5800
ALGORITHMS

PROJECT REPORT

TITLE:
“Optimizing Data Retrieval for Improved User Experience in an E-commerce Website”

BY
KARTHIK CHINTAMANI DILEEP
KARTHIK JAYARAM BHARADWAJ
SHRADA CHELLASAMI
SANKARA NARAYANAN RAJAGOPAL

Youtube video link: https://www.youtube.com/watch?v=4Jzc-dcF5_o

INTRODUCTION:

The topic is focused on the measures taken by a large e-commerce company, which has servers situated in different regions globally, to ensure that its website is accessible to users worldwide. The company has observed that certain users are encountering issues such as delayed page load times and prolonged wait times while attempting to finalize transactions on the website. To enhance user experience by optimizing data retrieval, the company decides to adopt a strategy that involves determining the most direct path among its servers. It utilizes network monitoring tools to gather information on the network's topology, which includes the servers' location and connectivity.

Karthik Chintamani Dileep - To enhance website performance and customer experience, e-commerce businesses must optimize data retrieval. Companies may decrease page load times and wait times, resulting in higher customer satisfaction and repeat business, by putting in place strategies like load balancing, caching, and compression. In addition to the business benefits, optimizing data retrieval also has technical advantages, including improved network performance and reduced bandwidth usage, leading to more efficient use of resources and cost savings. Overall, in today's digital age, the ability to optimize data retrieval is crucial for businesses to stay competitive and succeed in the online marketplace. The business chooses to put into practice a method for determining the shortest path between servers in order to enhance data retrieval and enhance user experience. They gather data on the network topology, including the location and connection of every server in the network, using network monitoring tools. They can also use algorithms like Dijkstra's algorithm or the Bellman-Ford algorithm. This enables the business to route data requests more effectively, cutting down on the amount of time data must travel between servers and speeding up user page loads.

Karthik Jayaram Bharadwaj - Suppose we consider a hypothetical scenario in which a large e-commerce company has servers located in different regions globally, and the company wants to ensure that its website is accessible to users worldwide. The company has noticed that some users are facing issues such as slow page load times and long transaction wait times while completing purchases on the website. In response to this problem, the company decides to adopt a strategy that involves identifying the most direct path among its servers to enhance user experience by optimizing data retrieval. To determine the most direct path, the e-commerce company utilizes network monitoring tools to gather information about the network's topology, including the location and connectivity of its servers (in simple words, use a lookup table to determine the best possible way to fetch the required data). By analyzing this data, the company can identify the shortest path between the user's location and the nearest server, ensuring that the user can access the website and retrieve data quickly. Implementing this strategy can help the e-commerce company to reduce page load times and improve the user experience for its customers. Suitable cache mechanism and distributed content delivery networks can be used to optimize the page load times for the clients, thereby boosting the overall performance of their page.

Shrada Chellasami - In a market for e-commerce that is extremely competitive, website performance and user experience are essential for luring and keeping clients. Long wait times and slow page loads can aggravate users, who may then leave the website and lose money and sales. E-commerce businesses may enhance website speed and customer experience by improving data retrieval using techniques like load balancing, caching, compression, and determining the shortest path between servers. Increased client happiness, loyalty, and repeat business can result from this, which eventually spurs revenue development. Overall, the ability to optimize data retrieval is becoming increasingly important in today's digital age, as more and more businesses rely on online platforms to connect with customers and drive growth.

Sankara Narayanan Rajagopal - As e-commerce continues to grow in popularity, online businesses face the challenge of managing and retrieving large amounts of data, leading to increased complexity on their websites. This complexity can cause slow page load times and lengthy transaction processing, ultimately resulting in a poor user experience. This negative experience can result in reduced sales, decreased customer loyalty, and a damaged brand image. Therefore, I think it is essential for large e-commerce companies to optimize their data retrieval strategies to improve the user experience and avoid negative consequences.

DEFINED QUESTION

E-commerce has become an increasingly popular way for consumers to purchase goods and services, and as a result, e-commerce websites have become more complex, with more data to manage and retrieve. This has led to challenges in ensuring a smooth user experience, particularly for users who experience slow page load times and long wait times when completing transactions. A poor user experience can result in lost sales, reduced customer loyalty, and negative brand perception. As such, it's crucial for large e-commerce companies to optimize their data retrieval methods and improve the user experience on their websites. To achieve this goal, e-commerce companies must consider a range of strategies to reduce page load times and wait times for transactions. These strategies can include implementing content delivery networks (CDNs), caching, image and media optimization, prioritizing above-the-fold content, lazy loading, and data compression. By leveraging these strategies and continuously monitoring and optimizing performance, e-commerce companies can ensure a faster, more seamless user experience, which can lead to increased customer satisfaction, loyalty, and sales. Furthermore, as online shopping continues to grow in popularity and competition among e-commerce companies intensifies, optimizing data retrieval and improving the user experience has become more critical than ever. Companies that fail to keep up with user demands for fast and reliable websites risk losing customers to competitors who provide a smoother user experience. Therefore, it's important for large e-commerce companies to invest in ongoing optimization efforts to stay ahead of the curve and meet the evolving needs of their customers.

What strategies can a large e-commerce company implement to optimize data retrieval and improve the user experience on its website, especially for users experiencing slow page load times and long wait times when completing transactions?

ANALYSIS:

ALGORITHMS:

BFS is used to find the nearest server.

BFS stands for Breadth-First Search, which is a graph traversal algorithm used to explore and visit all the nodes of a graph or tree in a breadth-first manner. It starts at a designated starting node and explores all its neighboring nodes before moving on to the next level of nodes.

In other words, BFS explores all the nodes at the same depth or level before moving on to explore the nodes at the next level. This process continues until all the nodes in the graph or tree have been explored.

We use BFS for a variety of applications, including finding the shortest path between two nodes in an unweighted graph, checking if a graph is connected, finding all connected components in an undirected graph, and constructing a level-by-level tree traversal.

BFS is a very efficient algorithm and has a time complexity of O(V+E), where V is the number of vertices and E is the number of edges in the graph. It is widely used in various fields, including computer science, data science, and artificial intelligence, for various applications such as web crawlers, social network analysis, and game AI.

FLOYD WARSHALL is used to get the get the distance among nodes.

Floyd-Warshall is an algorithm for finding the shortest path between all pairs of vertices in a weighted graph. It uses dynamic programming to compute the shortest distances between all pairs of vertices in the graph. The algorithm works by considering all possible intermediate vertices between two vertices and updating the shortest path distances accordingly.

We use the Floyd-Warshall algorithm in situations where we need to find the shortest path between all pairs of vertices in a graph. This can be useful in various applications such as network routing, traffic flow analysis, and scheduling problems.

The main advantage of the Floyd-Warshall algorithm is that it can handle graphs with negative edge weights, unlike other shortest path algorithms like Dijkstra's algorithm. However, the time complexity of the Floyd-Warshall algorithm is O(V^3), where V is the number of vertices in the graph. Therefore, it is not suitable for large graphs with many vertices.

Overall, the Floyd-Warshall algorithm is a powerful tool for finding the shortest paths between all pairs of vertices in a graph, and its ability to handle negative edge weights makes it a useful algorithm in various applications.

DIJKSTRAS – for finding shortest path.

Dijkstra's algorithm is a shortest-path algorithm that is used to find the shortest path between a starting node and all other nodes in a weighted graph. The algorithm works by iteratively selecting the node with the smallest distance from the starting node and updating the distances of its neighboring nodes.

We use Dijkstra's algorithm in situations where we need to find the shortest path between a starting node and all other nodes in a graph, such as in network routing, GPS navigation, and resource allocation problems.

The main advantage of Dijkstra's algorithm is that it can handle graphs with non-negative edge weights, which makes it a popular choice for solving various real-world problems. Additionally, it is relatively easy to understand and implement compared to other shortest-path algorithms.

However, Dijkstra's algorithm may not work well on graphs with negative edge weights, as it can get stuck in a loop of updating the distances between nodes. In such cases, we may need to use other algorithms such as Bellman-Ford or Floyd-Warshall to find the shortest path.

Overall, Dijkstra's algorithm is a powerful and widely-used algorithm for finding the shortest path between a starting node and all other nodes in a graph with non-negative edge weights. Its simplicity and efficiency make it a popular choice for a variety of applications.

SERVER:

Each of the server constitutes of:

1. Storage which holds the data required
2. Small Cache which holds data **frequently** used.
3. Lookup Table which indicates where to find the required data.

1. STORAGE:

The storage of the server holds all data that is permanent to it.

2. CACHE:

We make use of an LFU (Least Frequently Used) cache within each server, which is a type of cache that removes the least frequently used item when it reaches its capacity limit.

The LFUCache class has a constructor that takes a capacity parameter and initializes the cache dictionary, as well as two dummy nodes for the left and right ends of the cache.

The Node class represents a node in a doubly linked list. Each node has a key, a value, and a count of the number of times it has been accessed (occurrences). It also has a reference to its next and previous nodes in the list.

The insert method is used to insert a new node after a given previous node. It takes two parameters: the previous node and the new node.

The get method takes a key as input and returns the corresponding value if it exists in the cache. If the key is present in the cache, it increments the occurrence count of the corresponding node and moves it to the left of the next node with a higher count (or to the left end if there is no such node). This helps to ensure that the least frequently used nodes are located on the right side of the cache.

The moveLeft method takes a node as input and moves it to the left of the next node with a higher occurrence count (or to the left end if there is no such node). It does this by traversing the list from the current node's left neighbor to the left end of the cache, looking for the first node with a higher count. It then removes the node from its current position and inserts it after the found node.

The moveRight method is similar to moveLeft, but it moves a node to the right of the previous node with a lower occurrence count (or to the right end if there is no such node). It is currently not used in this implementation.

The put method is used to insert a new key-value pair into the cache. If the key is already present in the cache, it updates the value and occurrence count of the corresponding node, and then moves it to the left of the next node with a higher count. If the cache is at its capacity limit, it removes the least frequently used node from the right end of the cache before inserting the new node. If the cache is not at its capacity limit, it simply inserts the new node at the left end of the cache, unless there is already a node with an occurrence count of 1, in which case it inserts the new node after the node with an occurrence count of 1 and to the left of the next node with a higher count.

Benefits of having cache in the server:

1. Improved performance: Cache allows frequently accessed data to be stored in memory closer to the processor, reducing the latency and time required to access the data from a slower secondary storage device. This can result in improved application performance, faster response times, and reduced resource consumption.

2. Reduced network traffic: Caching can also reduce the amount of data sent over a network by storing frequently requested data locally. This reduces the need to retrieve data from a remote server or database, which can be slow and expensive.
3. Scalability: Caching can improve the scalability of an application by reducing the load on backend servers and databases. By caching frequently accessed data, the application can handle more requests without requiring additional server resources.
4. Availability: Caching can also improve the availability of an application by providing a fallback mechanism when the primary data source is unavailable. Cached data can be served to users during periods of downtime or when a service disruption occurs.
5. Cost savings: Caching can result in cost savings by reducing the need for expensive hardware and network infrastructure. By caching data, applications can operate more efficiently with fewer resources, which can save on operating costs and capital expenditures.

3. LOOKUP-TABLE

The lookup table is present in each node to hold the location to be accessed for each data item requested. Having a lookup is beneficial because it helps us to easily know which nearby node has the data that is requested. This information can be used to fetch the required data in the shortest way possible.

CONCLUSION:

In conclusion, optimizing data retrieval and improving user experience in e-commerce websites is a complex task that requires careful consideration of algorithms, data structures, and performance optimization techniques. While BFS, Floyd Warshall, and Dijkstra's algorithm can be effective in achieving these goals, they also have limitations that should be taken into account.

In order to continue improving e-commerce websites, developers can explore areas such as machine learning, natural language processing, distributed computing, and graph databases. By leveraging the latest technologies and algorithms, developers can create websites that are faster, more efficient, and provide better user experiences.

Working on a project to optimize data retrieval and improve user experience in an e-commerce website can provide valuable learning experiences in algorithms, data structures, performance optimization, and collaboration. These skills can be useful in a variety of other software development projects and can help to improve the quality and functionality of software systems.

SOLUTION TO THE PROPOSED PROBLEM:

We used algorithms such as Dijkstra's to calculate the shortest path from a user's location to the server that is hosting the data they are requesting. This will allow the company to route data requests more efficiently, reducing the time required for data to travel between servers and improving page load times for users.

LIMITATION:

While the algorithms mentioned (BFS, Floyd Warshall, and Dijkstra's algorithm) can be effective in optimizing data retrieval in an e-commerce website, they also have some limitations. While BFS can be useful for finding related products, it can also lead to a large amount of data being retrieved, which can slow down the website's performance. Additionally, BFS may not always be the best algorithm to use for complex search queries or when there are a large number of nodes in the graph.

While Floyd Warshall can be effective in finding the shortest path between all pairs of nodes in a graph, it can be computationally expensive and take a long time to execute, especially when dealing with large graphs. This can result in slower response times for the user. While Dijkstra's algorithm is efficient in finding the shortest path between two nodes in a graph, it assumes that all edges have non-negative weights. If there are negative weights, this algorithm may not work properly and may give incorrect results. Additionally, Dijkstra's algorithm can be slow when dealing with large graphs, which can lead to slower response times for the user.

FUTURE WORK:

There are many potential areas for future work in optimizing data retrieval and improving user experience in e-commerce websites. By leveraging the latest technologies and algorithms, developers can continue to improve the performance and functionality of these websites to better serve their users.

LEARNING:

Working on a project to optimize data retrieval and improve user experience in an e-commerce website provided valuable learning experiences in algorithms, data structures, performance optimization, and collaboration. These skills can be useful in a variety of other software development projects and can help to improve the quality and functionality of software systems. Working on the project provided the experience of working as part of a team and communicating effectively to achieve project goals. Most importantly the project provided hands-on experience in working with algorithms such as BFS, Floyd Warshall, and Dijkstra's algorithm.

APPENDIX:

BACKEND:

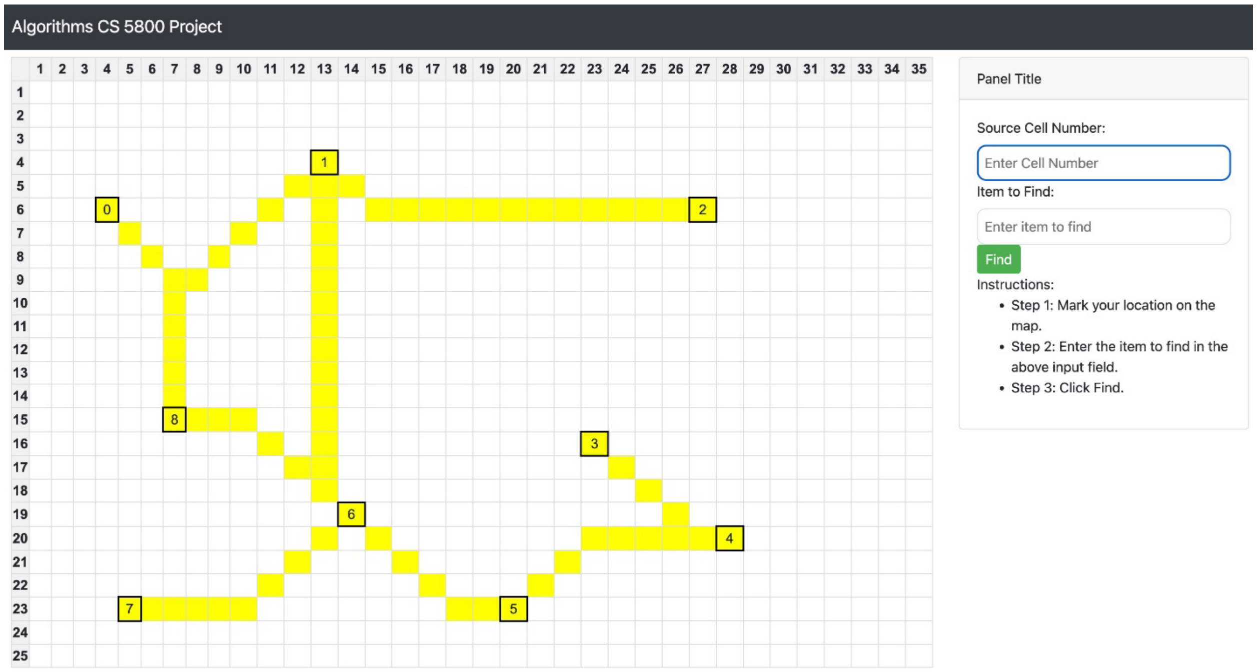
```
from collections import deque
from heapq import heappop, heappush

class Node:
    def __init__(self, key, val):
        self.key, self.val = key, val
        self.occurences = 1
        self.next, self.prev = None, None


class LFUCache:
    def __init__(self, capacity: int):
        self.capacity = capacity
        self.cache = {}
        self.left, self.right = Node(0, 0), Node(0, 0)
        self.left.next, self.right.prev = self.right, self.left

    def insert(self, prev, Node):
        prevNext = prev.next
        prev.next, prevNext.prev = Node, Node
        Node.next, Node.prev = prevNext, prev

    def get(self, key: int) -> int:
        if self.capacity == 0:
            return -1
        if key in self.cache:
            self.cache[key].occurences += 1
            self.moveLeft(self.cache[key])
            return self.cache[key].val
        else:
            return -1

    def moveLeft(self, node):
        cur = node.occurences
        prevLeft = node.prev
        left = prevLeft
        prevRight = node.next
        while left != self.left and left.occurences <= cur:
            left = left.prev
        if prevLeft == left:
            return
        prevLeft.next, prevRight.prev = prevRight, prevLeft
        self.insert(left, node)

    def moveRight(self, node):
        first = self.left.next
        cur = node.occurences
        prevLeft = node.prev
        left = prevLeft
        prevRight = node.next
        while left != self.left and left.occurences <= cur:
            left = left.prev
```

```python
        if prevLeft == left:
            return
        prevLeft.next, prevRight.prev = prevRight, prevLeft
        self.insert(left, node)

    def put(self, key: int, value: int) -> None:
        if key in self.cache.keys():
            self.cache[key].val = value
            self.cache[key].occurences += 1
            self.moveLeft(self.cache[key])
        else:
            if self.capacity == 0:
                return
            if self.capacity == len(self.cache.keys()):
                prev = self.right.prev
                prev.prev.next, self.right.prev = self.right, prev.prev
                del self.cache[prev.key]
            newNode = Node(key, value)
            self.cache[key] = newNode
            if self.left.next.occurences == 1:
                self.insert(self.left, newNode)
            else:
                prev = self.right.prev
                prev.next, self.right.prev = newNode, newNode
                newNode.next, newNode.prev = self.right, prev
                self.moveLeft(newNode)


class Server:
    def __init__(self, x, y, vals, number):
        self.cache = LFUCache(2)
        self.storage = vals
        self.x = x
        self.y = y
        self.number = number
        self.lookup = {}
        self.neighbor = set()

    def addNeighbor(self, server, distance):
        self.neighbor.add((server, distance))

    def updateCache(self, val):
        self.cache.put(val, val)

    def addLookup(self, data, dest):
        self.lookup[data] = dest


class UserNode:
    def __init__(self, x, y):
        self.x = x
        self.y = y


def initialize():
    serverMap = {}
    maxRows = 25
    maxCols = 25
    serverlocations = {}

    def findNearestServer(user):
        q = deque([[user.x, user.y]])
        directions = [[-1, 0], [1, 0], [0, 1], [0, -1], [1, 1], [1, -1], [-1,
1], [-1, -1]]
        visited = set()
        visited.add((user.x, user.y))
        while q:
            for i in range(len(q)):
                r, c = q.popleft()
                for dr, dc in directions:
                    nr, nc = r + dr, c + dc
                    if (nr, nc) in serverlocations:
                        return serverlocations[(nr, nc)]
                    if nr in range(maxRows) and nc in range(maxCols) and (nr,
nc) not in visited:
                        visited.add((nr, nc))
                        q.append([nr, nc])
        return None

    def navigateFromUserToData(user, data):
        src = findNearestServer(user)
        path = navigateToData(src, data)
        return path

    def createServer(x, y, data, number):
        if x not in range(maxRows) or y not in range(maxCols):
            return
        server = Server(x, y, data, number)
        serverMap[number] = server

    createServer(10, 10, ["A", "B", "C"], 0)
    createServer(20, 20, ["B", "C", "D"], 1)
    createServer(15, 20, ["E", "A", "F"], 2)
    createServer(10, 12, ["A", "C", "X"], 3)

    def getServerLocations():
        for serverNumber, server in serverMap.items():
            serverlocations[(server.x, server.y)] = serverMap[serverNumber]

    INF = 999
    G = [[0, 3, INF, 5],
         [3, 0, 1, 4],
         [INF, 1, 0, INF],
         [5, 4, INF, 0]]
    for row in range(len(G)):
        for col in range(len(G)):
            if G[row][col] != INF and G[row][col] != 0:
                serverMap[row].addNeighbor(serverMap[col], G[row][col])
                serverMap[col].addNeighbor(serverMap[row], G[row][col])

    distance = list(map(lambda i: list(map(lambda j: j, i)), G))

    def floyd_warshall():
        for k in range(len(G)):
            for i in range(len(G)):
                for j in range(len(G)):
                    distance[i][j] = min(distance[i][j], distance[i][k] +
distance[k][j])

    adjList = {}

    def calculate_data_adjList(distance):
        for node in range(len(G)):
            for i, dist in enumerate(distance[node]):
                for data in serverMap[i].storage:
                    if (node, data) not in adjList:
                        adjList[(node, data)] = (i, dist)
                    else:
                        adjList[(node, data)] = (i, dist) if dist <
adjList[(node, data)][1] else \
                        adjList[(node, data)]

    def update_lookups(adjList):
        for node, data in adjList:
            serverMap[node].addLookup(data, adjList[(node, data)][0])

    def navigateToData(src, data):
        if src.cache.get(data) != -1:
            return [src.number]
        dest = src.lookup[data]
        minHeap = [(0, src, [])]
        visited = set()
        while minHeap:
            distance, node, path = heappop(minHeap)
            if node in visited:
                continue
            visited.add(node)
            path.append(node.number)
            if node.number == dest:
                src.updateCache(data)
                serverMap[dest].updateCache(data)
                return path
            for nei, dist in node.neighbor:
                if nei not in visited:
                    heappush(minHeap, (distance + dist, nei, path[:]))
        return []

    getServerLocations()
    floyd_warshall()
    calculate_data_adjList(distance)
    update_lookups(adjList)

    user = UserNode(15, 10)
    print(navigateFromUserToData(user, "X"))
    print(navigateFromUserToData(user, "X"))
    print(navigateFromUserToData(user, "F"))
    print(navigateFromUserToData(user, "F"))
    print(navigateFromUserToData(user, "X"))


initialize()
```

FRONTEND:

```
import time
import matplotlib.pyplot as plt
import pandas as pd
from collections import deque
from heapq import heappop, heappush
from UserNode import UserNode
from Server import Server

from flask import Flask, render_template, request, redirect, url_for, jsonify

app = Flask(__name__)


defaultData = {
    "input":"",
    "source":""
}

serverMap = {}
maxRows = 35
maxCols = 25
serverlocations = {}
INF = 999
G = [[0, INF, INF, INF, INF, INF, INF, INF, 8],
     [INF, 0, 13, INF, INF, INF, 14, INF, 11],
     [INF, 13, 0, INF, INF, INF, INF, INF, INF],
     [INF, INF, INF, 0, 4, INF, INF, INF, INF],
     [INF, INF, INF, 4, 0, 7, INF, INF, INF],
     [INF, INF, INF, INF, 7, 0, 5, INF, INF],
     [INF, 14, INF, INF, INF, 5, 0, 8, 6],
     [INF, INF, INF, INF, INF, INF, 8, 0, INF],
     [8, 11, INF, INF, INF, INF, 6, INF, 0]]
adjList = {}

distance = list(map(lambda i: list(map(lambda j: j, i)), G))

@app.route('/', methods=['GET', 'POST'])
@app.route('/home', methods=['GET', 'POST'])
def home():
    start()
    if request.method == 'POST':
        defaultData['input'] = request.form["input"]
        defaultData['source'] = request.form["source"]
        return redirect(url_for('path'))
    return render_template('index.html', defaultData=defaultData, serverMap =
serverMap)

@app.route('/findPath', methods=['GET', 'POST'])
def path():
    if request.method == 'POST':
        defaultData['input'] = request.form["input"]
        defaultData['source'] = request.form["source"]
    print(defaultData)
    # print(serverMap[0].cache)
    data = defaultData["source"].split("-")
    x = int(data[2])
    y = int(data[1])
    user = UserNode(x,y)
    output = navigateFromUserToData(user, defaultData['input'])
    print(output)
    x = defaultData["source"].split("-")
    y = int(x[1])
    x = int(x[2])
    source = []
    source.append([y,x])
    return render_template('index.html', defaultData=defaultData, output=output,
source=source,
                           cells_to_color=output, serverMap = serverMap, bfs=bfs)

def start():
    createServer(4, 6, data[0], 0)
    createServer(13, 4, data[1], 1)
    createServer(27, 6, data[2], 2)
    createServer(23, 16, data[3], 3)
    createServer(28, 20, data[4], 4)
    createServer(20, 23, data[5], 5)
    createServer(14, 19, data[6], 6)
    createServer(5, 23, data[7], 7)
    createServer(7, 15, data[8], 8)

    for row in range(len(G)):
        for col in range(len(G)):
            if G[row][col] != INF and G[row][col] != 0:
                serverMap[row].addNeighbor(serverMap[col], G[row][col])
                serverMap[col].addNeighbor(serverMap[row], G[row][col])

    getServerLocations()
    floyd_warshall()
    calculate_data_adjList(distance)
    update_lookups(adjList)
```

SCREENSHOTS:



Algorithms CS 5800 Project

Panel Title

Source Cell Number:

Enter Cell Number

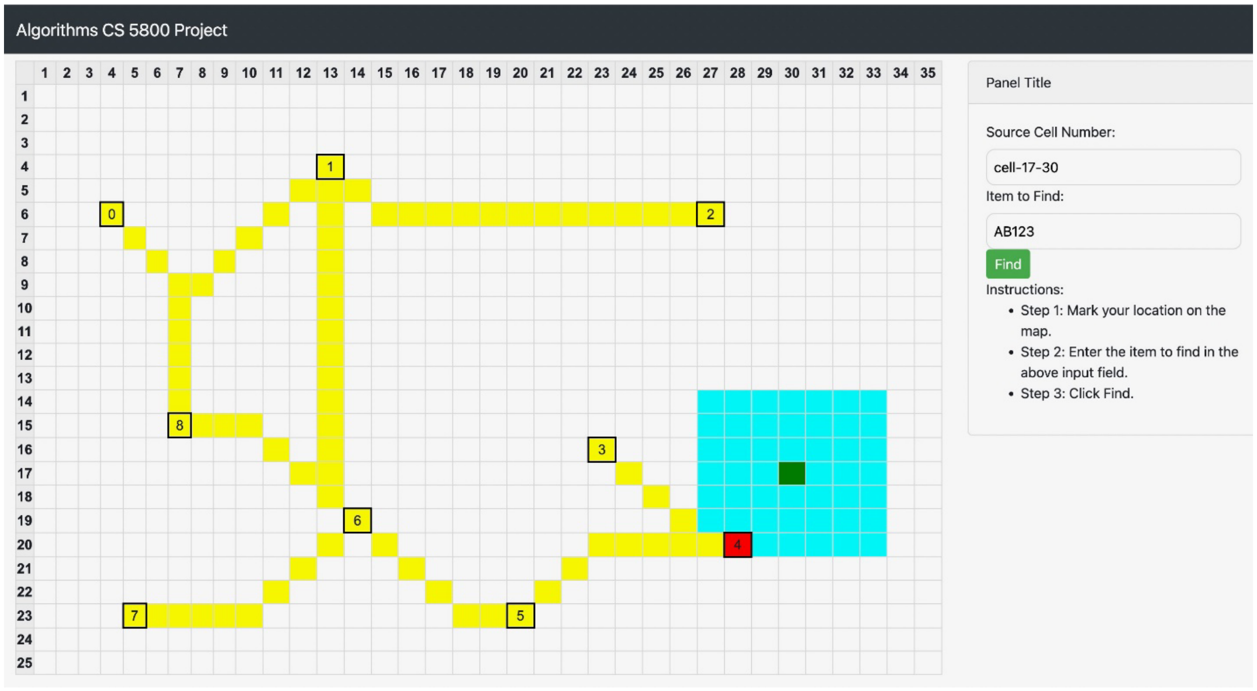
Item to Find:

Enter item to find

Find

Instructions:

- Step 1: Mark your location on the map.
- Step 2: Enter the item to find in the above input field.
- Step 3: Click Find.



Algorithms CS 5800 Project

Panel Title

Source Cell Number:

cell-17-30

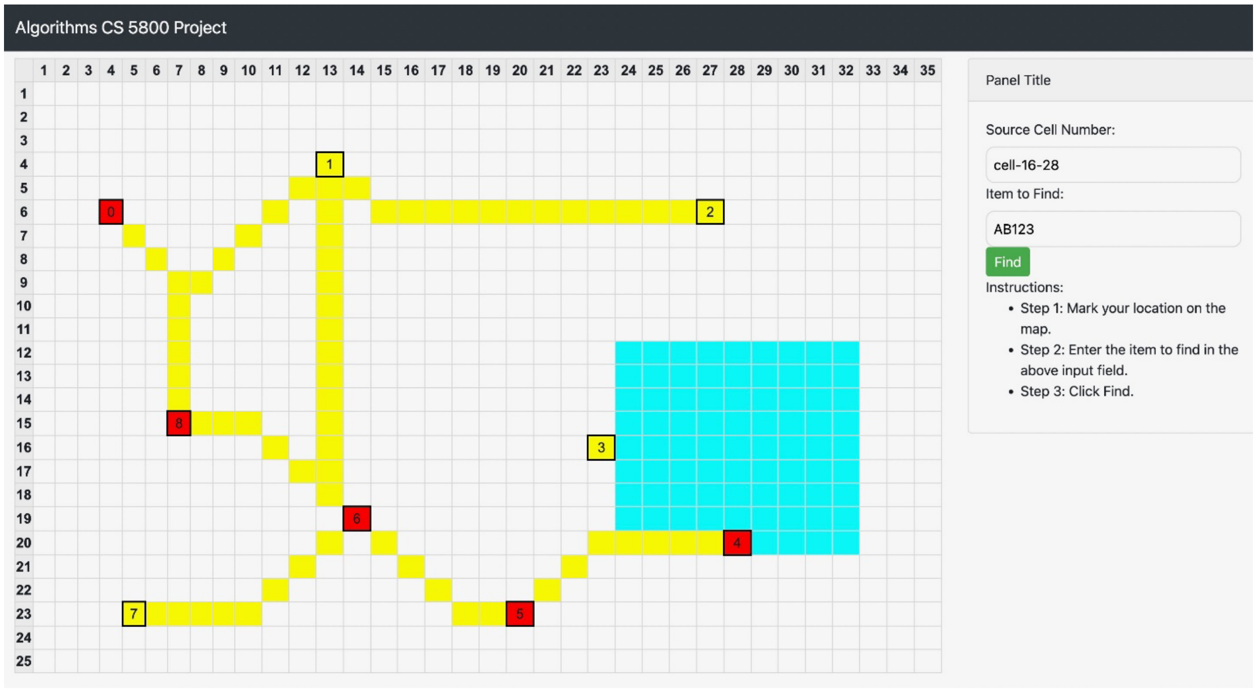
Item to Find:

AB123

Find

Instructions:

- Step 1: Mark your location on the map.
- Step 2: Enter the item to find in the above input field.
- Step 3: Click Find.


Algorithms CS 5800 Project

Panel Title

Source Cell Number:

cell-16-28

Item to Find:

AB123

Find

Instructions:

- Step 1: Mark your location on the map.
- Step 2: Enter the item to find in the above input field.
- Step 3: Click Find.