# **Software Architecture Document**
for
Warehouse Product Locating and Routing System (WPLRS)
Release

Prepared by Ahmet Aksakal(aaksakal@uci.edu)
Hao Chen(haoc19@uci.edu)
Kelvin Phan(kelvinhp@uci.edu)
Ziwen Ning(ziwenn1@uci.edu)

University of California, Irvine
EECS 221
Advanced Application of Algorithms

7 June 2019

Contents

Version History

Revision	Date	Author(s)	Description
1.0	12.04.2019	A. Aksakal, H. Chen, K. Phan, Z. Ning	Initial version
1.5	05.03.2019	A. Aksakal, H. Chen, K. Phan, Z. Ning	Alpha/Beta Pre-Check Version
2.0	05.10.2019	A. Aksakal, H. Chen, K. Phan, Z. Ning	Beta Version • *Alpha to Beta Improvements:* – Map re-oriented to follow absolute North with origin (0,0) at bottom-left corner – User directions are more readable – Output errors resolved – Binary now available for installation • *Beta Features:* – Supports importing order lists of size 1,5, 10, and 15 – Dynamic start location

2.5	05.24.2019	A. Aksakal, H. Chen, K. Phan, Z. Ning	Beta v2 Version • *Beta to Beta v2 Improvements:* – Made path directions consistent and more user friendly – Changed empty spaces character from '0' to '.' for improved visibility – Distances are only calculated once • *Beta Features:* – Better than brute force performance – Supports manual input of order list – Supports loading input file into order list ∗ User can request next unfulfilled order ∗ User can request specific order number/line – Dynamic start and end locations
Release	06.07.2019	A. Aksakal, H. Chen, K. Phan, Z. Ning	Release Version • *Beta to Release Improvements:* – Improved error handling with exceptions for unloaded data, unavailable menu options, etc. – Test document includes input cases and covers wider range for chosen genetic algorithm • *Release Features:* – Error handling without crashing, alerts user instead – Parallelized genetic algorithm for better performance – Timeout feature that limits time taken to compute path and calculate distances

Chapter 1

Front Matter

1.1 Glossary

- *Warehouse Terminology*:
  - **Item/Product:** a warehouse object that has attributes such as ID, name,quantity, and location inside the warehouse.
  - **Location:** an (x,y) coordinate pair representation of where a warehouse object based on a grid representation of the warehouse
  - **Order:** a compiled list of items/products that need to be retrieved from the warehouse to be sent out
- *Application Terminology*:
  - *Classes/Modules:*
    - *Product:* class that represents items with ProductID, xLocation, yLocation, AccessN, AccessW, AccessS, AccessE
    - *Inventory:* class responsible for tracking Products and determine paths based on product(s) inputted.
    - *ComputePath:* class responsible for calculating paths for orders using selected algorithms
    - *Order:* class responsible for importing orders and adding products to existing lists
    - *Order List:* class responsible for list of orders and tracking fulfilled vs. unfulfilled orders
    - *Path Finder:* module responsible for converting algorithm output to human readable directions

Chapter 2

Overview of System

2.1 Data Type and Structures

Basic Data Structure Definitions:

- *Array/List:* list of objects where each object has an index and associated data value
- *Dictionary:* group of objects where each object is made up of a key-value pair
- *Graph:* group of objects where each object is a node or vertex that is connected to other nodes via edges

Implemented Data Structures:

- *Inventory:* dictionary of items/products that tracks the location and availability, and accessibility of all products.
- *Distances:* graph of items/products that represents information about the distance between items.
- *Orders:* list of orders that tracks the products requested per order and the each order's fulfillment status
- *Path:* ordered list of products that represents the pickup order of items in an order.
- *Distances:* ordered list that records the cost of each pickup for an item in a Path

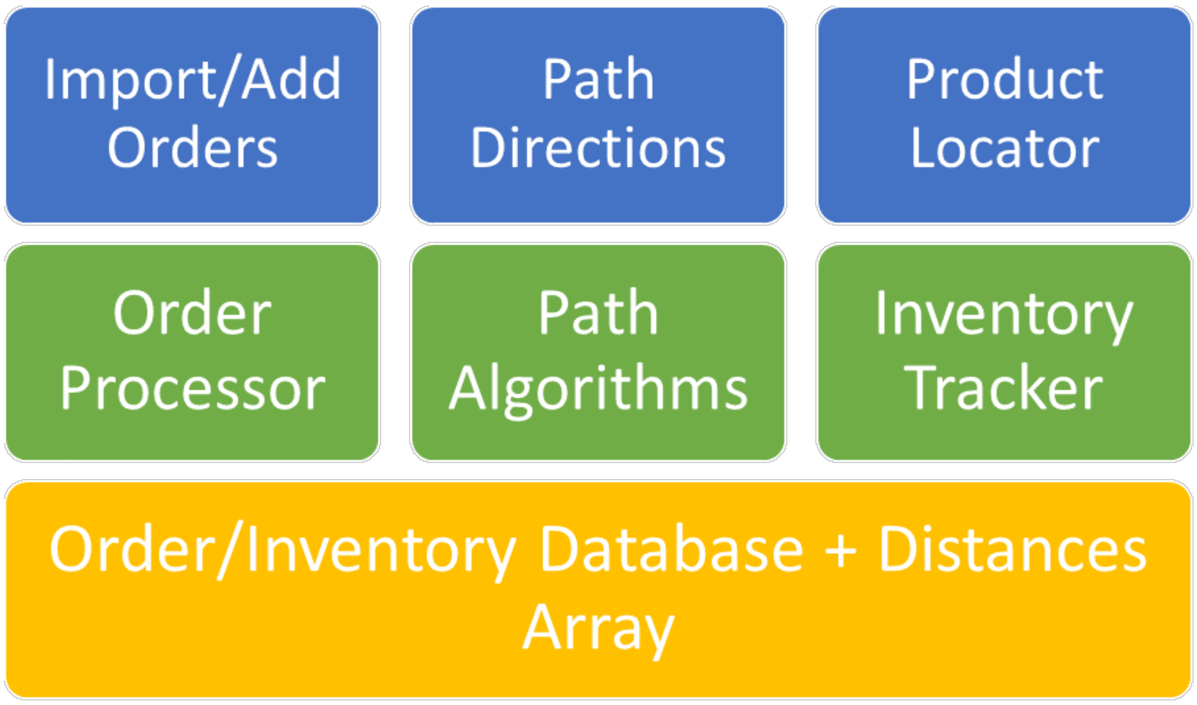

Figure 2.1: Displays key modules organized from top to bottom for front end to back end

2.2 Software Components

- **Front End(Blue):**
  - *Menu:* interface that allows user to select importing inventory, displaying the map, showing the location of a product, computing path to a product, changing the start point, importing an order list, adding an order to a list, and calculating path for order.
  - *Order Creation/Import:* interface that allows user to import orders and add to import lists. Also allows user to import lists of orders and get specific orders or next unfulfilled
  - *Map Display:* interface that displays the current location of all products in the inventory
- **Algorithms and Functions(Green):**
  - *Order Processor:* parses information from “Order Creation/Import” to import new orders or to process/prepare order to path calculation.
  - *Path Algorithms:* takes information from order processor to calculate path for most efficient product pickup and outputs list of directions
  - *Inventory Tracker:* tracks results of order processing in order to keep information about available products updated in “Inventory Database”. This information is displayed in the “Product Directory Interface”
- **Back End(Orange):**
  - *Order/Inventory Database + Distances Array:* stores information about orders including relevant products. stores information about products including availability and accessibility. Distances array stores data about distance between products in a 2D array. Together, this information is used by other layers to perform necessary functions

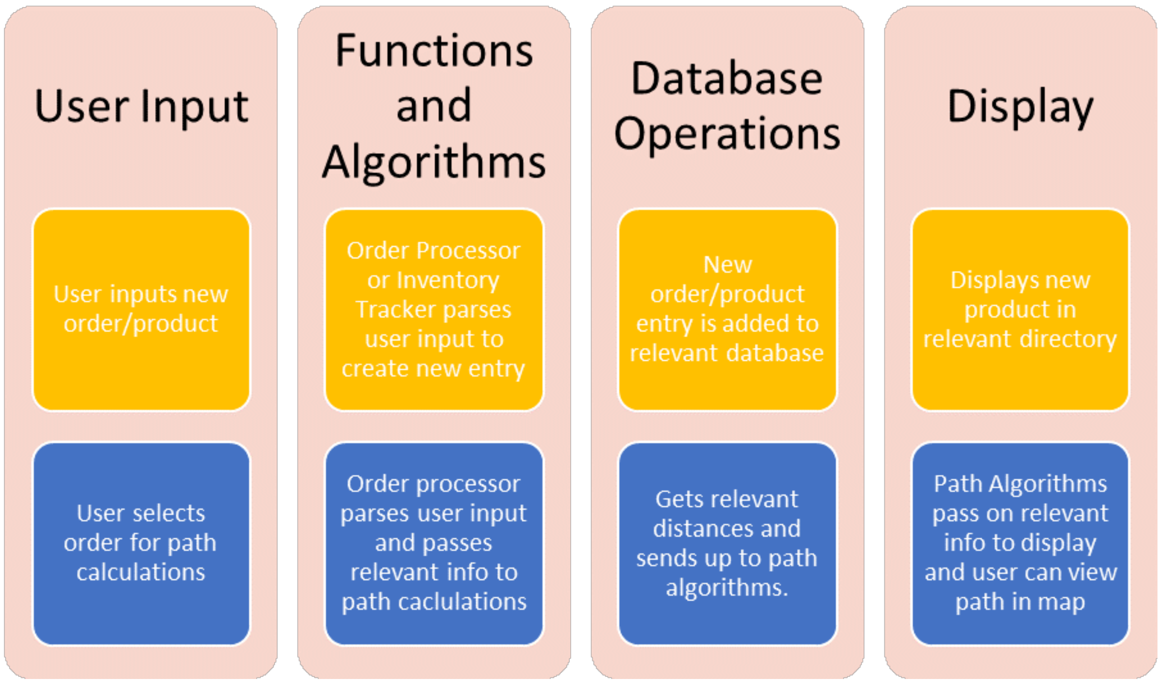
User Input
User inputs new order/product
User selects order for path calculations
Functions and Algorithms
Order Processor or Inventory Tracker parses user input to create new entry
Order processor parses user input and passes relevant info to path caclulations
Database Operations
New order/product entry is added to relevant database
Gets relevant distances and sends up to path algorithms.
Display
Displays new product in relevant directory
Path Algorithms pass on relevant info to display and user can view path in map

Figure 2.2: Displays 2 typical program flows for order/product creation(Gold) and path calculation(Blue)

2.3 Modules API

- **Product:**
  - *get“‘Placeholder”’():* returns data related to text replacing “‘Placeholder”’
- **Order:**
  - *addOrder():* add order to list
  - *importOrder():* read in file with list of orders
- **Distances:**
  - *calculateDistances():* calculates distance between all products in inventory to form weighted graph
- **ComputePath:**
  - *run():* uses distances value to run brute force, genetic, dynamic programming, or greedy algorithm to find shortest path
- **WPLRS:**
  - *“‘Placeholder”’isEmpty():* returns if “‘Placeholder”’ is empty
  - *set“‘Placeholder”’():* set “‘Placeholder”’ to value including setting start point, end point
  - *get“‘Placeholder”’():* returns“‘Placeholder”’ including next order, specific order, and direction
  - *add/import “‘Placeholder”’():* adds new “‘Placeholder”’ object to relevant class
  - *display“‘Placeholder”’():* display“‘Placeholder”’ data according to UI such as map and directions

```
class WPLRS:

    def __init__(self):
        self.inventory = None
        self.order = Order()
        self.start_point = (0, 0)
        self.end_point = (0, 0)
        self.order_list = OrderList()

    def importInventory(self, file_name):
        self.inventory = Inventory(start_point=self.start_point,
                                   end_point=self.end_point)
        self.inventory.importItems(file_name)
        print("\nDefault start point is (0, 0).")
        print("Default end point is (0, 0).\n")
        self.inventory.calculateDistances()

    def inventoryIsEmpty(self):
        return len(self.inventory.inventory) == 0

    def displayInventory(self):
        self.inventory.displayMap()

    def displayPathInventory(self, paths, sequence):
        self.inventory.displayPathMap(paths, sequence)

    def importOrder(self, file_name):
        self.order.importOrder(file_name)

    def orderIsEmpty(self):
        return len(self.order.id_list) == 0

    @timeout_decorator.timeout(25, timeout_exception=StopIteration)
    def computePath(self, order = []):
        if not self.inventory:
            print("\n\nInventory has not been imported yet.\n\n")
            return None
        computer = ComputePath(self.inventory.distance_array,
                               self.inventory.ID2Index)
        if len(order) == 0:
            order = self.order.id_list
        if len(order) < 51:
            algo = 'GA'
        else:
            algo = 'GREEDY'
        sequence = computer.run(order, algo)
        return sequence

    def getPathBySequence(self, sequence):
        curSource = -1 # Start point
        paths = []
        for productId in sequence:
            if curSource == -1:
                path = self.inventory.getPathToProduct(productId)
                curSource = productId
            else:
                path = self.inventory.getPathBetweenProduct(curSource, productId)
                curSource = productId
            paths.append(path)
        # From last item to end point:
        end_point_ID = '-1'
        path = self.inventory.getPathBetweenProduct(curSource, end_point_ID)
        paths.append(path)
        return paths

    def getPathToProduct(self, productID):
```

```python
        return self.inventory.getPathToProduct(productID)

    def getLocationByID(self, productID):
        return self.inventory.getLocationByID(productID)

    def setStartPoint(self, point):
        if point == self.inventory.start_point:
            print("\n\nThe given point is already the start point.\n\n")
            return
        if point in self.inventory.shelves:
            print("\n\nStart point should not be on a shelf.\n\n")
            return
        self.start_point = point
        if self.inventory:
            self.inventory.setStartPoint(point)

    def setEndPoint(self, point):
        if point == self.inventory.end_point:
            print("\n\nThe given point is already the end point.\n\n")
            return
        if point in self.inventory.shelves:
            print("\n\nEnd point should not be on a shelf.\n\n")
            return
        self.end_point = point
        if self.inventory:
            self.inventory.setEndPoint(point)

    def addOrder(self, productID):
        if productID in self.inventory.inventory:
            self.order.addOrder(productID)
        else:
            print("\n\nGiven ID is not in the inventory.")

    def paths2Instrs(self, paths, order = []):
        '''Get the user friendly instruction from a path list'''
        instructions = []
        if len(order) == 0:
            item_ids = self.order.id_list
        else:
            item_ids = order
        # Dictionary indicating where to turn
        direct2turn = {('North', 'East'): 'Right', ('North', 'South'): 'Back', ('North', 'West'): 'Left',
                    ('East', 'North'): 'Left', ('East', 'South'): 'Right', ('East', 'West'): 'Back',
                    ('South', 'North'): 'Back', ('South', 'East'): 'Left', ('South', 'West'): 'Right',
                    ('West', 'North'): 'Right', ('West', 'East'): 'Back', ('West', 'South'): 'Left'}
        # Each path has a bunch on coordinates. Starts with the first coordinate.
        coord = paths[0][0] # Start point
        next_coord = paths[0][1]
        direction = 'North'
        direction = self.getDirection(coord, next_coord, direction)
        instructions.append("Start at {}".format(coord))
        step_count = 0
        for idx, path in enumerate(paths):
            if len(path) == 1:
                instructions.append("Pick Up Item: {} at {}".format(item_ids[idx], next_coord))
                continue
            for i, coord in enumerate(path[:-1]):
                next_coord = path[i+1]
                next_direct = self.getDirection(coord, next_coord, direction)
                if direction == next_direct:
                    step_count += 1
                    continue
                else:
                    turn_direct = direct2turn[(direction, next_direct)]
                    instructions.append('Move Towards {}, For {} Steps, Until '.format(direction, step_count) +
                                        'You Reach Point {} and Turn {} to {} Direction.'.format(coord, turn_direct, next_direct))
                    step_count = 1 # Reset step count
                    direction = next_direct
            if idx == len(paths) - 1:
                instructions.append("Move Towards {}, For {} Steps and End Your Tour at {}".format(direction, step_count, next_coord))
            else:
                instructions.append("Move Towards {}, For {} Steps, and Pick Up Item: {} at {}".format(direction, step_count, item_ids[idx], next_coord))
                step_count = 0
        return instructions

    def getDirection(self, first_pt, next_pt, direction):
        if first_pt == next_pt:
            return direction
        is_north = next_pt == (first_pt[0], first_pt[1] + 1)
        is_east = next_pt == (first_pt[0] + 1, first_pt[1])
        is_south = next_pt == (first_pt[0], first_pt[1] - 1)
        is_west = next_pt == (first_pt[0] - 1, first_pt[1])
        if is_north:
            return 'North'
        elif is_east:
            return 'East'
        elif is_south:
            return 'South'
        elif is_west:
            return 'West'
        else:
            raise ValueError("The Given Two Points Are NOT One Step From One Another!!")

    def loadOrderList(self, file_name):
        self.order_list.importOrderList(file_name)

    def getNextOrder(self):
        return self.order_list.nextOrder()

    def getSpecificOrder(self, index):
        return self.order_list.specificOrder(index)

    def inputOrder(self, order_string):
        self.order_list.inputOrder(order_string)

    def orderListIsEmpty(self):
        return len(self.order_list.order_list) == 0
```

Chapter 3

Installation

3.1 System Requirements

- **Windows or Mac or Linux**

3.2 Setup and Configuration

1. Open wlprs.zip included in submission
2. Extract wplrs.zip and change directory to wlprs/bin/dist/interactive
3. Run “interactive.exe”

3.3 Uninstalling

1. Remove wplrs and wplrs.zip directory

Chapter 4

Packages, Modules, Interfaces

4.1 Packages

4.2 Data Structure/Class Details

- **Interactive Class:**
  - *Attributes:*
    - N/A
  - *Methods:*
    - *run():* Prints menu and parses user input. Handles errors at the front end by only enabling certain options once proper steps are followed. Also manages timeout for long duration functions
  - *Critical Code:*

```python
class Interactive:

    def run(self):
        self.printTitle()
        imported = 0
        while(1):
            self.printOption()
            inp = input("Please select an option: ")
            print('')
            if inp == '1': # Import the inventory.
                inventory_file_name = input("Please enter the inventory file you want to load: ")
                try:
                    self.program.importInventory(inventory_file_name)
                except Exception:
                    print("Cannot find the file\n")
                    time.sleep(1)
                    continue
                self.inventory_file_label = '<' + inventory_file_name + '>'
                print('\n')
                time.sleep(1)
                imported = 1
            elif inp == '2': # Display the Map
                if not imported:
                    print("Please import the inventory with option [1] first.")
                    time.sleep(1)
                    continue
                self.program.displayInventory()
                print('\n')
            elif inp == '3': # Learn the location of a product.
                if not imported:
                    print("Please import the inventory with option [1] first.")
                    time.sleep(1)
                    continue
                productID = input('Please enter a product ID: ')
                (x, y) = self.program.getLocationByID(productID)
                if x == -1 and y == -1:
                    print("\n\nGiven product ID is not in the inventory")
                else:
                    print("\nPosition is (" + str(x) + ", " + str(y) + ")")
                print('\n')
                time.sleep(1)
            elif inp == '4': # Compute the path to a product.
                if not imported:
                    print("Please import the inventory with option [1] first.")
                    time.sleep(1)
                    continue
                productID = input('Please enter a product ID: ')

                (x, y) = self.program.getLocationByID(productID)
                if x == -1 and y == -1:
                    print("\n\nGiven product ID is not in the inventory")
                    print('\n')
                    time.sleep(1)
                    continue

                path = self.program.getPathToProduct(productID)
                paths = []
                paths.append(path)
                print("Please follow this instruction to get the product:\n")
                instrs = self.program.paths2Instrs(paths)
                for instr in instrs:
                    print(instr)
                    time.sleep(0.25)
                time.sleep(1)
            elif inp == '5': # Change start point.
                if not imported:
                    print("Please import the inventory with option [1] first.")
                    time.sleep(1)
                    continue
                x = int(input("Please enter x coordinate: "))
                y = int(input("Please enter y coordinate: "))
                point = (x, y)
                self.program.setStartPoint(point)
                print('\n')
                time.sleep(1)
            elif inp == '6': # Change end point.
                if not imported:
                    print("Please import the inventory with option [1] first.")
                    time.sleep(1)
                    continue
                x = int(input("Please enter x coordinate: "))
                y = int(input("Please enter y coordinate: "))
                point = (x, y)
                self.program.setEndPoint(point)
                print('\n')
                time.sleep(1)
            elif inp == '7': # Import Order List.
```

```python
                if not imported:
                    print("Please import the inventory with option [1] first.")
                    time.sleep(1)
                    continue
                list_name = input("Please enter the file you want to import: ")
                try:
                    self.program.importOrder(list_name)
                except Exception:
                    print("Cannot find the file\n")
                    time.sleep(1)
                    continue
                print('\n')
                time.sleep(1)
            elif inp == '8': # Add Order to the List
                if not imported:
                    print("Please import the inventory with option [1] first.")
                    time.sleep(1)
                    continue
                productID = input("Please enter the product ID: ")
                self.program.addOrder(productID)
                print('\n')
            elif inp == '9': # Calculate shortest path with the order list
                # algo = input("Please enter the algorithm you want to use: (BRUTEFORCE / DP / GREEDY)\n")
                if not imported:
                    print("Please import the inventory with option [1] first.")
                    time.sleep(1)
                    continue
                if self.program.orderIsEmpty():
                    print("Please import order file with option [7] or input products manually with option [8] first.")
                    time.sleep(1)
                    continue
                try:
                    sequence = self.program.computePath()
                except Exception:
                    sequence = self.program.order.id_list
                    print("The execution time is over the timeout.\n")
                paths = self.program.getPathBySequence(sequence)
                print("The optimal sequence is the following: ")
                print(sequence)
                print('\n')

                print("Please follow this instruction to get the products:")
                instrs = self.program.paths2Instrs(paths)
                for instr in instrs:
                    print(instr)
                    time.sleep(0.1)

                self.program.displayPathInventory(paths, sequence)
                time.sleep(1)
            elif inp == '10': # Input Order Manually
                if not imported:
                    print("Please import the inventory with option [1] first.")
                    time.sleep(1)
                    continue
                input_order = input("Please enter the order that you want to input manually:")
                self.program.inputOrder(input_order)
                time.sleep(1)
            elif inp == '11': # Load Order List
                if not imported:
                    print("Please import the inventory with option [1] first.")
                    time.sleep(1)
                    continue
                order_list_file_name = input("Please enter the order list file you want to load: ")
                try:
                    self.program.loadOrderList(order_list_file_name)
                except Exception:
                    print("Cannot find the file\n")
                    time.sleep(1)
                    continue
                self.order_list_file_label = '<' + order_list_file_name + '>'
                time.sleep(1)

            elif inp == '12': # Get next order from Order List
                if not imported:
                    print("Please import the inventory with option [1] first.")
                    time.sleep(1)
                    continue
                if self.program.orderListIsEmpty():
                    print("Please import the order list with option [11] or input order manually with option [10] first.")
                    time.sleep(1)
                    continue
                (cur_index, cur_order) = self.program.getNextOrder()
                if cur_index == -1:
                    print("The order list are all fulfilled.\n")
                    time.sleep(1)
                else:
                    print("Order to fulfill: Order " + ("%04d" % (cur_index + 1)) + "\n")
                    output_products = ""
                    for cur_product in cur_order:
                        output_products += cur_product + " "
                    print("Items: " + output_products + "\n")
                    try:
                        sequence = self.program.computePath(cur_order)
                    except Exception:
                        sequence = cur_order
                        print("The execution time is over the timeout.\n")
                    paths = self.program.getPathBySequence(sequence)
                    print("The optimal sequence is the following: ")
                    print(sequence)
                    print('\n')

                    print("Please follow this instruction to get the products:")
                    instrs = self.program.paths2Instrs(paths, cur_order)
                    for instr in instrs:
                        print(instr)
                        time.sleep(0.1)

                    self.program.displayPathInventory(paths, sequence)
                    time.sleep(1)

            elif inp == '13': # Get specific order from Order List
                if not imported:
                    print("Please import the inventory with option [1] first.")
                    time.sleep(1)
                    continue
                if self.program.orderListIsEmpty():
                    print("Please import the order list with option [11] or input order manually with option [10] first.")
                    time.sleep(1)
                    continue
                order_index = int(input("Please enter the order's index that you want to access:"))
                (flag, cur_order) = self.program.getSpecificOrder(order_index - 1)
                if flag == -1:
                    print("The index is out of range.\n")
                    time.sleep(1)
                elif flag == 0:
                    print("The order of this index is fulfilled.\n")
                    time.sleep(1)
                else:
                    print("Order to fulfill: Order " + ("%04d" % order_index) + "\n")
                    output_products = ""
                    for cur_product in cur_order:
                        output_products += cur_product + " "
                    print("Items: " + output_products + "\n")
                    try:
                        sequence = self.program.computePath(cur_order)
                    except Exception:
                        sequence = cur_order
                        print("The execution time is over the timeout.\n")
                    paths = self.program.getPathBySequence(sequence)
                    print("The optimal sequence is the following: ")
                    print(sequence)
                    print('\n')

                    print("Please follow this instruction to get the products:")
                    instrs = self.program.paths2Instrs(paths, cur_order)
                    for instr in instrs:
                        print(instr)
                        time.sleep(0.1)

                    self.program.displayPathInventory(paths, sequence)
                    time.sleep(1)



            elif inp == '0':
                print("\n\nTHANK YOU FOR USING WPLRS!!\n\n")
                time.sleep(1)
                break
            else:
                print("Invalid Option!\n")
                time.sleep(1)
                print("Please input a number between 1 and 13.\n\n")
                time.sleep(1)
```

- **Product Class:**
  - *Attributes:*
    * ID(Unique Long Value): unique ID for product that is autogenerated
    * Name(String): Human readable ID for product
    * Description(String): details about product
    * Location(Coordinate): tuple coordinate (x,y) value for location
    * Accessibility(Array for Cardinal Directions): Bit array for cardinal directions where '0' is not accessible from that direction and '1' means it is accessible from that direction
  - *Methods:*

  - *getID():* Returns ID of the product
  - *getName():* Returns Name of the product
  - *getDescription():* Returns Description of the product
  - *getLocation():* Returns Location of the product
  - *getAccessibility():* Returns Accessibility of the product
  - *setName(name):* Modifier method for updating the name of the product
  - *setDescription(description):* Modifier method for updating the Description of the product
  - *setLocation(loc):* Modifier method for updating the Location of the product
  - *setAccessibility(Accessibility):* Modifier method for updating the Accessibility of the product

- *Critical Code:*

```python
class Product:

    def __init__(self,
                 ID=None,
                 name=None,
                 description=None,
                 location=None,
                 accessibility=None):
        self.ID = ID # ID of the product
        self.name = name # string -> Human Readable form of the product ID
        self.description = description # string -> description of the product
        self.location = location # [(x,y)] tuple -> location of the product
        self.accessibility = accessibility # 2D array -> accessibility matrix

    def getID(self):
        '''Returns ID of the product'''
        return self.ID

    def getName(self):
        '''Returns name of the product'''
        return self.name

    def getDescription(self):
        '''Returns description of the product'''
        return self.description

    def getLocation(self):
        '''Returns location of the product'''
        return self.location

    def getAccessibility(self):
        '''Returns accessibility of the product'''
        return self.accessibility

    def setName(self, name):
        '''Sets the name of the product'''
        self.name = name
        print("Name of the product has been successfully set!")

    def setDescription(self, description):
        '''Sets the description of the product'''
        self.description = description
        print("Description of the product has been successfully set!")

    def setLocation(self, location):
        '''Sets the location of the product'''
        self.location = location
        print("Location of the product has been successfully set!")

    def setAccessibility(self, accessibility):
        '''Sets the accessibility of the product'''
        self.accessibility = accessibility
        print("Accessibility of the product has been successfully set!")
```

- **Order Class:**
  - *Attributes:*
    - ID(Unique Long Value): unique ID for order
  - *Methods:*
    - *addOrder(self, ID):* add existing product ID to current order list
    - *importOrder(self,filename):* import list of product IDs and add them to current order list
  - *Critical Code:*

```python
class Order:

    def __init__(self, id_list=[]):
        self.id_list = id_list

    def addOrder(self, ID):
        '''Adds the given ID to the order list'''
        if ID not in self.id_list:
            self.id_list.append(ID)
            print("Given ID has been successfully added to the order")
        else:
            print("Given product is already in the order list!")

    def importOrder(self, file_name):
        '''Gets the product ID list from the given file'''
        if file_name.endswith('.txt'):
            self._importfromtxt(file_name)
        elif file_name.endswith('.csv'):
            # self._importfromcsv(file_name) # TODO: implement import from csv
            raise ValueError("Import from csv is not implemented yet!")
        else:
            raise ValueError("Invalid file type is given!")

    def _importfromtxt(self, file_name):
        '''Imports the order list from a txt file, each line one product ID'''
        file_path = os.path.join(os.path.join(os.getcwd(),
                                              'order_lists'),
                                              file_name)
        with open(file_path, 'r') as f:
            file_contents = f.read().split('\n')
        for line in file_contents:
            if line:
                self.id_list.append(line)
        print("Order list has been successfully imported from the given file.")
```

- **Order List Class:**
  - *Attributes:*
    - curIndex
    - fulfilled: list that tracks whether order associated with index has been filled
    - order list: list of orders
  - *Methods:*

    * *inputOrder(self, order string):* add new order to existing order list
    * *importOrderList(self,filename):* import list of orders from a file
    * *nextOrder(self,filename):* retrieves next order from list that has not been fulfilled
    * *specificOrder(self,index):* retrieves order associated with passed index
- *Critical Code:*

```python
class OrderList:

    def __init__(self):
        self.curIndex = -1
        self.fulfilled = []
        self.order_list = []

    def importOrderList(self, file_name):
        if file_name.endswith('.txt'):
            self._importfromtxt(file_name)
        elif file_name.endswith('.csv'):
            # self._importfromcsv(file_name) # TODO: implement import from csv
            raise ValueError("Import from csv is not implemented yet!")
        else:
            raise ValueError("Invalid file type is given!")

    def _importfromtxt(self, file_name):
        self.order_list = []
        self.fulfilled = []
        self.curIndex = -1
        file_path = os.path.join(os.path.join(os.path.dirname(os.path.abspath(__file__)),
                                              'order_lists'),
                                              file_name)
        with open(file_path, 'r') as f:
            orders = f.read().strip().split('\n')

        for order in orders:
            products = order.strip().split('\t')
            self.order_list.append(products)
            self.fulfilled.append(0)
        print("\nOrder list has been successfully loaded from the given file.")

    def inputOrder(self, order_string):
        products = order_string.strip().split('\t')
        self.order_list.append(products)
        self.fulfilled.append(0)
        print("\nThe new order has been successfully added to the order list.")

    def nextOrder(self):
        if self.curIndex >= len(self.order_list):
            return (-1, [])

        self.curIndex += 1
        while self.fulfilled[self.curIndex] == 1:
            self.curIndex += 1
        self.fulfilled[self.curIndex] = 1
        return (self.curIndex, self.order_list[self.curIndex])

    def specificOrder(self, index):
        if index >= len(self.order_list):
            return (-1, [])
        if self.fulfilled[index] == 1:
            return (0, self.order_list[index])
        self.fulfilled[index] = 1
        return (1, self.order_list[index])
```

- **Path Finder Class:**
  - *Attributes:*
    - None
  - *Methods:*
    - *paths2Instrs(paths, item ids):* converts inputted paths to human readable instructions
  - *Critical Code:*

```
def paths2Instrs(paths, item_ids):
    '''Get the user friendly instruction from a path list'''
    # Dictionary indicating where to turn
    direct2turn = {('North', 'East'): 'Right', ('North', 'South'): 'Back', ('North', 'West'):
         'Left',
                   ('East', 'North'): 'Left', ('East', 'South'): 'Right', ('East', 'West'):
                        'Back',
                   ('South', 'North'): 'Back', ('South', 'East'): 'Left', ('South', 'West'):
                        'Right',
                   ('West', 'North'): 'Right', ('West', 'East'): 'Back', ('West', 'South'):
                        'Left'}
    # Each path has a bunch on coordinates. Starts with the first coordinate.
    coord = paths[0][0] # Start point
    next_coord = paths[0][1]
    direction = 'North'
    direction = getDirection(coord, next_coord, direction)
    print("Start at {}".format(coord))
    step_count = 0
    for idx, path in enumerate(paths):
        if len(path) == 1:
            print("Pick Up Item: {} at {}".format(item_ids[idx], next_coord))
            continue
        for i, coord in enumerate(path[:-1]):
            next_coord = path[i+1]
            next_direct = getDirection(coord, next_coord, direction)
            if direction == next_direct:
                step_count += 1
                continue
            else:
                turn_direct = direct2turn[(direction, next_direct)]
                print('Move Towards {}, For {} Steps, Until '.format(direction, step_count) +
                      'You Reach Point {} and Turn {} to {} Direction.'.format(coord,
                           turn_direct, next_direct))
                step_count = 1 # Reset step count
                direction = next_direct
        if idx == len(paths) - 1:
            print("Move Towards {}, For {} Steps and End Your Tour at {}".format(direction,
                 step_count, next_coord))
        else:
            print("Move Towards {}, For {} Steps, and Pick Up Item: {} at {}".format(direction,
                 step_count, item_ids[idx], next_coord))
            step_count = 0
```

- **Inventory:** dictionary of items/products that tracks the location and availability, and accessibility of all products. Details:
  - *Key:* Product ID
  - *Value:* Product Object

- *Methods:*
  - *addItem(item)*: adds item to Inventory dictionary
    - Input: Product object with requisite Name, Description, Location, and accessibility
    - Output: Code indicating success or error
  - *removeItem(ID)*: removes item from Inventory dictionary
    - Input: Product ID number
    - Output: Code indicating success or error
  - *getItem(ID)*: gets item from Inventory dictionary
    - Input: Product ID number
    - Output: Associated Product or error
  - *importItems(filename)*: parses file and adds products to Inventory
    - Input: CSV or txt file of products with each column representing a required attribute of a Product
    - Output: Number of products successfully added and number that failed
  - *getPathtoProduct(productID)*: based off productID, calculates path to the indicated product if product is in inventory
    - Input: ProductID
    - Output: Path to product as list of coordinates with directions
  - *getLocationByID(productID)*: returns product's location
    - Input: ProductID
    - Output: Coordinate location of product

- **Distances:** graph of items/products that represents information about the distance between items
  - *Vertices:* Products
  - *Edges:* Distance between Products
  - *Inventory:* The given inventory of the warehouse
  - *Methods:*
    - *calculateDistances(Inventory)*: calculates distances between all Products in an Inventory
      - Input: Inventory object representing Products in a warehouse
      - Output: 2D array representing distances between all Products
    - *traverse(source_point, destination_point)*: Calculates distance between given two points. This is a helper function for calculating the distances between all products in the warehouse.
      - Input: Source and Destination Point
      - Output: The distance between input points or error indicating that the product cannot be reached.

- **ComputePath Class:**
  - *Methods:*
    - *parallel(self, order list:* runs parallel genetic algorithms on list of orders in order to calculate shortest paths
    - *evaluation, crossover, Mutation, etc.:* helper functions to perform genetic algorithm. Details on algorithm performance and parameters included in test document "Final Release Test Document.pdf"
    -
  - *Critical Code:*

```
    def evaluation(self):
        best_score=self.Fitness(self.best)
        for life in self.lives:
            score = self.Fitness(life)
```

```python
            if score > best_score:
                self.best = life
                best_score = score

    def crossover(self, parent1, parent2):
        left = random.randint(1, len(parent1) - 2)
        right = random.randint(left, len(parent1) - 2)
        newgene = collections.deque()
        newgene.extend(parent1[left:right])
        point = 0
        for g in parent2[1:-1]:
            if g not in parent1[left:right]:
                if point<left:
                    newgene.appendleft(g)
                    point += 1
                else:
                    newgene.append(g)
        newgene.appendleft(0)
        newgene.append(len(parent1)-1)
        self.crossCount += 1
        return list(newgene)

    def Mutation(self, gene):
        newg = gene
        left = random.randint(1, len(gene) - 2)
        right = random.randint(1, len(gene) - 2)
        newg[left], newg[right] = newg[right], newg[left]
        self.mutationCount += 1
        #print(gene)
        return list(newg)

    def select(self):
        index = random.randint(0, len(self.lives)-1)
        life = self.lives[index]
        return life

    def Child(self):
        parent1 = self.select()
        rate = random.random()
        if rate < self.crossRate:
            parent2 = self.select()
            gene = self.crossover(parent1, parent2)
        else:
            gene = parent1
        # ra = random.random()
        # if ra < self.mutationRate:
        #     gene = self.Mutation(gene)
        return gene

    def next_generation(self):
        self.evaluation()
        newLives = []
        newLives.append(self.best)
        while len(newLives) < self.num_of_life:
            newLives.append(self.Child())
        self.lives[:] = newLives
        self.generation += 1

    def New_distance(self,lifes):
        new_distance = 0.0
        for i in range(0, self.geneLength - 1):
            index1, index2 = self.ID2Index[self.order[lifes[i]]],
                self.ID2Index[self.order[lifes[i + 1]]]
            new_distance+=self.distance[index1][index2]
        return new_distance

    def Fitness(self, life):
        return 1.0/self.New_distance(life)

    def GA(self,order_list,q):
        dis=0
        n=1500
        self.order = ['000'] + order_list + ['-1']
        self.geneLength=len(self.order)
        while n > 0:
            self.next_generation()
            dis = self.New_distance(self.best)
            n -= 1
        res=[]
        for i in self.best:
            res.append(self.order[i])
        q.put(res[1:-1])
        q.put(dis)

    def RGA(self,oder_list,L,NUM,q):
        self.lives=L
        self.num_of_life=NUM
        self.best=L[-1]
        self.GA(oder_list,q)

    def initialpopulation(self,num, geneLength, q):
        lives = []
        while len(lives) < num:
            temp = [x for x in range(1, geneLength + 1)]
            random.shuffle(temp)
            gene = [0] + temp + [geneLength + 1]
            if gene not in lives:
                lives.append(gene)
        q.put(lives)

    def parallel(self, order_list):
        length = len(order_list)
        num = 1
        if len(order_list) <= 5:
            for i in range(1, len(order_list) + 1):
                num = num * i
        else:
            num = 500
        s1 = mp.Queue()
        s2 = mp.Queue()
        st1 = mp.Process(target=self.initialpopulation, args=(num, length, s1,))
        st2 = mp.Process(target=self.gre, args=(order_list, s2,))
        st1.start()
        st2.start()
        st1.join()
        st2.join()
        st1.terminate()
        st2.terminate()
        lives = list(s1.get())
        best_gre = list(s2.get())
        a = int(len(lives) / 2)
        l1 = lives[:a] + [best_gre]
        l2 = lives[a:] + [best_gre]
        q1 = mp.Queue()
        q2 = mp.Queue()
        p1 = mp.Process(target=self.RGA, args=(order_list, l1, num/2, q1,))
        p2 = mp.Process(target=self.RGA, args=(order_list, l2, num/2, q2,))
        p1.start()
        p2.start()
        p1.join()
        p2.join()
        res1 = q1.get()
        dis_1 = q1.get()
        res2 = q2.get()
        dis_2 = q2.get()
        if dis_1 < dis_2:
            print(res1, dis_1)
            return res1
        else:
            print(res2, dis_2)
            return res2
```

Chapter 5

Development Plan and Timeline

5.1 Task Division

- **Documentation: Kelvin Phan**
  - Update Documentation and make poster
  - Compile application
  - **(DEADLINE: This should be done by next week Thursday. (June 6, 11:59pm)**
- **Algorithm Team: Ahmet Aksakal and Hao Chen:**
  - Update UI for easier to read text directions
  - **(DEADLINE: This should be done by next week Thursday. (June 6, 11:59pm)**
- **Algorithm Team: Ahmet Aksakal and Hao Chen:**
  - Parallelize genetic algorithm
  - Run varied test cases
- **Review Team: Ziwen Ning:**
  - Debug program
  - Add Error handling
  - **(DEADLINE: This should be done by next week Sunday. (June 6, 11:59pm)**

Chapter 6

Back Matter

6.1 Copyright

6.2 Error Messages

- *Cannot find file:* file used to import orders is an invalid file or is unreadable
- *Please import inventory with option [1] first :* prompts user to load inventory file before using related functions
- *Execution time is over the timeout:* Route calculation has gone over execution time limit of 1 minute
- *Please import order list with option [11] first or input order manually with option [10] :* prompts user to load order file before using related functions
- *All orders are fulfilled :* alerts user that all orders have been filled
- *Does Not Exist:* object with ID does not exist in its relevant database/structure

Index