Team Project Report

PathFinder

Listing of team members and their workload or contribution to the project

Sheharyar Khan - Full stack development, programming the backend and developed the pathfinding algorithm

Suraj Shah - Developed the frontend html and javascript pages to visualize the algorithm on a map

Xiuying Li - Worked on the frontend html page and major contributions on lab report

Purpose

The purpose of this team project is to combine life problems with theoretical knowledge. Solve practical problems by using the knowledge we learned, including C++ knowledge and logic operation knowledge. Now, COVID-19 is a major issue facing all over the world, which brings people a lot of trouble. To limit the spread and exposure of COVID-19 governments all across the world on the local and federal level have instantiated stay at home orders. Although people follow the stay at home order, shopping is still a big issue for people. Despite having the ability to leave the home for groceries, many people are concerned with the amount of time that they spend in a public setting with the risk of getting exposed. To tackle the problem of shopping for essentials during the pandemic, our group has decided to create PathFinder. It will help people find the shortest path from entrance to the items they want. It will reduce people's shopping time and allow people to have less contact with others.The target users are people that are uncomfortable to, or can't afford to order groceries online. It will also serve to protect the people shopping for delivery apps like Instacart. It is also beneficial for old people because they have high risk of infection during COVID-19.

Planning and organization

Planning: Our vision for this program is to allow the user to select multiple items and add to the shopping cart, then allow the customer to select a map, and then display the shortest path to buy all select items in the store. We plan to use an algorithm to find the shortest path from the entrance to all the items in the grocery list to the cash register. Our team has decided to read a map of the store using a CSV or JSON formatted file. We will parse this file into a grid using trees. Each tree node will contain a pointer to the node that is to the right of it, and above it. Each node pointer will also be stored in a map so that we can use random access to find the position fasters and generate a route that way. We also plan on using a C++ backend combined with a Javascript front end using WASM.

Planning the find_path function

```
// Plan for Pathfinding Algorithm
/*
 * ////TODO: Make a function that finds the shortest path
 * Approach: Recursion - Brute force
 ? Should I use an approach with loops or should I focus on a recursive approch and covert it to loops later on?
 @param
 * What to look out for:
    -   repition: don't search along the same path over and over again
TODO:   -   memory: probably going to have to convert this to a loop approach later on.
    -   Check for in boundaries : make function
    -   Keep track of steps taken : use a container of Coords
    -   returns shortest distance.
 *  requirements: (Args)
    -   Map to traverse through
    -   Current Coords to know where we are
    -   Final Coords to know when to stop
    -   Visited Coords to know not to repeat
*   return:
    -   Path
    ? What defines a path?
    -   An ordered list of Coords that connect Point A to Point B
*   Algorithm:
    create new empty Path
    save current coord in Path

    Check if Point B was reached
        - if true: return the Path
    Check if current coord is a valid position on the map.
        - if false: return an empty Path

    Create a container of paths
    add the path returned by the recursive call of the function to the container
    remove all failed/empty paths from the container
    check if the container contains elements
        - if false: return an empty path; there is no way to get to Point B from here
    sort the container by size of each path
    add shortest path in the container to the end of the Path in the function
    return the Path
 */
```

This image somewhat shows the thought process that the code is built upon. In order to make this we asked ourselves what needed to be done to find a path and what the issues we could face. The "what to look out for" part is what we used to establish our base cases for this recursive algorithm. The requirements section are the parameters for the function. The algorithm is briefly explained here as well.

Organization: First, create a path to be returned, then add the current position to the path. If the end point is reached, return the path. If the path leads out of bounds, return an empty path. If the default argument is given, then set checked equal to.

```
Path Pathfinder::find_path(Coords pos, Coords end, bool** checked)//, size_t current_size/*, std::string dir*/)
{
    // creates path to be returned
    Path final;
    // we add the current position to the path
    final.push_back(pos);
    //  if the end point is reached return the path
    if(pos == end)
    {
        // std::cout << "pos == end; "<< coord_to_string(pos) << "==" << coord_to_string(end) << std::endl;
        return final;
    }
    //  if path leads to out of bounds return an empty path
```

Then, use vectors to check the directions. First go to right, then go left, go down, and finally go up. We create a container of paths, and add the path returned by the recursive call for the function to the container. Then remove all failed and empty paths from the container. Check if the container contains elements. If false, there is no way to get to the end point, return an empty path.

```
std::vector<Path> paths;
// right
paths.push_back(find_path(Coords(pos.first+1,pos.second), end, check(pos, checked)));//, current_size + 1/*, dir+"r"*/));
// left
paths.push_back(find_path(Coords(pos.first-1,pos.second), end, check(pos, checked)));//, current_size + 1/*, dir+"l"*/));
// down
paths.push_back(find_path(Coords(pos.first,pos.second+1), end, check(pos, checked)));//, current_size + 1/*, dir+"u"*/));
// up
paths.push_back(find_path(Coords(pos.first,pos.second-1), end, check(pos, checked)));//, current_size + 1/*, dir+"d"*/));
```

After removing the empty path, sort paths by size in ascending order and concatenate the shortest path to the final path. If path size equals zero, shows no valid paths found. If the path leads to the end, return this path.

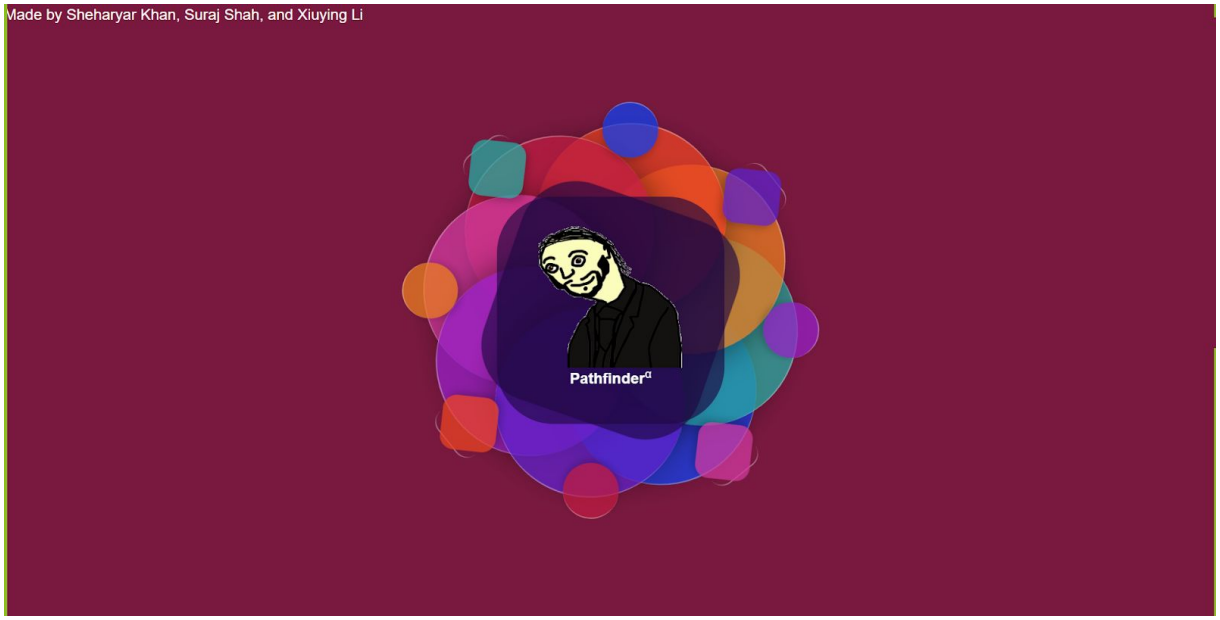
```
if(paths.size() == 0)
{
    // std::cerr << "Error: No valid paths found." << std::endl;
    return Path();
}
for(size_t i = 0; i < paths[0].size(); i++)
{
    final.push_back(paths[0][i]);
}
```

After completing the pathfinding algorithm we need to display the result on an HTML page. We tested and brainstormed several ways to do this and eventually found the canvas javascript element. We first create a canvas element and then begin a path using ctx.beginPath(). After that we use ctx.moveTo(x, y) and ctx.lineTo(x, y) to specify the start and end coordinates of the line we would like to draw. Finally using ctx.stroke() and ctx.strokeStyle we can connect the two coordinates and draw a line with a specified color through them.

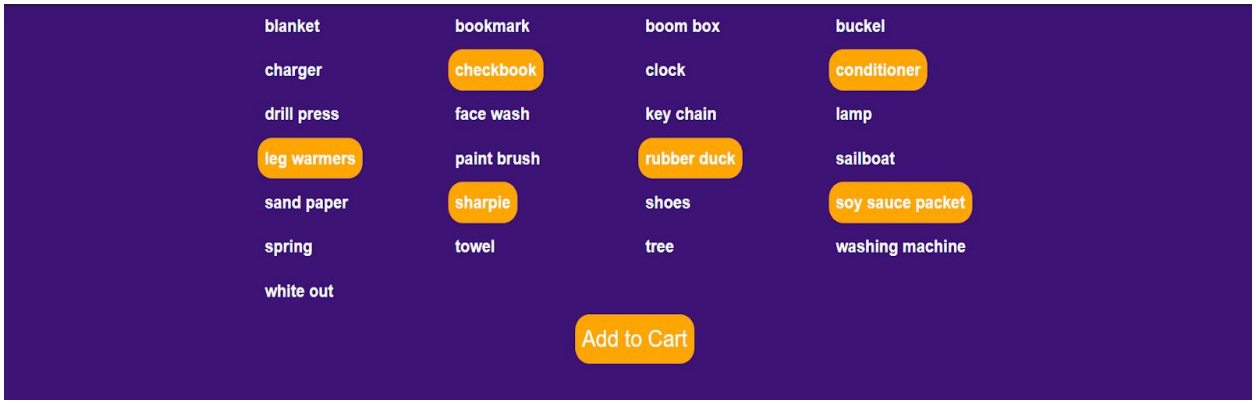
```
function draw() {
// Assign our canvas element to a variable
var canvas = document.getElementById("canvas1");
  // Create the HTML5 context object to enable draw methods
var ctx = canvas.getContext("2d");

  ctx.beginPath();
      ctx.moveTo(72, 60);
      ctx.lineTo(100, 72);
      ctx.lineTo(71, 94);
  ctx.lineTo(50, 70);
  ctx.lineTo(65, 40);
ctx.strokeStyle = "#FF00F0";
ctx.stroke();
```

Product

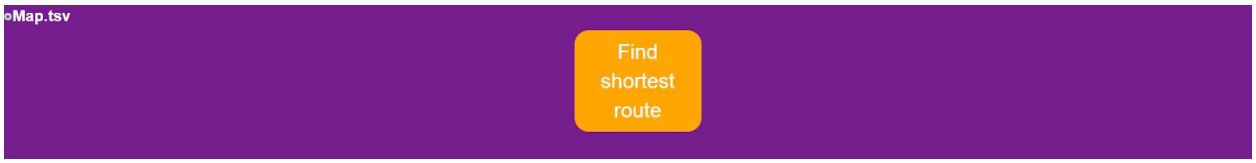
At the home page we're greeted by the logo of the project and a nice css animation. If you click on the circles it will change the background color. If you click on the center square it will take you to the next page.



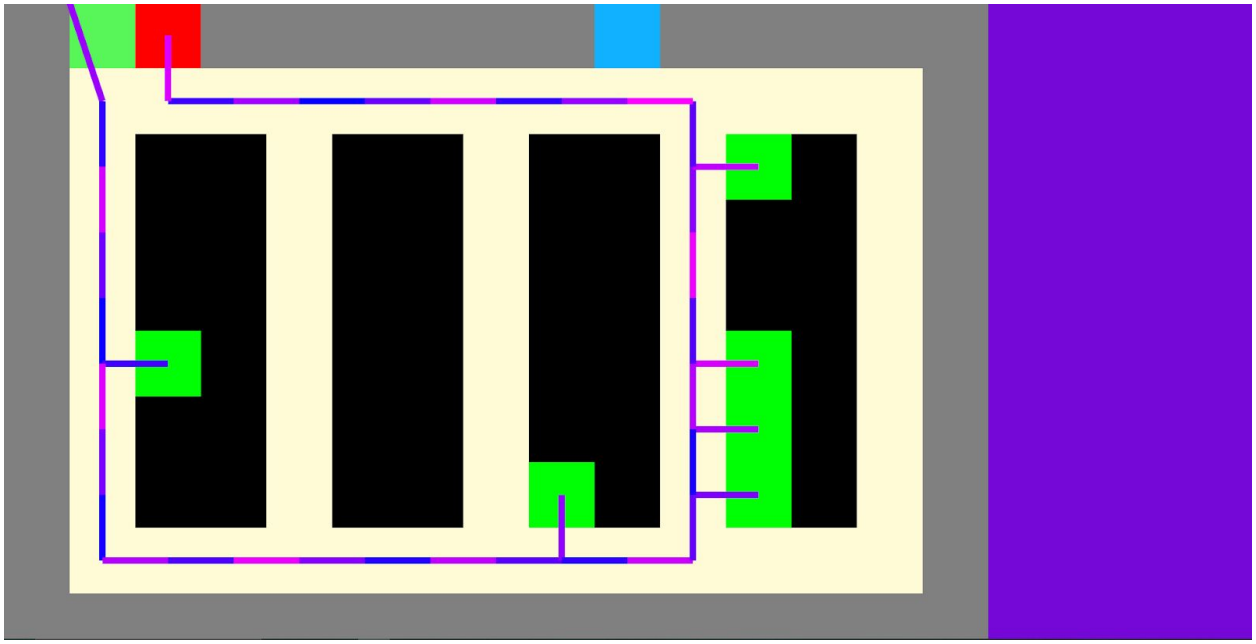
The next page is the inventory page. This is where you select the items you want and add them to the cart.



After adding the items to the cart you will be prompted to select a store. At this point we only have one store “Map.tsv” so you’ll have to select that and then click find the shortest route.



Upon clicking find shortest route it will display the shortest route on an html canvas element. You can right click the map to save it as an image.



Finding the path is an extremely heavy task. At each point there are four directions to check. So for a simple map with 4 coordinates there are already over 10 paths. And for a 25 point map that number rises to over 264 thousand. For an efficient map it is recommended that paths next to each other be grouped together to form a single path. This makes the dimension of the map smaller.

We used logical statements/ arguments all throughout the program. For example in our Cart.cpp file we needed to read and parse the input text file. Here we used an if statement and for loops to verify if the file was open and otherwise to end the program. We also implemented a while loop here to get and store each line of the file until we reach the end of the file.

```
// Reads and parses the text file
void    Cart::readList(std::string      filename)
{
        // create input stream object
        std::ifstream   in(filename);
        // check if the file was opened
        if(!in.is_open())
        {
                std::cerr << "Could not open file to read grocery list." << std::endl;
                // exit the program
                exit(-1);
        }
        // create string to read in each line
        std::string             line;
        while (!in.eof())
        {
                std::getline(in, line);
                // add line to list
                _list.push_back(line);
        }
        in.close();
```

We also used functions all throughout the project. One clear example is in our Pathfinder.cpp file where we used 6 getter functions as a part of our Pathfinding algorithm. This can be seen below.

```
// Getters
Map                     Pathfinder::map()       const
{
        return _map;
}
Map*            Pathfinder::map_view()  const
{
        return (Map*)(&_map);
}
Path            Pathfinder::path()      const
{
        return _path;
}
Inventory       Pathfinder::inventory() const
{
        return  _inv;
}
Path            Pathfinder::items()     const
{
        return  _items;
}
Cart            Pathfinder::cart()      const
{
        return _cart;
}
```

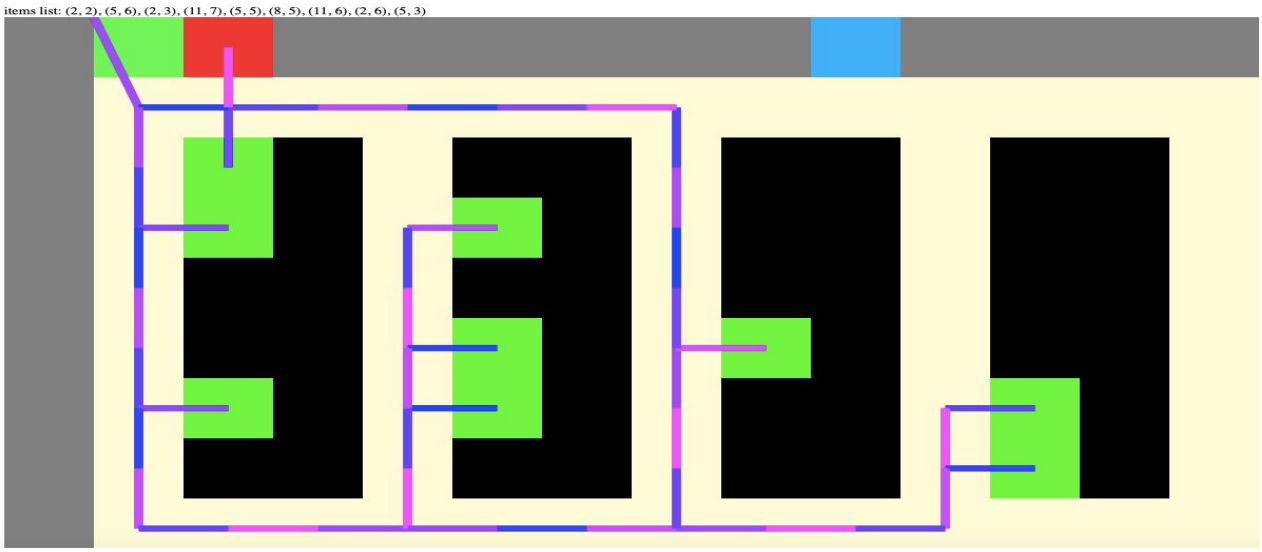
We use many counts in the program. For example, we use one counting in our pathfinder.cpp file to draw lines with different colors.

```
for (size_t i = 1; i < _path.size(); i++)
{
   std::cout << "ctx.lineTo(" << _path[i].first*map().scale() + map().scale()/2.0 << ", " << _path[i].second*map
   << "R = " << (255 - i*map().scale()) % 256 <<";" << std::endl
   << "G = " << 0<<";" << std::endl
   << "B = " << 255<<";" << std::endl

   <<   "color = \'rgba(\' + R + \', \' + G + \', \' + B + \', 1)\';" << std::endl
   <<   "ctx.strokeStyle = " << "color" << ";" << std::endl
   <<   "ctx.stroke(); ctx.closePath(); " << std::endl;
   std::cout <<   "ctx.beginPath(); ctx.moveTo(" << _path[i].first*map().scale() + map().scale()/2.0 << ", " << _
}
```

Recursion was essential to our Pathfinding algorithm as you can see from our brute force approach. As you can see from the screenshot below, we create a container of paths and add paths returned by the recursive call of the function to the container. After this we remove all the failed or empty paths from the container and then check if it contains any elements. On a basic level the program will continue to move in its current direction unless it hits a wall it cannot go past or reaches its final destination, the item.

```
// list paths which will contain the path checked in each direction
std::vector<Path> paths;
//      right
paths.push_back(find_path(Coords(pos.first+1,pos.second), end, check(pos, checked)));//, current_size + 1/*, dir+"r"*/));
//      left
paths.push_back(find_path(Coords(pos.first-1,pos.second), end, check(pos, checked)));//, current_size + 1/*, dir+"l"*/));
//      down
paths.push_back(find_path(Coords(pos.first,pos.second+1), end, check(pos, checked)));//, current_size + 1/*, dir+"u"*/));
//      up
paths.push_back(find_path(Coords(pos.first,pos.second-1), end, check(pos, checked)));//, current_size + 1/*, dir+"d"*/));
```

Limitations: One limitation of our program is that the exit route and entry route are the same color. If many parts of two lines are repeated together, it may confuse the user.



Another limitation is this program does not work for multiple entry points. Now it only works for one entry.

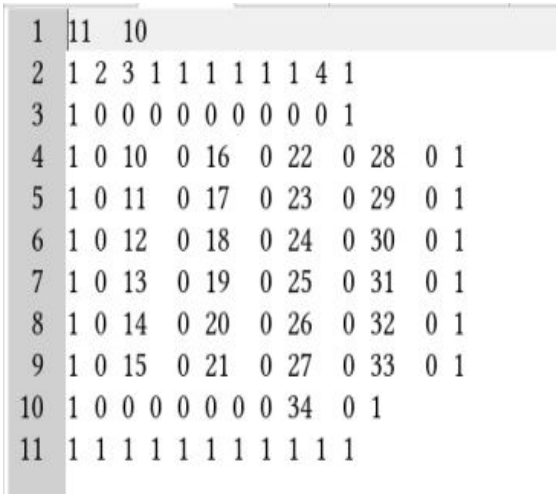
Also, the algorithm of our program is not simple enough. It is an inefficient brute force algorithm, and it is completely dependent on the user's ram.

Pitfalls

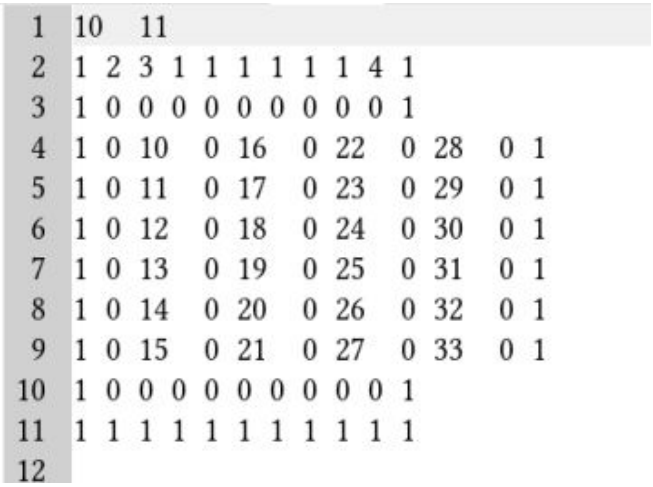
One challenge we met is that the algorithm of finding the shortest path didn't work. When we ran it, it did show up the shortest path. It only randomly displays several paths. We decided to debug step by step to fix this problem. We found that we reversed the number of rows and columns in the map. The map input file had the height and width switched up, so it resulted in the error when reading the map. After we fixed the number of rows and columns, the shortest path was calculated successfully. The screenshot below is an example of how we implemented the width and height to read the map out and where we had made this error of swapping the variables value.

```
// prints map for debugging
void    Map::printMap()
{
        // prints map exatly as it was read
        std::cout << _width << "\t" << _height << std::endl;
        for(size_t i = 0; i < _height; i++)
        {
                for(size_t j = 0; j < _width; j++)
                {
                        std::cout << _map[i][j];
                        if(j == _width - 1)
                                std::cout << std::endl;
                        else
                                std::cout << "\t";
                }
        }
}
```

Additionally, here are two screenshots before and after we fix it.



```
11   10
1 2 3 1 1 1 1 1 1 4 1
1 0 0 0 0 0 0 0 0 0 1
1 0 10   0 16   0 22   0 28   0 1
1 0 11   0 17   0 23   0 29   0 1
1 0 12   0 18   0 24   0 30   0 1
1 0 13   0 19   0 25   0 31   0 1
1 0 14   0 20   0 26   0 32   0 1
1 0 15   0 21   0 27   0 33   0 1
1 0 0 0 0 0 0 0 0 34   0 1
1 1 1 1 1 1 1 1 1 1 1
```



```
10   11
1 2 3 1 1 1 1 1 1 4 1
1 0 0 0 0 0 0 0 0 0 1
1 0 10   0 16   0 22   0 28   0 1
1 0 11   0 17   0 23   0 29   0 1
1 0 12   0 18   0 24   0 30   0 1
1 0 13   0 19   0 25   0 31   0 1
1 0 14   0 20   0 26   0 32   0 1
1 0 15   0 21   0 27   0 33   0 1
1 0 0 0 0 0 0 0 0 0 1
1 1 1 1 1 1 1 1 1 1 1
```

The other issue was there was an infinite loop the first time when writing the algorithm, and it didn't keep tracking. It just keeps going over and over again about the same path. To fix it, we made an array that would mark all the locations it had been to. All the paths were sharing this array.

Possible improvements

One possible improvement is that we can distinguish the color of the entrance and the item. So far, they are both green. If we can seperate the color, it can make the user more clearly

distinguish the entry and items. Also, we can use different colors to distinguish the exit route and entry route. In this way, when people go shopping, they can clearly distinguish the way in and the way out. The radio button on the Map page also needs additional CSS so it is easier to see what exactly you are clicking and so that the button does not blend into the background with the same color. We can make each route use a different color.

We also need to add more maps and inventory, so far, we only have one map. For each map, we need to add one inventory. This will give users more choices about the map.

Another possible improvement is that we can use a simple and efficient algorithm to avoid some of the possible pitfalls, the current algorithm is a time-consuming and inefficient algorithm, it has a lot of limitations.

In addition, if we had more time, we could try to make the program allow multiple entrances and exits. Let users choose which entrance they want to enter the supermarket from and which exit they want to exit from. Most supermarkets have multiple entrances. This allows users to plan their shopping routes and time more freely and efficiently.

Overall, this team project combines the knowledge of C++ with the knowledge of logical operations in this class, and then links them to real life examples. This allows students to think more about what they are learning. Time permitting, students can make a very perfect program. One possible suggestion is that it would be more helpful if students had more time to plan and think about the project.