Developer Manual

Name: Lazy picker

Author: JYZ Team

- Jiati Zhao
- Zeqiang Zheng
- Yukkei He

Version: Beta v2.0.0

Contents

Glossary:

- **UI:** User Interface. The part of a software application that users interact with directly, including graphical elements like buttons, text boxes, and menus.
- **UML:** Unified Modeling Language. A standardized notation for creating visual models of software systems.
- **TSP:** The Traveling Salesman Problem (TSP) is a classic optimization problem in computer science and mathematics. It involves finding the shortest possible route that a salesman can take to visit a given set of cities and return to the starting point, while visiting each city exactly once. The problem is considered to be NP-hard, meaning that no efficient algorithm exists to solve it for large problem instances, and it often requires the use of approximation algorithms or heuristics to find near-optimal solutions.
- **Branch and bound:** Branch and Bound is an algorithmic technique that systematically explores the solution space of optimization problems, like the Traveling Salesman Problem. It prunes suboptimal branches and partitions the problem into smaller subproblems, using lower bounds to guide the search. By iteratively branching and bounding, it aims to find the optimal or near-optimal solution efficiently.
- **Nearest Neighbor:** The Nearest Neighbor algorithm is a heuristic for the Traveling Salesman Problem. It starts at an arbitrary city and repeatedly chooses the nearest unvisited city until all cities have been visited. While not optimal, it provides decent solutions for smaller TSP instances.
- **A Star:** A search algorithm used to find the relative short path between two points in a graph. It is widely used in game development and robotics.
- **Dijkstra:** A graph search algorithm used to find the shortest path between a starting node and all other nodes in the graph. It is commonly used in routing and network analysis.
- **BFS:** Breadth-First Search. A graph traversal algorithm that visits all the nodes at a given depth level before moving on to deeper levels. It is often used in tree traversal and graph search problems.
- **DFS:** Depth-First Search. A graph traversal algorithm that explores as far as possible along each branch before backtracking. It is often used in tree traversal and graph search problems.
- **JSON:** JavaScript Object Notation. A lightweight data interchange format that is easy for humans to read and write, and easy for machines to parse and generate. JSON is commonly used for transmitting data between a server and a web application.

Software Architecture Overview

The Lazy Picker is a standalone program designed to assist warehouse workers in efficiently picking products for orders. The program takes the product's unique ID, xLocation, and yLocation from the warehouse's existing database and provides the worker with the location of the product in the warehouse. This program does not require an internet connection and does not rely on a web server.

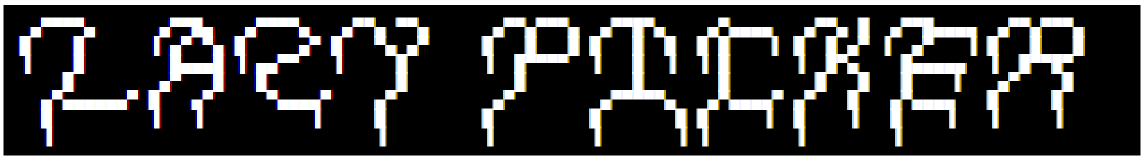


Figure 1: Logo

Main data types and structures:

In order to facilitate scalability and efficient management of data, we have designed a system where the data from the file is stored in separate classes. For instance, we have created a class called "Item" to store all relevant information pertaining to an individual item. Similarly, we have designed classes for other entities such as workers and shelves. This approach enables us to easily accommodate additional data and functionalities in the future.

The visualize map is using 2D arrays as data structure. To enhance the performance and modularity of the system, we have also mapped each worker, item and shelf into separate blocks in the map. This decouples the UI display from the real world objects and allows us to associate various attributes and characteristics with each block that are more suitable for algorithm calculations. However, it is not practical to add these characteristics to individual workers or items. Therefore, we have created a specialized "MapData" class that is responsible for storing and transforming the data from real world entities to map components.

File tree structure

```Python
+-- lazy-picker
|   +-- qvBox-warehouse-data-s23-v01.txt #database file
|   +-- data.py #data module
|   +-- entities.py #entity module
|   +-- lazy_picker.py #lazy_picker module
|   +-- service.py #service module
|   +-- tsp_algo.py #travel sales man solvng extension
|   +-- visualize.py #UI rendering part
```

data module

The data module provides necessary information and messages required by the application. These components are designed to be serializable and can be easily converted to formats such as JSON that are easy to transmit and receive.

entity module

The Entity module includes models that play a crucial role in the program. These components can represent real-world objects that the program operates on.

service module

The service module encompasses all the essential components that constitute the graph, facilitate the computation of the optimal path, and exhibit the results of the algorithm in a visually intuitive manner.

lazy_picker module

The lazy_picker module is the entry point of the program. It initializes the map data and contains all the transitions between the different states of the program and all the functions that interact with user.

tsp_algo module

The tsp_algo module contains all necessary functions and data structures to implement Traveling sales man problem(select for multiple targets)

Module interfaces & API Guide

Structure:

function_name():

- Brief description of the functionalities
- Inputs: values of `params:` tag
- Outputs: values of `return` tag
- Responsibility: values of `authors:` tag

(Note: Getter, Setter, and str() are not included in this guide)

lazy_picker

Python

```python
def read_map_data(filename):
    """
    Reads the map data from the given file.
    It first generates a list of items by reading the file line by line.
    Then it generates a list of shelves by calling the gen_shelves function.


    :param filename: A string representing the name of the file to read from.
    :return: items and shelves
    :author: Jiati Zhao
    """


def gen_shelves(items):
    """
    The gen_shelves function takes in a list of items and returns a list of shelves.
    The function first sorts the items by their position (x-coordinate, y-coordinate).
    It then iterates through the sorted list, if the current item has the same position as the previous
item,
    it adds the item to the existing shelf. Otherwise, it creates a new shelf and adds the item to the
shelf.

    :param items: A list of items read from the database file(from read_map_data function)
    :return: A list of shelves
    :author: Jiati Zhao
    """


def get_worker_pos():
    """
    The get_worker_pos function base on the user's input to create a worker.
    If the user wants to use the default position, the function returns a worker with the default
position.
    If the user wants to enter a custom position, the function prompts the user to enter the x-coordinate
and y-coordinate.
    The function then returns a worker with the custom position.

    :return: A worker object, which is used to create a worker
    :author: Jiati Zhao
    """


def peek_items(items):
    """
    Prints the items in the list.
    :param items: The list of all items(from read_map_data function)
    :author: Zeqiang Zheng
    """


def set_targets(items):
    """
    The set_target_item function takes in a list of items and prompts the user to enter an item id.
    If the input is not numeric, it will prompt again until a valid number is entered.
    It then checks if that number matches any of the item ids in the list, and returns that item if so.

    :param items: The list of all items(from read_map_data function)
    :return: The target item
    :author: Jiati Zhao
    """

def set_target_id_once(items_map):
    """
    The set_target_item function takes in a list of items and prompts the user to enter an item id.
    If the input is not numeric, it will prompt again until a valid number is entered.
    It then checks if that number matches any of the item ids in the list, and returns that item if so.

    :param items_map: a dict, mapping items id with  Item
    :return: The target item
    """

def set_targets_one_by_one(items_map):
    """
    The set_target_item function takes in a list of items and prompts the user to enter an item id.
    If the input is not numeric, it will prompt again until a valid number is entered.
    It then checks if that number matches any of the item ids in the list, and returns that item if so.

    :param items_map: Pass the list of items to the function
    :return: The target item
    """

def set_target_from_file(items_map):
    """
    The set_target_item function takes in a list of items and prompts the user to enter an item id.
    If the input is not numeric, it will prompt again until a valid number is entered.
    It then checks if that number matches any of the item ids in the list, and returns that item if so.

    :param items_map: Pass the list of items to the function
    :return: The target item
    """

def initialize_data():
    """
    The initialize_data function reads the data from the database file to get the items and shelves,
    It first calls the read_map_data function to read the data from the file,
    then calls the get_worker_pos function to get the worker's starting position,
    then calls the set_target_item function to get the target item.
    finally, it generates a MapData object with the data it got from the previous functions.

    :return: A MapData object, which contains all the data needed to create a map
    :author: Jiati Zhao
    """

def set_access_mode():
    """
    The set_access_mode function prompts the user to choose single or multiple access point mode.
    """


def display_welcome():
    """ Displays the welcome screen for the program.

    :author: Zeqiang Zheng
    """

def find_path(map_data):
    """
    The find_path function takes in a MapData object to the service module and generate a graph,
    then prompts the user to select an algorithm.
    It then calls the corresponding function in the Map class.
    If the user input 1, it calls the branch and bound function in the Map class.
    If the user input 2, it calls the dummy greedy in the Map classS



    :param map_data: The Mapdata class stores all the map info from the initial settings
    :author: Zeqiang Zheng
    """

def setting(map_data):
    """
    The setting function prompts the user to enter a new target item or algorithm.
    It then returns the new MapData structure.

    :param map_data: Pass the map data to the function
    :return: Mapdata
    :author: Zeqiang Zheng
    """

def display_menu(map_data):
    """
    Displays the menu for the user to choose from.
    The function will display a list of options and then prompt the user to enter their choice.

    :return: The chosen screen
    :author: Zeqiang Zheng
    """

def main():
    """
    The entry point of the program.
    """
```

data

Python

```
class Algorithm(Enum):
    """
    An enumeration of the different algorithms that can be used to solve the problem.
    0: A*, 1: BFS, 2: DFS, 3: Dijkstra

    :author: Yukkei
    """

class NodeState(Enum):
    """
    An enumeration of the different states that a node can be in;
    0: GOAL, represents the target node(the target item); 1: NEW, represents a node that has not been
visited;
    2: OPEN, represents a node in the open list(will be visited in the future);
    3: BLOCK, represents a node that is a shelf; 4: PATH, represents a node that is in the path
    5: CLOSE, represents a node that has been visited; 6: START, represents the starting node(worker start
position)
    7: STOP, represents the access point of the shelf

    :author: Yukkei
    """


class MapData:
    """
    A class to store the data for the map.
    """

    def __init__(self, worker, shelves, items, targets, destination=None, time_limit=60,
algorithm=Algorithm.A_STAR, map_row=40, map_col=21,access_mode="multiple"):

    def update(self, attribute, value):
        """
        The update function takes in an attribute and a value.
        It then checks to see if the attribute is valid, and if it is,
        it updates the value of that attribute.

        :param self: Represent the instance of the class
        :param attribute: Specify which attribute of the object is being updated
        :param value: Update the attribute of the object
        :author: Yukkei
        """


    def toJSON(self):
        """
        The toJSON function is used to convert the object into a JSON string.

        :param self: Refer to the object itself
        :return: A Json String with the map_row, map_col, worker, shelves and items
        :author: Yukkei
        """
```

entities

Python

```python
class Entity:
    """Act like a basic class for extending """
    def __init__(self, x, y):


class Item:
    """ A class to represent an item in the warehouse. """

    def __init__(self, item_id, x, y):

    def toJSON(self):
        """
        The toJSON function is used to convert the object into a JSON string.

        :param self: Refer to the object itself
        :return: Return a JSON representation of the item
        :author: Jiati Zhao
        """


class Shelf(Entity):
    """ A class to represent a shelf in the warehouse."""

    def __init__(self, shelf_id, x, y):

    def add_item(self, item):
        """
        The add_item function adds an item to the list of items in a given order.

        :param self: Refer to the current instance of a class
        :param item: The item to be added
        :return: The item that was added to the list of items on this shelf
        :author: Yukkei
        """


    def remove_item(self, item):
        """
        The remove_item function removes an item from the list of items in a room.

        :param self: Refer to the instance of the class
        :param item: Specify the item that is being removed from the list
        :author: Yukkei
        """


    def get_item(self, item_id):
        """ Return the item with the given name.

        :param self: Refer to the instance of the class
        :param item_id: Specify the item that is being returned
        :return: The wanted item
        :author: Yukkei
        """

    def get_item_count(self):
        """
        The get_item_count function returns the number of items in the list.

        :param self: Represent the instance of the class
        :return: The number of items in the list
        :author: Yukkei
        """

    def toJSON(self):
        """
        The toJSON function is used to convert the object into a JSON string.

        :param self: Refer to the object itself
        :return: Return a JSON representation of the shelf
        :author: Yukkei
        """

class Worker(Entity):
    def __init__(self, x, y):

    def pick_up_item(self, item):
        """
        The pick_up_item function takes an item as a parameter and sets the player's is_carrying attribute
to True.
        It also sets the carrying_item attribute to the item that was passed in.

        :param self: Refer to the object itself
        :param item: Set the carrying_item attribute to the item
        :return: The carrying_item
        :author: Zeqiang Zheng
        """


    def drop_off_item(self):
        """
        The drop_off_item function sets the is_carrying attribute to False and the carrying_item attribute
to None.
        This means that after this function is called, the robot will no longer be carrying an item.

        :param self: Refer to the instance of the class
        :author: Zeqiang Zheng
        """

    def toJSON(self):
        """
        The toJSON function is used to convert the object into a JSON string.

        :param self: Refer to the object itself
        :return: Return a JSON representation of the worker
        :author: Yukkei
        """
```

service

Python

```python
class Block:
    def __init__(self, x, y):

    def cal_total_cost(self, new_total_cost):
        """ The cal_total_cost function calculates the total cost of the current node.
        It does this by adding the new total cost and the given cost of the current node.

        :param new_total_cost: The new total cost of the current node
        :return: The total cost of the current node(used for a Dijkstra)
        :author: Zeqiang Zheng
        """


    def set_parent(self, parent):
        """
        The set_parent function sets the parent of the current node.

        :param parent: The parent node
        :author: Yukkei
        """


    def get_parent(self):
        """
        The get_parent function returns the parent of the current node.

        :param self: Refer to the object itself
        :author: Yukkei
        """


    def cal_heuristic(self, target):
        """ The cal_heuristic function calculates the heuristic of the current node.
        It does this by using the Euclidean distance formula.

        :param self: Refer to the object itself
        :param target: The target node
        :return: The heuristic of the current node(used for a star)
        :author: Jiati Zhao
        """


    def get_final_cost(self, target):
        """ The get_final_cost function calculates the final cost of the current node.
        It does this by adding the total cost and the heuristic of the current node.

        :param self: Refer to the object itself
        :param target: The target node
        :return: The final cost of the current node
        :author: Jiati Zhao
        """


    def is_next_to(self, next_block):
        """
        The is_next_to function checks if the current node is next to the next_node.
        It does this by checking if any of the following conditions are true:

        :param self: Represent the instance of the class
        :param next_block: The input block
        :return: Whether the input block is adjacent to the current block
        :author: Yukkei
        """



class Map:
    """A class to represent the map of the warehouse."""

    def __init__(self, map_data):

    def init_for_tsp(self):
        """
        Initialize the map for the TSP algorithm.
        First, set all the target nodes.
        Second, set the adjacency map.
        """

    def find_single_target(self, start=None, target=None):
        """
        Find the shortest path to a single target. the algorithm defaults to A*. But you can change it to
other algorithms.

        :param start: The start node
        :param target: The target node
        :return: A list of nodes representing the path from the worker to the target.
        """
    def reset_map(self):
        """
        Reset the map for the next iteration.
        """

    def iterate(self, algorithm):
        """
        Allow the user to iterate through any algorithm.
        This function will call the corresponding function to the algorithm.

        :param algorithm: The algorithm to be used
        :param curr: The current node
        :return: A list of nodes representing the path from the worker to the target.
        :author: Yukkei
        """

    def set_target_entrances(self):
        """
        A function to set the state of the nodes in the target entrances to TARGET.
        """

    def set_adjacency_map(self, all_path_nodes):
        """
        A function to set the adjacency map of the nodes in the total path.
        The function will check all the nodes in the total path and do the following:

        """


    def a_star(self):
        """
        A function to find a path from the worker to the target using the A* algorithm.
        The function will keep iterating until it finds a path from the worker to the target.
        In each iteration, it will pick the first node from the open list,
        which is the node with the lowest final cost, and call the function astar_iterate().

        :return: A list of nodes representing the path from the worker to the target.
        :author: Jiati Zhao
        """


    def tsp(self, algorithm="branch_and_bound"):
        """
        The main function to solve the TSP problem. It calls the tsp extensition's method to solve
        :param algorithm: The algorithm to solve the TSP problem. The default is branch and bound.
        :return: The total path, the total path description and the total length.
        """

    def astar_iterate(self, curr):
        """
        A function to represent an iteration of A* algorithm.
        The function will check all the neighbours of the current node and do the following:
        1. If the neighbour is next to the target node, then the path is found.
        2. If the neighbour is a new node, then add it to the open list.
        3. If the neighbour is an open node, then check if the new final cost is lower than the current
final cost.
        4. If the neighbour is a closed node, then check if the new final cost is lower than the current
final cost.
        5. When all the neighbours are checked, add the current node to the closed list.
        6. Sort the open list by the final cost of the nodes.
        (final cost = total cost + heuristic cost, total cost = given cost + parent's total cost)

        :param curr: The current node
        :author: Jiati Zhao
        """


    def bfs(self):
        """
```

```python
        A function to find the shortest path from the worker to the target using the BFS algorithm.
        The function will keep iterating until it finds a path from the worker to the target.
        In each iteration, it will pick the first node from the open list,
        which is the earliest node to be added to the open list, and call the function bfs_iterate().

        :return: A list of nodes representing the shortest path from the worker to the target.
        :author: Yukkei
        """


    def dfs(self):
        """
        A function to find a path from the worker to the target using the DFS algorithm.
        The function will keep iterating until it finds a path from the worker to the target.
        In each iteration, it will pick the last node from the open list,
        which is the latest node to be added to the open list, and call the function dfs_iterate().

        :return: A list of nodes representing the path from the worker to the target.
        :author: Yukkei
        """


    def dfs_iterate(self, curr):
        """ A function to represent an iteration of DFS algorithm.
        The function will check all the neighbours of the current node and do the following:
        1. If the neighbour is next to the target node, then the path is found.
        2. If the neighbour is a new node, then add it to the open list.
        3. When all the neighbours are checked, add the current node to the closed list.

        :param curr: The current node
        :author: Yukkei
        """


    def bfs_iterate(self, curr):
        """
        A function to represent an iteration of BFS algorithm.
        The function will check all the neighbours of the current node and do the following:
        1. If the neighbour is next to the target node, then the path is found.
        2. If the neighbour is a new node, then add it to the open list.
        3. When all the neighbours are checked, add the current node to the closed list.

        :param curr: The current node
        :author: Yukkei
        """


    def dijkstra(self):
        """
        A function to find the shortest path from the worker to the target using the Dijkstra algorithm.
        The function will keep iterating until it finds a path from the worker to the target.
        In each iteration, it will pick the first node from the open list,
        which is the node with the lowest total cost, and call the function dijkstra_iterate().


        :return: A list of nodes representing the shortest path from the worker to the target.
        :author: Zeqiang Zheng
        """


    def dijkstra_iteration(self, curr):
        """
        A function to iterate through the Dijkstra algorithm.
        The function will check all the neighbours of the current node and do the following:
        1. If the neighbour is next to the target node, then the path is found.
        2. If the neighbour is in the open list, then update its total cost if necessary.
        3. If the neighbour is a new node, then add it to the open list.
        4. When all the neighbours are checked, add the current node to the closed list.
        5. Sort the open list by the total cost of the nodes. (total cost = given cost + total cost of the
parent node)

        :param curr: The current node
        :author: Zeqiang Zheng
        """
    def get_path(self, curr):
       """
        A function to get the path from the worker to the target.
        The function will start from the target and reach the start node recursively.
        It will update the state of the nodes to represent the path.

        :param curr: The current node
        :author: Yukkei
        """


    def get_neighbours(self, curr):
        """
        A function to get the neighbours of the current node.
        The function will check all the neighbours whether it is a block(Shelf), if it is not, add it to
the list.

        :param curr: The current node
        :return: A list of nodes representing the neighbours of the current node.
        :author: Jiati Zhao
        """



    def visualize(self, is_refresh=True, refresh_rate=0.2):
        """A function to visualize the map.
        First, the function will print the map in the terminal.
        The map will be printed in the format of a 2D grid, where origin point is in the bottom left
corner of the map.
        To make the index of the map easier to read, the function does the following:
        The map is printed row by row, the first index of each row will be printed to represent the y-
axis.
        If the x-axis index is less than 10, then the row number will be printed with 2 spaces.
        If the x-axis index is more than 10, then the row number will be printed with 1 space.
        The last row of will be printed as the x-axis.

        Then, the function will refresh the map with the given refresh rate if is_refresh signal is True.


        :param is_refresh: A boolean to determine whether to refresh the map.
        :param refresh_rate: The refresh rate of the map.
        :author: Yukkei
        """


    def print_path_description(self):
        """
        A function to print the text path description.
        The function will first check whether the path is found.
        If the path is found, then the function will print the path length and the number of iterations.
        Then the function will record each direction of the current node to the bext node.
        The text path description will be reconstructed by the recorded directions.
        Use two pointers to implement the reconstruction, one pointer points to the current direction,
        if the next direction is the same as the current direction, then the pointer will move to the next
direction.
        If the next direction is different from the current direction, then the pointer will stop and
record how many
        times the current direction appears and add the text description to the path description list.
        Then the two pointer will move to the next direction.

        :author: Yukkei
        """



def refresh():
    """
    A function to clear the screen by using the os.system() function.

    :author: Yukkei
    """


def print_banner():
    """
    Used to print the banner.
    :author: Zeqiang Zheng
    """
```

tsp_algo

```python
class Branch_n_Bound:
    """
    Branch and bound algorithm to find the shortest path
    """


    def solve(self):
        """
        Solve the problem with branch and bound algorithm
        1. Select a random vertex as the root
        2. set the other entrance with the same target to infinity
        2. Reduce the matrix
        3. Put the root into the priority queue
        4. While the priority queue is not empty:
            4.1 Get the node with the smallest cost
            4.2 If the node is a leaf node:
                return the result
            4.3 Else:
                Keep reducing the matrix and put the node into the priority queue
        """


    def reduce_matrix(self, matrix, curr_tree_node):
        """
        Reduce Matrix (rows & columns to at least one 0) = reduced cost
        1. Set the vertex of same target nodes' row and column to infinity
        2. If it is a root node, reduce the matrix to get the main reduced cost
        3. Else, the main reduced cost is the parent's main reduced cost
            3.1 get the matrix[pre_index][curr_index] the value as reduction
            3.2 reduce the matrix to get the reduced cost
            3.3 set the current node's cost to the parent's cost + reduction + reduced cost
        """

    def set_matrix(self):
        """
        Iterate through the adjacent map and set the matrix
        If the adjacent map is None, set the matrix to infinity
        Else set the matrix to the weight of the edge
        """

    def generate_result(self, curr_node):
        """
        Generate the result of the TSP problem by the final node of the tree

        :param curr_node: the current node
        :return: the final path
        """
class NearestNeighbor:
    """
    A dummy greedy algorithm to find the nearest vertex to the current vertex for every step
    """

    def __init__(self, adjacent_map, vertexes, size):


    def choose_first_vertex(self):
        """
        Choose the first vertex randomly
        :return: The first vertex
        """

    def solver(self):
        """
        Solve the problem using the nearest neighbor algorithm
        :return: the result path, path description and total cost
        """

    def backtrack(self, curr_vertex):
        """
        Backtrack until find the answer

        :param curr_vertex: The current vertex
        """

    def iterate(self, curr_vertex):
        """
        Iterate each step of the algorithm
        :param curr_vertex: The current vertex
        """

    def generate_result(self):
        """
        Generate the result path
        :return: the result path, path description and total cost
        """


def print_matrix(matrix):
    """
    Print matrix , used for debugging
    Iterate through the matrix and print each element
    :param matrix: The matrix to be printed
    """

def color_edge(edge):
    """
    Color the edges
    :param edge: The edge
    """

class Edge:
    def __init__(self, node1, node2, path, path_description):

    def __lt__(self, other):
        return self.weight < other.weight

class Vertex:
    def __init__(self, node, index, is_entrance_of=None):

    def __eq__(self, other):
        return self.block == other.block

class PathTreeNode:
    """
    This class represents a Treed node to store
    """
    def __init__(self, vertex, num_of_vertexes, cost=0, parent=None):


    def __lt__(self, other):
        return self.cost < other.cost
```

Control Flow

Overall program control flow

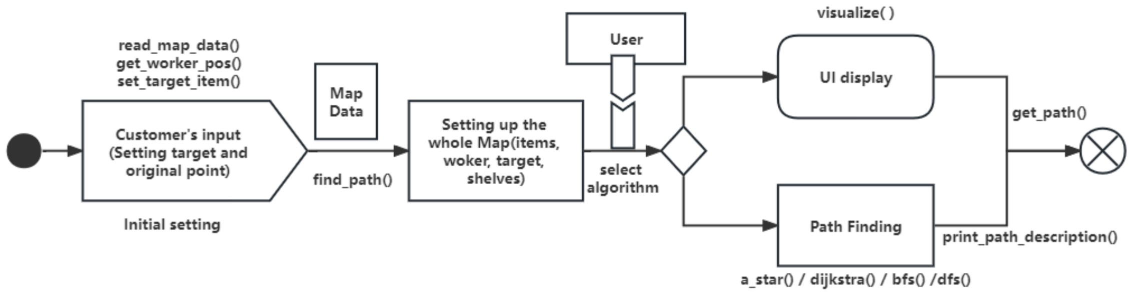


Figure 3: System Flow Chart

Initial settings flow

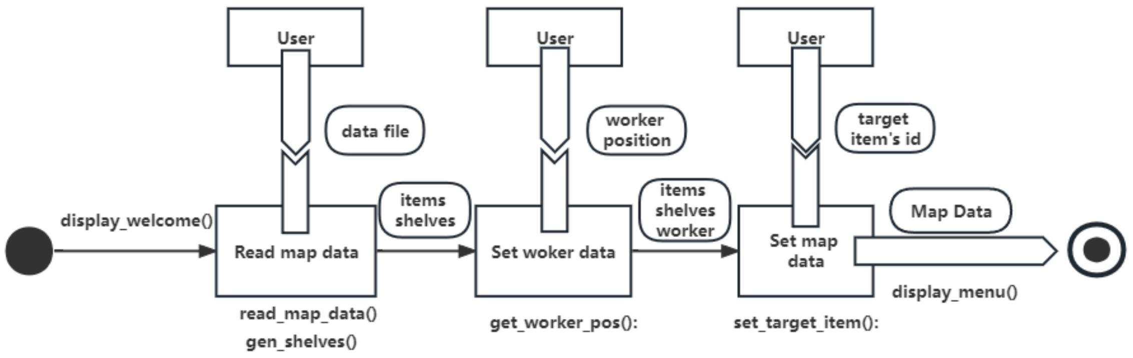


Figure 4: Setting Flow Chart

Installation

System Requirements

To use the program, the user need to have

- A working Python 3 environment installed on your computer. This can be downloaded and installed from the official Python website (https://www.python.org/downloads/).
- 2 GB of RAM
- 50 MB of free disk space

Setup & Config

- Open with `.exe` file
  Just double click it in the windows environment

- Install form the source code
  1. Download the program files from the source.
  2. Extract the files to a directory of your choice.
  3. Open a terminal or command prompt and navigate to the directory where the program files are located.
  4. Make sure that the warehouse product information file is under the same folder as the lazy_picker program, than run the program by typing the following command:

Shell

```shell
python lazy_picker.py
```

Uninstalling

- If our users' computer has a graphical user interface, simply click the `delete` button as you would when deleting a normal folder.
- For operating in command line, navigate to the directory where the `lazy_picker` folder is located and execute the following command:
  - For Linux user

    Shell

    ```shell
    rm lazy_picker
    ```

  - For Windows server

    Shell

    ```shell
    del lazy_picker
    ```

Documentation

Core Code

How the data are read and stored:

1. Calls the `read_map_data()` function to read the data from the file.
2. Calls the `get_worker_pos` function to get the worker's starting position.
3. Calls the `set_targets` function to get the target item.
4. Generates a `MapData` object with the data it got from the previous functions.

Python

```python
map_data = initialize_data()

def initialize_data():

    items, shelves = read_map_data('qvBox-warehouse-data-s23-v01.txt')
    worker = get_worker_pos()
    target = set_target_item(items)
    map_data = MapData(worker, shelves, items, target)

    return map_data

def read_map_data(filename):
    items = []
    with open(filename, 'r') as file:
        # Skip the first line of the file

        next(file)
        # Read the file line by line
```

```python
        for line in file:
            data = line.strip().split()
            item = Item(int(data[0]), float(data[1]), float(data[2]))

            items.append(item)

    shelves = gen_shelves(items)

    return items, shelves

def gen_shelves(items):

    temp = list(items)
    # Sort the items by their position
    temp.sort(key=lambda item: item.pos)
    index = 0

    pre = temp[0]
    shelf = Shelf(index, pre.pos[0], pre.pos[1])
    shelf.add_item(pre)
    shelves = [shelf]
    # Iterate through the sorted list of items
    for i in range(1, len(temp)):

        curr = temp[i]
        # If the current item has the same position as the previous item, add the item to the existing
shelf
        if curr.pos == pre.pos:
            i += 1
            shelf.add_item(curr)
        # Otherwise, create a new shelf and add the item to the shelf
        else:
            pre = curr
            index += 1

            shelf = Shelf(index, pre.pos[0], pre.pos[1])
            shelf.add_item(pre)
            shelves.append(shelf)

    return shelves

def get_worker_pos():
    while True:
        print()
        print("please enter the worker's starting position")
        is_default = input("Do you want to use the default position(0,0)? (y/n)")
        # If the user wants to use the default position, return a worker with the default position
        if is_default == "y":
            return Worker(0, 0)
        # If the user wants to enter a custom position, prompt the user to enter the x-coordinate and y-
coordinate
        elif is_default == "n":
            print("worker's x-coordinate is:")
            worker_x = input()
            # Check if the input is numeric
            if worker_x.isnumeric():
                # Convert the input to an integer
                converted_x = int(worker_x)
                print("worker's y-coordinate is:")
                worker_y = input()
                # Check if the input is numeric
                if worker_y.isnumeric():
                    # Convert the input to an integer
                    converted_y = int(worker_y)
                    # Create a worker with the given position
                    worker_pos = (converted_x, converted_y)
                    # Display the worker's position(Let the user confirm the position)
                    print("worker's position is:", worker_pos)
                    print("Press any key to continue")
                    input()
                    return Worker(worker_pos[0], worker_pos[1])  # return the worker's position
                else:
                    print("must be number")
            else:
                print("must be number")
        else:
            print("invalid input")


def set_target_item(items):
    print()
    print("Please enter the target item's id:")
    print("(If you forgot the id, you can press 'p' to see all the items' information)")
    target_id = input()
    if target_id.isnumeric():
        converted_id = int(target_id)
        for item in items:
            if item.item_id == converted_id:
                print("Target item is:", item)
                return item
            else:
                continue
        print()
        print("Cannot find item with that id, please try again!")

        return set_target_item(items)

    elif target_id == "p":
        peek_items(items)
        return set_target_item(items)
    else:
        print("Invalid input")
        return set_target_item(items)
```

Path finding algorithm framework

This frame work contains two major components: 1. Functional block, 2. Map helper functions

Functional Block

This block contains all the arguments and methods that are convenient to path-finding implementation.

It has five state, each state represent a different roles: GOAL, represents the the target item; NEW, represents a node that has not been visited; OPEN, represents a node in the open list(will be visited in the future); BLOCK, represents a node that is a shelf; PATH, represents a node that is in the path to the target node; 5: CLOSE, represents a node that has been visited; 6: START, represents the starting node(worker start position)

Python

```python
class NodeState(Enum):

    GOAL = 0
    NEW = 1
    OPEN = 2
    BLOCK = 3
    PATH = 4
    CLOSE = 5
    START = 6

class Block:
    def __init__(self, x, y):
        self.x = x
        self.y = y
        self.pos = (x, y)
        self.state = NodeState.NEW
        self.parent = None
        self.given_cost = 1
        self.heuristic = 0
        self.total_cost = 0
        self.final_cost = 0
        self.neighbours = []

    def cal_total_cost(self, new_total_cost):
        return new_total_cost + self.given_cost

    def set_parent(self, parent):
        self.parent = parent

    def get_parent(self):
        try:
            return self.parent
        except AttributeError:
            return None

    def cal_heuristic(self, target):

        self.heuristic = math.sqrt((self.x - target.x) ** 2 + (self.y - target.y) ** 2)
        return self.heuristic

    def get_final_cost(self, target):

        self.cal_heuristic(target)
        # self.heuristic = abs(self.x - target.x) + abs(self.y - target.y)
        self.final_cost = self.total_cost + FACTOR * self.heuristic
        return self.final_cost

    def is_next_to(self, next_block):

        if (self.x == next_block.x + 1 and self.y == next_block.y) or (
                # self.x == next_block.x - 1 and self.y == next_block.y) or (
                self.x == next_block.x and self.y == next_block.y + 1) or (
                self.x == next_block.x and self.y == next_block.y - 1):
            return True
        else:
            return False

    def __str__(self):
        return str(self.pos)
```

Map helper functions

We have encapsulated some necessary helper functions for algorithm implementation.

Python

```python
class Map:


    def get_neighbours(self, curr):
        """
        A function to get the neighbours of the current node.
        :param curr: The current node
        :return: A list of nodes representing the neighbours of the current node.
        """

        directions = [(0, 1), (0, -1), (1, 0), (-1, 0)]
        neighbours = []

        # Check all the neighbours of the current node
        # If the neighbour is not a block(Shelf), then add it to the list
        for x_diff, y_diff in directions:
            x = curr.x + x_diff
            y = curr.y + y_diff
            if 0 <= x < self.map_row and 0 <= y < self.map_col:
                if self.grid[x][y].state != NodeState.BLOCK:
                    neighbours.append(self.grid[x][y])
        return neighbours



    def get_path(self, curr):
        self.has_path = True
        curr.state = NodeState.GOAL
        self.path.append(curr)
        # Start from the target and reach the start node recursively
        while curr.parent is not None:
            curr = curr.parent
            curr.state = NodeState.PATH
            self.path.append(curr)

        curr.state = NodeState.START

        # Reverse the path to get the correct order
        self.path.reverse()
        self.visualize(False)
        self.print_path_description()
```

A Star Search

A star search will keep iterating until it finds a path from the worker to the target. In each iteration, it will pick the first node from the open list, which is the node with the lowest final cost, and call the function `astar_iterate()` . The function will check all the neighbours of the current node and do the following:

1. If the neighbour is next to the target node, then the path is found.
2. If the neighbour is a new node, then add it to the open list.
3. If the neighbour is an open node, then check if the new final cost is lower than the current final cost.
4. If the neighbour is a closed node, then check if the new final cost is lower than the current final cost.
5. When all the neighbours are checked, add the current node to the closed list.
6. Sort the open list by the final cost of the nodes.
   (final cost = total cost + heuristic cost, total cost = given cost + parent's total cost)

Python

```
def a_star(self):

        # Initialize the open and closed list
        curr = self.start_block
        self.open_list = [curr]
        self.closed_list = []

        while not self.has_path:  # Keep iterating until a path is found
            self.visualize()
            self.iteration += 1  # Record the number of iterations
            curr = self.open_list.pop(0)  # Pick the first node from the open list
            self.astar_iterate(curr)

        return self.path


    def astar_iterate(self, curr):
        for neighbour in self.get_neighbours(curr):
            state = neighbour.state
            new_cost = curr.total_cost + neighbour.given_cost
            new_final_cost = new_cost + FACTOR * neighbour.cal_heuristic(self.target_block)

            if state == NodeState.GOAL:
                # If the neighbour is next to the target node, then the path is found
                neighbour.parent = curr
                self.get_path(neighbour)
                return self.path

            elif state == NodeState.NEW:
                # If the neighbour is a new node, then add it to the open list
                neighbour.state = NodeState.OPEN
                neighbour.parent = curr
                neighbour.total_cost = new_cost
                neighbour.final_cost = new_final_cost
                self.open_list.append(neighbour)
                continue

            elif state == NodeState.OPEN:
                # If the neighbour is an open node, then check if the new cost is lower than the current
cost
                # If the new cost is lower, then update the neighbour's cost and parent
                if new_final_cost < neighbour.final_cost:
                    neighbour.total_cost = new_cost
                    neighbour.final_cost = new_final_cost
                    neighbour.parent = curr
                    continue

            elif state == NodeState.CLOSE:
                # If the neighbour is a closed node, then check if the new cost is lower than the current
cost
                # If the new cost is lower, then remove the neighbour from the closed list and add it to
the open list
                if new_final_cost < neighbour.final_cost:
                    neighbour.total_cost = new_cost
                    neighbour.final_cost = new_final_cost
                    neighbour.parent = curr

                    # Also update the neighbour's state
                    neighbour.state = NodeState.OPEN
                    # Also update the neighbour's cost and parent
                    self.closed_list.remove(neighbour)
                    self.open_list.append(neighbour)
                    continue
        # Add the current node to the closed list and set its state to closed
        self.closed_list.append(curr)
        curr.state = NodeState.CLOSE
        # Sort the open list by the final cost
        self.open_list.sort(key=lambda x: x.final_cost)
```

Branch and bound

Solve the problem with branch and bound algorithm

1. Select a random vertex as the root
2. set the other entrance with the same target to infinity
3. Reduce the matrix
4. Put the root into the priority queue
5. While the priority queue is not empty:
    1. 4.1 Get the node with the smallest cost
    2. 4.2 If the node is a leaf node

        return the result
    3. Else:

        Keep reducing the matrix and put the node into the priority queue*

Select a random vertex as the root

Python

```
class Branch_n_Bound:
    """
    Branch and bound algorithm to find the shortest path
```

```python
    """

    def __init__(self, adjacent_map, vertexes, size, limit_time=True):

        self.adjacent_map = adjacent_map
        self.vertexes = vertexes
        self.nums_of_vertexes = len(vertexes)
        self.matrix = None
        self.limit_time = limit_time
        self.size = size
        self.pq = PriorityQueue()
        self.same_target = {}
        self.result = []
        # Set up the index mapping to the same target vertexes
        for i in range(self.nums_of_vertexes):
            curr_vertex = set()

            for j in range(self.nums_of_vertexes):
                if i != j and vertexes[i].is_entrance_of == vertexes[j].is_entrance_of:
                    curr_vertex.add(j)
            self.same_target[i] = curr_vertex
            # print(i, " ", curr_vertex)

    def set_matrix(self):
        """
        Iterate through the adjacent map and set the matrix
        f the adjacent map is None, set the matrix to infinity
        Else set the matrix to the weight of the edge
        """

        self.matrix = [[0 for i in range(self.nums_of_vertexes)] for j in range(self.nums_of_vertexes)]
        for i in range(len(self.matrix)):
            for j in range(len(self.matrix[0])):
                if self.adjacent_map[i][j] is None:

                    self.matrix[i][j] = float("inf")
                else:
                    self.matrix[i][j] = self.adjacent_map[i][j].weight
        print("Original matrix: ")
        print_matrix(self.matrix)

    def solve(self):
        """
        Solve the problem with branch and bound algorithm
        1. Select a random vertex as the root
        2. set the other entrance with the same target to infinity
        2. Reduce the matrix
        3. Put the root into the priority queue
        4. While the priority queue is not empty:
            4.1 Get the node with the smallest cost
            4.2 If the node is a leaf node:
                return the result
            4.3 Else:
                Keep reducing the matrix and put the node into the priority queue
        """
        start_time = time.time()
        if self.nums_of_vertexes > 1 and not self.has_destination:
            random_index = random.randint(0, self.nums_of_vertexes - 1)
        else:
            # print((len(self.vertexes)))
            random_index = 0
        # random_vertexes = []
        # random_index = 0
        random_vertex = self.vertexes[random_index]
        root = PathTreeNode(random_vertex, self.nums_of_vertexes)
        start_matrix = copy.deepcopy(self.matrix)

        self.reduce_matrix(start_matrix, root)
        self.pq.put(root)

        for other_index in self.same_target[random_index]:
            root = PathTreeNode(self.vertexes[other_index], self.nums_of_vertexes)
            start_matrix = copy.deepcopy(self.matrix)

            self.reduce_matrix(start_matrix, root)
            self.pq.put(root)
        print("Initial end")
        iterate = 0
        while not self.pq.empty():
            iterate += 1
            # print("pq size: ", self.pq.qsize())

            curr_node = self.pq.get()

            # print_matrix(curr_node.matrix)
            curr_time = time.time()

            # if all(curr_node.visited):
            if curr_node.len == self.size:
                # print("Success!")
                # print("Path: ")
                # for vertex in curr_node.path:
                #     print(vertex.block, end=" ")
                print("start_index: ", self.vertexes[random_index].block)

                return self.generate_result(curr_node)

            elif self.has_destination and curr_node.len == self.size - 1:

                return self.generate_result_with_destination(curr_node, self.destination)

            # set the limit time, which default is 60 seconds, stop the algorithm and return a result
            if self.is_limit_time and self.limit_time and curr_time - start_time > self.limit_time:
                print("Time is up! Returning what we current got")
                max_node = curr_node
                while not self.pq.empty():
                    temp_node = self.pq.get()
                    if max_node.len < temp_node.len:
                        max_node = temp_node

                for index in range(self.nums_of_vertexes):
                    if not max_node.visited[index]:
                        next_vertex = self.vertexes[index]
                        new_matrix = copy.deepcopy(max_node.matrix)
                        next_node = PathTreeNode(next_vertex, self.nums_of_vertexes, max_node.cost,
max_node)
                        self.reduce_matrix(new_matrix, next_node)
                        max_node = next_node

                return self.generate_result(max_node)

            for index in range(self.nums_of_vertexes):
                if not curr_node.visited[index]:
                    next_vertex = self.vertexes[index]
                    new_matrix = copy.deepcopy(curr_node.matrix)

                    next_node = PathTreeNode(next_vertex, self.nums_of_vertexes, curr_node.cost,
curr_node)
                    self.reduce_matrix(new_matrix, next_node)

                    self.pq.put(next_node)

    def reduce_matrix(self, matrix, curr_tree_node):
        """
        Reduce Matrix (rows & columns to at least one 0) = reduced cost
        1. Set the vertex of same target nodes' row and column to infinity
        2. If it is a root node, reduce the matrix to get the main reduced cost
        3. Else, the main reduced cost is the parent's main reduced cost
            3.1 get the matrix[pre_index][curr_index] the value as reduction
            3.2 reduce the matrix to get the reduced cost
            3.3 set the current node's cost to the parent's cost + reduction + reduced cost
        """

        # input("Press Enter to continue...")
        curr_vertex = curr_tree_node.vertex
        curr_index = curr_vertex.index
        curr_tree_node.visited[curr_index] = True
        self.ignore_same_target_entrance(curr_index, curr_tree_node, matrix)
        # if curr_tree_node.parent is None: it is a root node
        if curr_tree_node.parent is None:
            reduce_cost, matrix = self.reduce_row_n_col(matrix)
            curr_tree_node.cost = reduce_cost
            print_matrix(matrix)
            # ignore the entrance with same target as current vertex

        else:
            # if it is not the root node

            parent = curr_tree_node.parent
            pre_vertex = parent.vertex
            pre_index = pre_vertex.index
            first_index = curr_tree_node.first_index

            main_reduce_cost = parent.cost
            reduction = matrix[pre_index][curr_index]


            # delete the row of the pre_vertex
            for j in range(len(matrix)):
                matrix[pre_index][j] = float("inf")
            # delete the column of the curr_vertex
            for i in range(len(matrix)):
                matrix[i][curr_index] = float("inf")

            matrix[curr_index][first_index] = float("inf")

            reduce_cost, matrix = self.reduce_row_n_col(matrix)

            curr_tree_node.cost += reduction
            curr_tree_node.cost += reduce_cost


        curr_tree_node.matrix = matrix
        print("------------------------------------------")

    def ignore_same_target_entrance(self, curr_index, curr_tree_node, matrix):
        for index in self.same_target[curr_index]:
            curr_tree_node.visited[index] = True
            # set the row and column of the same target to inf
            for i in range(len(matrix)):
                matrix[i][index] = float("inf")
            for j in range(len(matrix)):
                matrix[index][j] = float("inf")

    def reduce_row_n_col(self, matrix):


        reduced_cost = 0
        for i in range(len(matrix)):

            row = matrix[i]
            # get the min value of the row
            min_value = min(row)
            print(min_value, end=" ")
            if min_value == float("inf") or min_value == 0:
                continue

            else:
                # reduce each element of the row by the min value
                for j in range(len(row)):
                    if row[j] != float("inf") and row[j] != 0:
                        row[j] -= min_value
                matrix[i] = row
                # get the total cost of each row
                reduced_cost += min_value


        for j in range(len(matrix[0])):
            column = [row[j] for row in matrix]
            min_value = min(column)
            print(min_value, end=" ")
            if min_value == float("inf") or min_value == 0:
                continue
            else:
                # reduce each element of the column by the min value
                for i in range(len(column)):
                    if column[i] != float("inf") and column[i] != 0:
                        column[i] -= min_value
                for i in range(len(matrix)):
                    matrix[i][j] = column[i]
                reduced_cost += min_value

        return reduced_cost, matrix

    def generate_result(self, curr_node):
        """
        Generate the result
        """
        print("curr_node: ", curr_node.vertex.block)

        path_vertexes = curr_node.path
        final_path = []
        final_path_description = []
        total_cost = 0
        start_index = 0
        for i in range(len(path_vertexes)):
            if path_vertexes[i].index == 0:
                start_index = i
                # print(path_vertexes[i].block)
                break
        curr_index = start_index
        for i in range(start_index, len(path_vertexes) - 1):
            curr_index = i + 1
            self.result.append(self.adjacent_map[path_vertexes[i].index][path_vertexes[curr_index].index])

        self.result.append(self.adjacent_map[path_vertexes[curr_index].index][path_vertexes[0].index])
        for i in range(start_index):
            self.result.append(self.adjacent_map[path_vertexes[i].index][path_vertexes[i + 1].index])

        for edge in self.result:
            edge.path[0].state = NodeState.STOP
            total_cost += edge.weight
            for i in range(1, len(edge.path)):
                if edge.path[i].state == NodeState.STOP:
                    continue

                edge.path[i].state = NodeState.PATH

            final_path += edge.path
            final_path_description.append(edge.path_description)
        final_path[-1].state = NodeState.START
        return final_path, final_path_description, total_cost
```

Guide: Add your Own Algorithm

We have created a highly extensible algorithm framework that allows you to add your own algorithms. Here is a step-by-step guide on how to add your own algorithm:

1. Define your algorithm's name and the corresponding function.

Python

~~~python
class Map:
    ``` ... ```

    def {your_algo}(self):
        # Initialize the open and closed list
        curr = self.start_block
        self.open_list = [curr]
        self.closed_list = []

        while not self.has_path:  # Keep iterating until a path is found
            self.visualize()
            self.iteration += 1  # Record the number of iterations
~~~

```
            curr = self.open_list.pop(0)  # Pick the first node from the open list
            self.{your_algo}_iterate(curr)
```

2. Based on your algorithm's logic, add the necessary code to the relevant parts of the framework. For example, you could modify the sorting logic of the open list to ensure that you obtain the desired block each time.

Python

~~~python
class Map:
    ``` ... ```

    def {your_algo}(self):
        ``` ... ```

    def {your_algo}_iterate(self, curr):

        for neighbour in self.get_neighbours(curr):
            state = neighbour.state
            # Update logic

            if state == NodeState.GOAL:
                # If the neighbour is next to the target node, then the path is found
                neighbour.parent = curr
                self.get_path(neighbour)
                return self.path

            elif state == NodeState.NEW:
                # If the neighbour is a new node, then add it to the open list
                neighbour.state = NodeState.OPEN
                neighbour.parent = curr

                # Update base on your algorithm logic

                self.open_list.append(neighbour)
                continue

            elif state == NodeState.OPEN:
                # Update base on your algorithm logic
                continue

            elif state == NodeState.CLOSE:
                # Update base on your algorithm logic
                continue

        # Add the current node to the closed list and set its state to closed
        self.closed_list.append(curr)
        curr.state = NodeState.CLOSE
        # Sort the open list base on your algorithm logic
        self.open_list.sort(key=lambda x: x.{Your algo logic})
~~~

Timeline and plan

Major Milestones

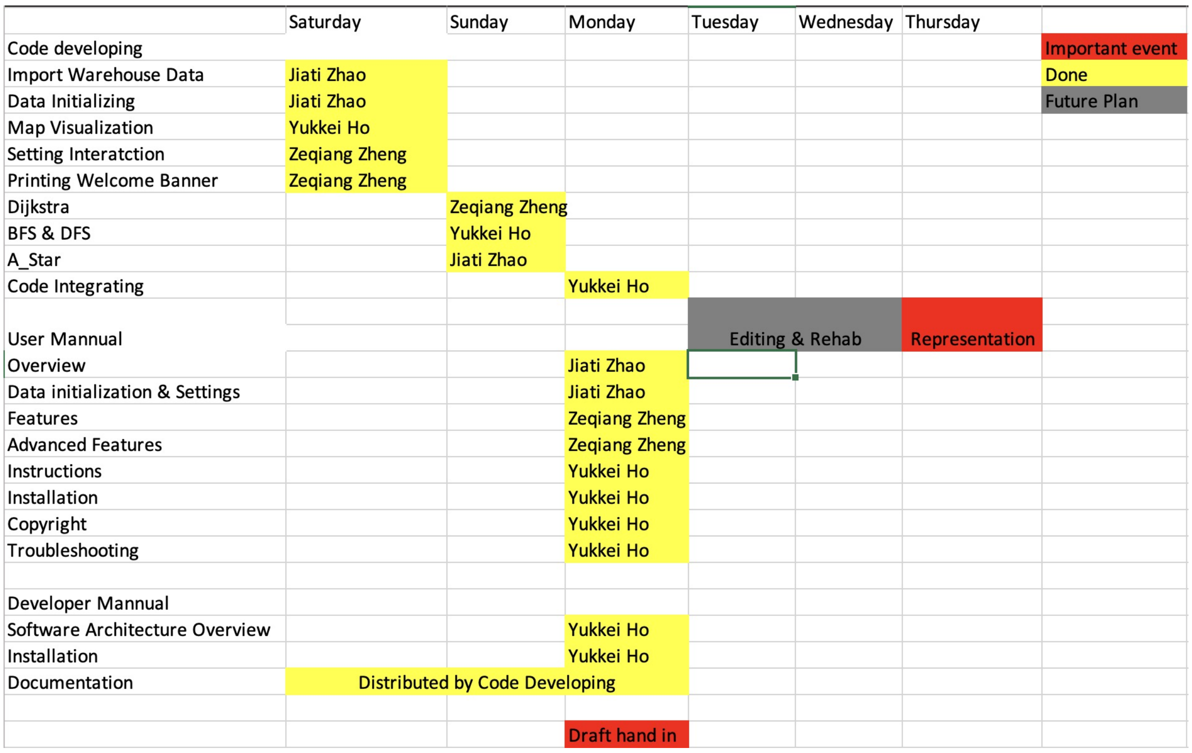
- alpha

Achievement	Owner	Status	End date	Obstacles
Agree on the software architecture	whole Team	Completed	Thu, May 02	Brainstorm
Basic Feature fulfilled	whole Team	In Progress	SAT,May 06	Merge Code
Pass testing	whole Team	Not yet start	Mon, May 08	Bugs fixing

- beta v1:
    - Implement the travel - sales man extension(Branch and bound)
    - Enhance error handling and UI elements
- beta v2:
    - Add nearest neighbors' algorithm
    - Trying to fix the bugs occur in the multiple access point case with branch and bound
    - Allow user to upload their own file through file picker.

Team member responsibilities

- alpha



	Saturday	Sunday	Monday	Tuesday	Wednesday	Thursday	
Code developing							Important event
Import Warehouse Data	Jiati Zhao						Done
Data Initializing	Jiati Zhao						Future Plan
Map Visualization	Yukkei Ho						
Setting Interatction	Zeqiang Zheng						
Printing Welcome Banner	Zeqiang Zheng						
Dijkstra		Zeqiang Zheng					
BFS & DFS		Yukkei Ho					
A_Star		Jiati Zhao					
Code Integrating			Yukkei Ho				
User Mannual				Editing & Rehab		Representation	
Overview			Jiati Zhao				
Data initialization & Settings			Jiati Zhao				
Features			Zeqiang Zheng				
Advanced Features			Zeqiang Zheng				
Instructions			Yukkei Ho				
Installation			Yukkei Ho				
Copyright			Yukkei Ho				
Troubleshooting			Yukkei Ho				
Developer Mannual							
Software Architecture Overview			Yukkei Ho				
Installation			Yukkei Ho				
Documentation	Distributed by Code Developing						
			Draft hand in				

Figure 4: develope plan

Copyright

Error Messages and Handling

This section provides solutions for common issues that you may encounter while using our program.

Error handling 1: Input Error handling

The application has built-in error handling to prevent the user from entering invalid input. If invalid input is entered, the application will provide an error message indicating the issue.

Error handling 2: File not find error handling

If there is an issue with the input file, the program should not crash

Maintain Repo Address:

https://github.com/yuk-kei/lazy-picker-beta