A Multipurpose E-commerce System Using Haversine formula and A* Algorithm

Emad Sa'adeh[1*], Bara Barham[2], Osama Odeh[3]

[1,2,3] An-Najah National University, Computerized Information System Department
Email: [1*]esaadeh@najah.edu, [2]baraheyl@gmail.com

ABSTRACT

In this paper we provide the design and implementation of a web-based real-time interactive e-commerce system. Besides the ordinary features of currently working e-commerce systems in the market, we fully implement three advanced features in our new system. The first feature provides customers with the ability to access products of all stores within a specific area surrounding them, implemented by using a customized Haversine Formula. The second feature fetches the shortest path between the store's location and the customer's location, implemented by using a customized A* algorithm. The last feature implemented by an enhanced observer design pattern gives customers the ability to watch product price modifications according to their individual criteria. All the customized algorithms used in the system are fully evaluated and tested. Different technologies are used to implement our system like ReactJS, NodeJS, Socket.Io , Firebase , and Google API. The complete code is available on the GitHub code hosting website.

Keywords

shortest path, haversine formula, observer pattern

Article Received: 10 August 2020, Revised: 25 October 2020, Accepted: 18 November 2020

Introduction

The world of e-commerce is interesting, as it is constantly growing to meet customer needs. It has many benefits such as locating the product quicker, eliminate travel time and cost, provide comparison shopping, and so on. Although these benefits are significant for online shopping consumers, sometimes people need something that works faster to access products in their area that is close to their location. Our system not designed for a specific store or a specific company, it is a global system that can include open-numbers of stores. This gives an additional feature to the customer to compare between the different stores and choose the suitable one.

We design our system to access all the registered stores in the customer's area within a specified distance. Using google maps API and a customized A* algorithm, the system can fetch the shortest path between customer and store which makes it easier for the delivery service to complete their job easily. The paper will show how the Haversine formula is used and customized to achieve such functionality.

Our system gathers all the stores available in a customer's area using google maps API, so customers can do shopping while they are at their home with no need to go to crowded places which is beneficial for conditions like COVID-19. The system can fetch the customer location automatically via the google maps API, send it to the chosen store, and then to the delivery company as soon as the store approves the customer order.

Besides the above features, our system gives the ability for customers to watch the products they want. When the price of a product goes down to a certain value specified previously by a customer, then that customer will be notified immediately. Note that–up to our knowledge–the watching features implemented in the current e-commerce systems do not allow customers to specify any criteria on watching. This means that when any changes to the price done on that product, all the customers on the watching list will be notified. The major drawback of this approach is that the customer may interest in the product when it is below a certain value and not interested in all other modifications done above that value. Also, sending notifications on any price modification may disturb customers. Finally, this for sure will increase the overhead on the system.

System architecture

The architecture of the main components of our system is displayed in Figure 1. It is a client/server architecture comprising the client-side and the server-side. The server side

comprises the customer, merchant, and admin modules. The client-side comprises a web-based user interface that can support three kinds of actors: customer, merchant, and admin actors.

-

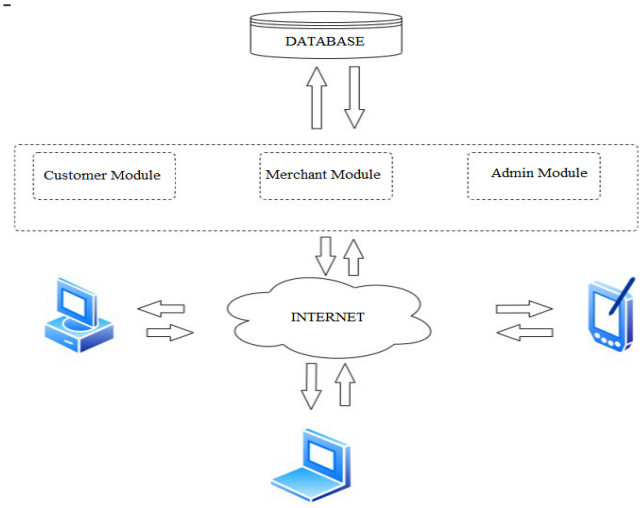

Figure 1. System architecture

A. The Client-Side

The client-side provides an entry point for customers and merchants through a web browser. The system provides customers with many functions that can be accessed through the client-side (web browsers). For example, the customer can search, book, buy, rate, and watch products inside all the stores registered in the system. A powerful search engine with many search options is implemented inside the system. One of the search criteria is to search for stores that are closest to the customer's location. So the system may be used as a navigation tool if the customer 'for instance' suggested eating in the nearest restaurant, for example.

Merchants also have many functions inside the system. After registering and having accounts on the system. They can add, review, search for their products, change product prices, printing many kinds of reports. Merchants can also check the orders they received and checked the path from the store to the customers.

B. The Server-Side

The server side acts as a mediator between the actors and the system and between the actors themselves. To accomplish these tasks, the server uses several modules as shown in Figure 1.

Customer Module

The major task of this module is to manage the interaction between the system and the customers. The module has access to the database of the system. Provides customers with the ability to shop easily after automatically extracting their locations.

Merchant Module

The major task of this module is to manage the interaction between the system and the merchants. The module has access to the database of the system. Provides merchants with the ability to manage their products easily and efficiently.

Admin Module

The major task of this module is to manage all the merchants' accounts inside the system. The admin actor can activate, suspend, resume merchants' subscriptions to the system.

Technologies used

In this section, we describe the key technologies used in our implementation of the system. These technologies used to incorporate google maps API and implement different functionalities provided in the application thus making the application very reliable.

A. React (also known as React.js or ReactJS) [1]

Is an open-source JavaScript framework that is used by Facebook. It is chosen to be used in our system because:

- Well developed with a vibrant ecosystem of developer tools with many pre-made front-end components like charts, tables, and etc.
- React does not offer any concept of a built-in container for dependency. For that, we use Browserify, Require JS, ECMAScript 6 modules via Babel, ReactJS-di to inject dependencies automatically. This increases the performance of our system.
- Efficiently used with NodeJS: ReactJS is the best option to be used with google maps API and NodeJS connection.

B. NodeJS [2]

Is a server-side non-blocking event-driven application that uses Googles's V8 virtual machine to run JavaScript code. It supports push technology over web-socket, which allows the development of real-time highly interactive web

applications. One advantage of Node.js is its scalability. It can handle many simultaneous connections with high throughput. In our system, we used Express.js [3] which is a web application framework for Node.js.

C. Firebase

Is a storage facility used to efficiently store and serve user-generated content such as images. We used it in our system to store all the images uploaded by the different actors like bank receipts or product images uploaded by merchants.

D. Socket.Io

Is a JavaScript client library that provides reliability for handling proxies and load balancers. We used it in our system to chat and sending real-time notifications when the product price goes down a value specified by the customer.

E. Google Maps API

Allow developers to integrate Google Maps into their systems using their own data points. In our application, we used it to represent stores and customers' locations on the map. And exploiting graph representations of the different points in order to be used by our customized A* algorithm as explain thoroughly later in the paper.

F. MongoDB [4]

Is a database management system designed for the Internet and web-based applications. It is a document-based No-SQL database and used in our system because:

- Data is stored as JSON style documents index on any attributes.
- Fast In-place updates.
- Very easy to incorporate with NodeJS thus we achieved the "MERN" usage that stands for (MongoDB+ExpressJS+ReactJS+NodeJS).
- Support auto-sharing feature [5] which gives the ability to store data records across multiple machines and meet the demands of data growth. As the size of the data increases, a single machine may not be sufficient to store the data nor provide an acceptable read or write throughput.

Our approach: Enhancements and Customizing

In this section, we describe the three features implemented in our system (finding all stores nearby the customer, finding the shortest path between customer and store, and product watching functionality). It discusses the different algorithms used to implement each one of these features.

A. Shortest path between customer and store

Fetching the shortest path between the customer and the chosen store is the first feature implemented in our system. We use the google maps API to get the graph of the specified coordinates of the customer and the store. Then, a customized A* algorithm is used to fetch the shortest path between them by using Euclidean Distance as our heuristic function. Finally, the resulted points are sent back to google API in order to display the path on the map graphically as shown in figure 2.

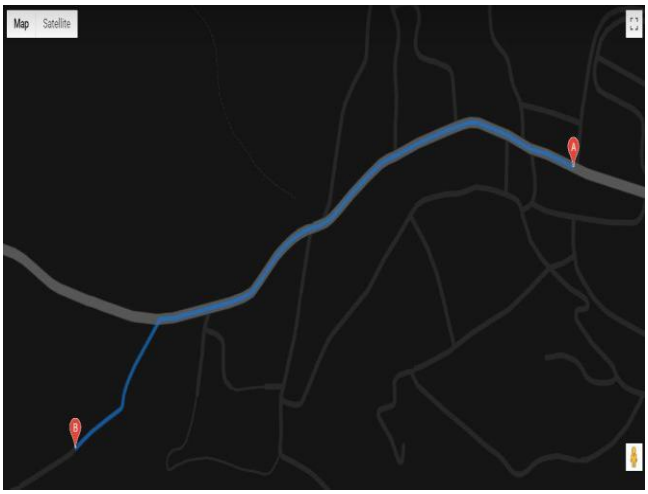

Figure 2. Shortest Path between customer and store

A Algorithm vs Dijkstra Algorithm*

This section explains the reason for using A* algorithm instead of Dijkstra's algorithm to find the shortest path. Although the Dijkstra algorithm finds the optimal shortest path between two points over the A* algorithm, it has a time complexity of O(E+VlogV) [9]. We find that the number of visited nodes by the Dijkstra algorithm is very high when compared to the nodes visited by the A* algorithm which considered being time-consuming. Note that the time factor is very important in finding the shortest path in our system, especially if the customer is walking in

the street and quickly needs to fetch the path to the required store. Figure 3 shows the difference between A* and Dijkstra algorithms. The figure shows that the Dijkstra algorithm got the optimal path. At the same time, it shows that the path got by the A* algorithm is close to being optimal. In contrast, A* visited only 201 nodes while Dijkstra visited 2186 nodes, which is very high and time-consuming compared to the A* algorithm. As future work, we plan to increase the optimality of the A* algorithm more by implementing a better Heuristic function in order to calculate the optimal path more efficiently.

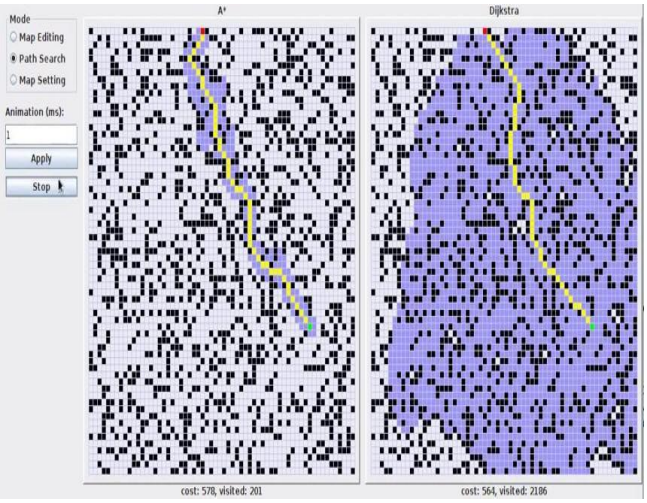

Figure 3. A* vs Dijkstra

A Heuristic Function*

Many options for implementing the A* heuristic function are evaluated and tested. Such as Manhattan Distance, Diagonal Distance, and Euclidean Distance. Euclidean Distance in our approach is chosen as a heuristic function because it gives the possibility to move in any direction, unlike the Manhattan Distance which just moves in four different directions (up, down, left, and rightwards). We use the following equation to compute the Euclidean distance, where p1 and q1 stand for the latitude of customer and store, respectively. While p2 and q1 stand for customer's and store's longitude.

$$d(\mathbf{p},\mathbf{q}) = \sqrt{(p_1 - q_1)^2 + (p_2 - q_2)^2 + \cdots + (p_i - q_i)^2 + \cdots + (p_n - q_n)^2}.$$

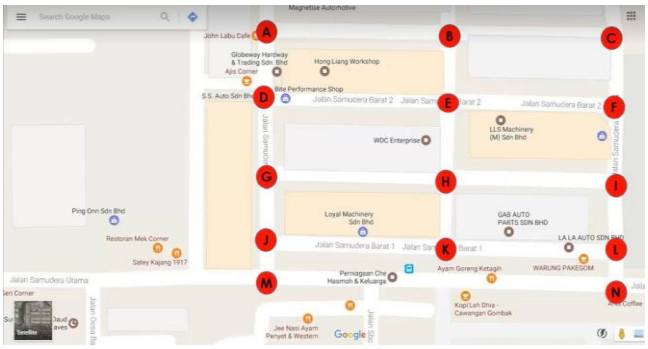

Figure 4. Google map represented as a graph

Figure 4 shows how a google map is represented as a graph. Each intersection between two or more roads represents a vertex. While the road connects two vertices represents an edge of the graph.

B. All stores nearby the customer

The second feature in our system is giving the customer the ability to find all stores within a specific area around him. The system automatically fetches the customer's coordinate (latitude and longitude) using Geolocation API in vanilla JS function called "getCurrentPosition()"[6]. Customer coordinate, required distance in KM, and coordinates list of stores found in the customer region are sent to our customized Haversine formula[7]. The Haversine formula calculates the distance between two points on spherical objects and works as the followings: The formula takes the mean radius of the earth (r), which is estimated to be 6371. Also it takes the coordinates (latitude and longitude) of the customer and the store (φ, λ). Then, it applies the following formula on them:

$$d = 2r\arcsin\left(\sqrt{\operatorname{hav}(\varphi_2 - \varphi_1) + \cos(\varphi_1)\cos(\varphi_2)\operatorname{hav}(\lambda_2 - \lambda_1)}\right)$$
$$= 2r\arcsin\left(\sqrt{\sin^2\left(\frac{\varphi_2 - \varphi_1}{2}\right) + \cos(\varphi_1)\cos(\varphi_2)\sin^2\left(\frac{\lambda_2 - \lambda_1}{2}\right)}\right)$$

C. Product watching functionality

The third advanced features implemented in our system is giving the ability for customers to watch the different products they want. When the price of a product goes down to a certain value specified previously by a customer,

then that customer will be notified immediately. A customized observer pattern is used to implement the feature. In order for the notification process to be done in a real-time fashion, a Socket.Io[8] library is used. Once a merchant changes the price of a product, the system on the server-side checks the watching list of that

product and sends notifications for any customer whose target price -saved in the watching list- is equal or less than the new price.

Future improvements

Some possible future improvements that should make our system perform in a more optimized way can be summarized as the followings:

- Improve the heuristic function to have better results in locating the optimal path between customers and stores. Regarding correctness and time needed for calculations.
- Implement a more accurate formula other than Haversine formula to get better results when determining the closest stores nearby the customer.

Conclusion

The discussion in this paper focuses on the main advanced features implemented in our e-commerce system: Finding all stores nearby the customer, finding the shortest path between a customer and the store, and product watching functionality. The paper presents the various improvements done on the different algorithms used to implement each one of these features.

Such improvements to the e-commerce system make it a multipurpose system. While the system can be used as an ordinary e-commerce tool. It can be used as a navigational tool for finding stores located around while travelling. And also as an alert tool that sends notifications for customers on responding to price modifications.

Using such a system will be very beneficial, especially in lock-downs circumstances because for conditions like COVID-19 which give the possibility for stores and markets to continue working while avoiding crowded places.

References

[1] React - A JavaScript library for building user interfaces https://reactjs.org/.
[2] https://openjsf.org/.
[3] https://expressjs.com/.
[4] http://www.tutorialspoint.com/mongodb/.
[5] Sanobar Khan, P. M. \SQL Suport over MongoDB using Metadata". in International Journal of Scienti c and Research Publications Vol 3 (2013).
[6] https: //developer.mozilla.org/en-US/docs /Web/API/Geolocation/getCurrentPosition.
[7] Prof. Nitin R.Chopde1, M. M. K. N. "Landmark Based Shortest Path Detection by Using A* and Haversine Formula"
[8] https://socket.io/.
[9] https://everythingcomputerscience.com/algorithms/Dijkstras_Algorithm. html.
[10] https://neo4j.com/docs/graph-algorithms/current/labs-algorithms/euclidean/.