating an application that is actually useful. While doing the data collection, I found out how hard it is to get some Singapore-specific data, especially the fact that there was no way to get the locations of all the convenience stores in Singapore other than to visit all of them. There wasn't much data about the items too, so I had to generate some fake items, and the process of doing that was... enjoyable. The project is coded in both Java and Kotlin, and I realised how easy it is to use both languages in one project. While working on the application after the user interface was made, I used the data I had collected and the data structure that Vikram implemented to make the application work, and it was interesting to see all the data and the quadtree come together to make it useful and efficient.

6.4 Vikram

As the one who was responsible for the actual implementation of the Quad-Tree, I have learned a lot about the virtues and horrors of abstraction. The upside: It makes the code a lot neater to write, a lot easier to read, and is generally just elegant. The downside: It makes debugging code paradoxically more difficult, because when things are more elegant, and they don't work, trying to figure out what part of the elegance is breaking is very difficult. One particularly memorable aspect of debugging process (for me), was when I spent two days, on and off, trying to track down a bug present in my Queries. In the end, it turned out to not be in my queries, but in my insertions; my code was correct, I could swear it, and I was right. It was someone else's code which had broken, and abstraction had hidden it from my attention.

Still, I learned a lot in the overall process. Not so much about the technical aspects of the programming, but in the process of researching which trees to use, I did hit upon somewhat of a realisation regarding data structure development. The distinction between implicit properties of data and explicit choices in organisation is not one that I had thought about much before this assignment, but now I understand more.

7 Work Distribution Matrix

	Vishal	Prannaya	Yu Pin	Vikram
Ideation				✓
R-Tree Implementation		✓		
Quad-Tree Implementation				✓
Data Collection			✓	
User Interface	✓	✓	✓	
Report	✓	✓		✓

*The R-Tree was cut in the end, but still worked on by Prannaya

8 References

[1] Ravi Kanth V Kothuri, Siva Ravada, and Daniel Abugov. "Quadtree and R-Tree Indexes in Oracle Spatial: A Comparison Using GIS Data". In: *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*. SIGMOD '02. Madison, Wisconsin: Association for Computing Machinery, 2002, pp. 546–557. ISBN: 1581134975. DOI: 10.1145/564691.564755. URL: https://doi.org/10.1145/564691.564755.