St. Xavier’s College (Autonomous), Kolkata

Department of Computer Science

QUANTUM RIDESHARING

SUVRANIL DUTTA BISWAS
MONIMOY GHOSH
KARISHMA ACHARYA

UNDER THE GUIDANCE OF
DR. ROMIT S. BEED

Submitted to the Department of Computer Science in partial fulfilment of the requirements for the degree of M.Sc. Computer Science

CERTIFICATE OF AUTHENTICATED WORK

This is to certify that the project report entitled …………………………………. submitted to Department of Computer Science, ST. XAVIER'S COLLEGE [AUTONOMOUS], KOLKATA, in partial fulfilment of the requirement for the award of the degree of Master of Science (M. Sc) is an original work carried out by:

Name	Roll No	Registration No
SUVRANIL DUTTA BISWAS	517	A01-1112-0799-17
MONIMOY GHOSH	559	A01-1112-0802-17
KARISHMA ACHARYA	538	A01-2112-0053-20

under my guidance. The matter embodied in this project is authentic and is genuine work done by the student and has not been submitted whether to this College or to any other Institute for the fulfilment of the requirement of any course of study.

...…………………………………. ...………………………………….

Signature of the Supervisor Signature of the Head of the Department

…………..………………………..

Name of the Supervisor

Date: ………………. Date: …………………

Name and Signature of the **Project Team members:**

Name Signature

1………………………………. ………………………………..……..

2………………………………. …………………………………….

3………………………………. ……………………………………

ABSTRACT

The objective of Ridesharing is to satisfy clients travelling to various destinations while minimizing the cost of travel. The proposed task is to devise an efficient Ridesharing system that operates on quantum algorithms. This project attempts to emulate a Capacitated Vehicle Routing Problem (CVRP) having a variable number of passengers travelling to various destinations from a central hub with a fixed number of vehicles. The CVRP is a combinatorial explosion of possible solutions, which increases super-exponentially with the number of passengers. The goal of the model is to distribute the passengers amongst the vehicles available and to formulate the routes to their destinations such that the total distance covered is minimal. The model uses Quadratic Unconstrained Binary Optimization (QUBO) implemented via Quantum Annealing on D-Wave systems.

ACKNOWLEDGEMENT

I would like to express my gratitude to our principal, Rev. Dr. Dominic Savio for giving us this opportunity to work on this project. I would like to that our Head of Department and mentor, Dr. Romit S. Beed for guiding us through the project. I would also like to thank the others in the team for their contribution and collaboration that made the task of completing the project seamless and enjoyable.

TABLE OF CONTENTS

TABLE OF DIAGRAMS

Figures

1. INTRODUCTION

1.1 Background

This project deals with solving the CVRP using a quantum annealer. Several approaches have been discussed to explore the work done in this domain. Laporte and Semet (2002) [1] presented a few heuristics for solving the CVRP namely construction heuristics, improvement heuristics, and metaheuristics.

Metaheuristics are top-level methods that assist local improvement operators in identifying a global solution. For the (C)VRP, Groër et al. (2010) [2] described a library of local search heuristics. Crispin and Syrichas (2013) [3] suggested a metaheuristic for vehicle scheduling based on classical-quantum annealing. They employed a stochastic variation dubbed Path-Integral Monte Carlo (PIMC) to replicate the quantum fluctuations of a quantum system on a classical computer to approximate quantum annealing. This is due to the quantum annealing hardware. The Sweep algorithm (Gillett and Miller, 1974) [4] is one of the most important classical 2-Phase-Heuristics, in which viable clusters are produced by rotating a ray centered at the depot. The TSP is then solved for each cluster. Fisher and Jaikumar (1981) [5] tried a cluster-first, route-second technique to solve the VRP. Instead of utilizing a geometry-based strategy to generate the clusters, they devised a Generalized Assignment Problem (GAP). The seeds were chosen by solving capacitated location problems, and the remaining vertices were gradually integrated in their allotted route in a second stage [6]. The efficiency of a quantum annealer in solving small instances inside families of hard operational planning problems under various mappings to QUBO issues and embeddings was investigated by Rieffel et al. (2015) [7]. While their research did not provide results that were competitive with state-of-the-art classical techniques, they did gain insights from the findings that can be used to program and construct future quantum annealers. A tree-search-based quantum-classical framework is described by Tran et al. (2016) [8]. To get strong candidate solutions, the authors utilize a quantum annealer to sample from the configuration space of a relaxed issue, followed by a classical processor that maintains a global search tree. They put their technique to the test and compare variants on minor problem examples from three scheduling fields. Many approaches have a hybrid structure, as can be seen in general. That is, traditional bottlenecks are delegated to quantum computing devices that perform local quantum searches iteratively [9][8][10].

Using the classical 2-Phase-Heuristic, the CVRP can be partitioned into smaller optimization problems. Laporte and Semet (2002) [1] divided the CVRP into two phases using this heuristic: the clustering phase and the routing phase. The clustering phase can be compared to the NP-complete Knapsack Problem (KP) (Karp, 1972) [11], which involved cramming various sized things (in this case, customers) into a limited number of knapsacks (here, vehicles). The sum of the objective values of the things in a knapsack should be maximized as a result, reducing the Euclidean distance between consumers assigned to a vehicle. Feld et al. (2019) [12] had provided an intuitive way to partition the problem into smaller subproblems.

1.2 Objectives

This project aims to develop a model for the CVRP such that:

- All the commuters are served while making the overall journey minimal.
- All the vehicles are utilized to their fullest.

- The model is no less efficient than other existing solutions.

1.3 Purpose, Scope and Applicability

- **Purpose:** The purpose of this project is to create a ridesharing system based on quantum methodologies. Many implementations of ridesharing have already been carried out using the classical setup. Quantum Computing is an upcoming research area of Computer Science. This field has till date been limited to theoretical and experimental aspects due to unavailability of quantum computers for practical use. Although quantum computers are now more widely available as compared to the past, they are still limited in terms of computing power, storage and usability. Despite these limitations, quantum computing has started to show promise in various fields, achieving results that are impossible for classical computers. This project tries to establish a groundwork in the domain of ridesharing using quantum computers with the vision that, in the future, when quantum computers become more suitable for practical applications, the foundations laid down by this project can be expanded for more extensive applications.
- **Scope:** This project considers a ridesharing ecosystem where there exists a particular location called hub, say H, where M vehicles (called shuttles) are present. There are N commuters at H who want to go to locations $X_1, X_2, \ldots, X_N$. The goal is to distribute the N commuters amongst the M shuttles and plan the routes of sending them to their destinations such that the overall journey undertaken by the shuttles is minimum.

  To create this model the following assumptions have been made:

  a) A commuter can board the vehicle only from the hub.

  b) The total number of commuters is always less than the total combined capacity of the vehicles available.

  c) The journey of a shuttle stops when it has dropped of its last onboard passenger.

  d) All the destinations are within a radius of 50 km from the hub.

  As current gate-based quantum computers have limited computing capability due to very limited number of qubits, they are only able to solve simplest versions of the VRP. According to the model used by this project, the number of commuters should not exceed 5 while the number of shuttles cannot be greater than 2.
- **Applicability:** The model described in this project can be adapted to many real-world use cases. A few of them are as follows:

  a) The model can be extended to cater to any ridesharing application which provides carpooling services similar to Uber and Ola.

  b) It can be used for a company carpool system for the employees.

  c) It can be incorporated in a navigation application to route a journey with multiple stops.

2. SURVEY OF TECHNOLOGIES

To successfully implement a project, apart from the programming language, suitable software development kits (SDKs) and runtime ecosystems needs to be chosen. A brief description about the relevant technologies that are available for quantum computing are as follows:

- **Programming Language:** The programming language is a means of interaction between a developer and a computing device. Factors that affect the choice of a programming language include familiarity, ease of use, community, and availability of required functionality. The following languages have been surveyed while considering an appropriate choice for this project:
  a) **Python:** Python is a high-level object-oriented programming language. Python has an easy-to-understand syntax. It is easy to use and is currently one of the most popular programming languages. Python is open-source and has an enormous community support. It has an extensive library of modules and is greatly suitable for data analysis, processing, visualization, quantum programming and creation of APIs.
  b) Q#: Q# is an open-sourced programming language developed by Microsoft for developing as well as executing quantum algorithms. It enables inclusion of classical code inside a quantum program. During the implementation of an operation, Q# allows easy representation of algorithms in the circuit of quantum gates.
  c) QCL: Quantum Computing Language (QCL) is a high level, architecture independent programming language for quantum computers with C-like syntax. It allows the implementation and simulation of quantum algorithms.
- **Software Development Kit:** A software development kit is a collection of software development tools that facilitate the process of application creation. They come incorporated with advanced functionalities and greatly reduce the overhead work of a developer. The creation of certain platform-specific applications requires the use of SDKs. The following SDKs have been considered while choosing the apt ones for this project:
  a) Qiskit: Qiskit is an open-source SDK founded by IBM Research for developing circuits, pulses, and algorithms. It uses Python and provides tools for creating, manipulating and running quantum programs. It has many well-defined quantum libraries, an active community, and modules for solving optimization problems.
  b) D-Wave Ocean: D-Wave Ocean is an open-source suite of tools based on Python developed by D-Wave for solving hard problems with quantum computers. It has tools for the creation of quadratic models, constraint satisfaction, and various samplers and solvers.
  c) PennyLane: PennyLane is a cross-platform Python library for differentiable programming of quantum computers. It enables the training of a quantum computer in a similar manner as a neural network. It is primarily focused on Quantum Machine Learning and supports a variety of machine learning development tools.
  d) Cirq: Cirq is a Python library for writing, manipulating, and optimizing quantum circuits, and then running them on quantum computers and simulators. It includes abstractions for working with noisy intermediate-

scale quantum computers, where details of the hardware are vital to achieving state-of-the-art results.

- **Runtime Ecosystem:** A runtime-ecosystem is a development environment that has the provision to build, test, run and analyse programs and algorithms. They include IDEs for building programs, simulators for testing, access to runtime systems, and tools for analysis. A few popular ecosystems for quantum programming are described below:
  - **a) Amazon Braket:** Amazon Braket is a fully managed quantum computing service designed for scientific research and software development for quantum computing. It has provisions for building, testing, running, and analysing quantum programs. It provides access to quantum computers via Amazon's cloud service AWS.
  - **b) IBM Quantum Experience:** It is an online platform from IBM Quantum that provides cloud-based quantum computing services. It provides access to IBM's quantum processors, tutorials on quantum computation, and an interactive textbook. It can be used to run algorithms, experiments, and simulations on quantum computers.
  - **c) D-Wave Leap:** D-Wave Leap is a quantum cloud service that delivers real-time access to a quantum computer and a suite of quantum hybrid solvers. It also includes development kit, live editor, demos, learning resources, and a vibrant developer community. Its services are mainly suited to solve various business problems.

Based on the requirements of this project, **Python** was chosen as the preferred programming language. It is suitable for data analysis as well as quantum computing. It has well-defined quantum frameworks. The SDKs, Qiskit and D-Wave Ocean, both of which are integral for this project are based on Python. **Qiskit** is needed due to its ability to its ease-of-use and extensive tools for designing, altering, and executing quantum circuits. It has various modules for solving optimization problems which are required by the project. **Ocean** is integral to this project due to its extensive portfolio of solvers and support for quantum optimization. The **D-Wave Leap** ecosystem is considered appropriate for this project as their quantum computers have significantly larger number of qubits as compared to their compatriots.

3. REQUIREMENTS AND ANALYSIS

3.1 Problem Definition

Let there be M shuttles at the Hub H. There are N passengers at H who want to go to locations X_1, X_2, …, X_N. The input is provided to the system in the form of geographic coordinates of each node X_i, i =1, …, N.

The problem can be represented as a fully connected graph having N + 1 nodes with the hub being node 0 and the various destinations being nodes {1, 2, …, N}. Every edge (i, j) of this graph represents the cost of traveling from node i to node j. The cost is calculated as the Euclidean distance between the geographic coordinates of node i and node j and is represented as C_{ij}. The objective is to find the optimal route, for each of the M vehicles, such that it can take all the commuters allotted to it to their destinations at minimal cost.

The problem can be divided into three subproblems:

a) **Binary Quadratic Model:** The cost matrix is calculated from the inputs provided and used to create a binary quadratic model of the problem.
b) **Clustering:** The commuters are distributed amongst the M available shuttles.
c) **Routing:** For each shuttle, the minimal route, to take the allotted commuters to their destinations, is found.

Constraints: The constraints are as follows:

- The maximum capacity of each shuttle is four commuters.
- There can be only one shuttle at any node (other than the hub) at any time instant.
- A shuttle must be at only one place at any time instant.

3.2 Requirement Specification

The system has the following requirements:

- The input for each destination X_i, i =1, …, N, is to be provided in the form of the geographic coordinates of that destination.
- The inputs are to be translated into a fully connected graph of N+1 nodes and a corresponding cost matrix is to be created.
- A binary quadratic model is to be created using the cost matrix.
- The output of the solver is to be represented in the form of a graph displaying the generated routes.

3.3 Software and Hardware Requirements

Hardware Requirements:

- Processor: Intel® Core 2 Duo E8400 or higher/ AMD® Athlon II X2 or higher
- Clock Speed: 2.2 GHz or higher
- System Architecture: 64-bit x86
- RAM: 4GB or higher

Software Requirements:

- OS: Windows 8 or higher / Ubuntu 16.04 or higher / mac OS X 10.13 or higher
- Python 3.6 or higher
- Qiskit
- D-Wave Ocean
- Jupyter Notebook

4. SYSTEM DESIGN

4.1 Conceptual Model

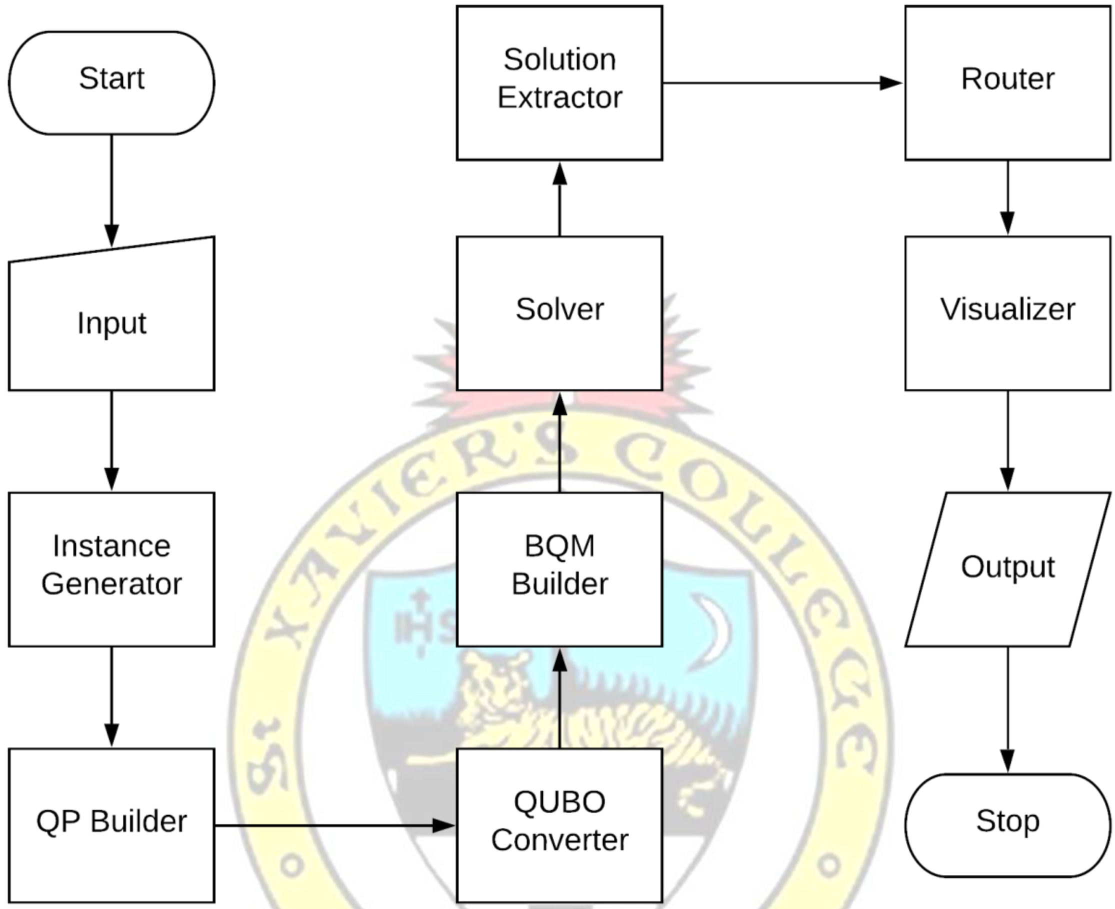

Figure 1. Data Flow Diagram of QUBO Solver

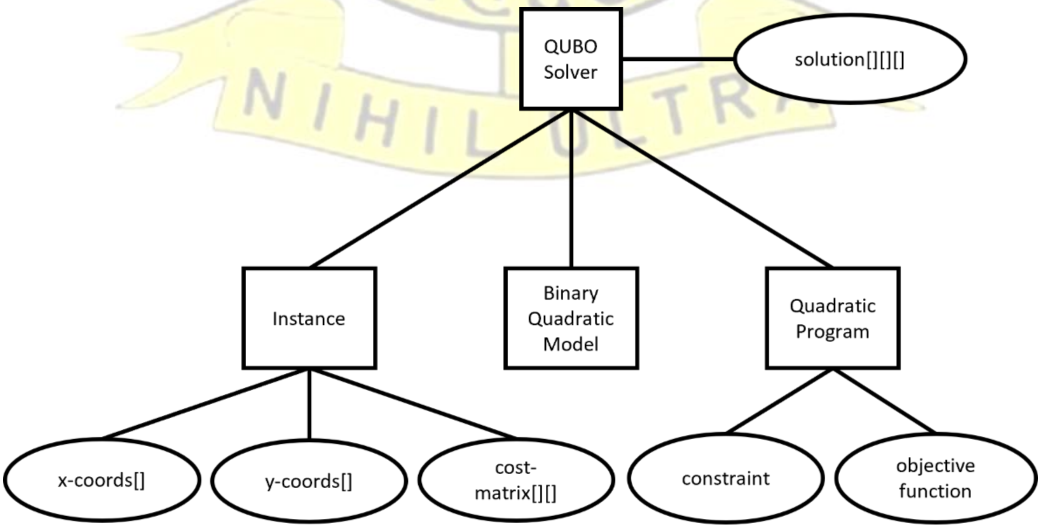

Figure 2. Object Model of QUBO Solver

4.2 Basic Modules

The QUBO solver comprises of the following modules:

a) Instance Generator: Creates an instance that is used to build the Quadratic Program (QP). The instance comprises of two linear arrays containing the x and y coordinates of the various nodes and a 2-D matrix containing the cost to travel from one node to another.

b) QP Builder: Builds the QP from the instance and adds the corresponding constraints and the objective function.

c) QUBO Converter: Converts the QP to a QUBO instance by appending all constraints to the objective function in the form of penalties.

d) BQM Builder: Builds a Binary Quadratic Model (BQM) from the QUBO instance.

e) Solver: Solves the BQM.

f) Solution Extractor: Extracts the solution and maps it to a 3-D list.

g) Router: Using the distribution made by the solver, generates the appropriate routes.

h) Visualizer: Displays the generated routes in a graphical format.

i) Simulator: Using the distribution obtained from the solver, initializes a simulation which runs for a fixed number of iterations.

j) Request Handler: For a random request generated in an iteration, handles the request and either assigns it to a suitable shuttle or discards it.

k) Map Drawer: Draws the state of the simulation in every iteration on the map.

4.3 Data Design

Object Model of the QUBO Solver: The QUBO Solver is a user-defined datatype. It acts as a wrapper for the entire solution model. It has the following components:

a) **Instance**: It is a user-defined datatype. It represents the problem space in the form of a graph. It has the following three attributes:
   - **x-coords:** It is an array that stores the x-coordinates of the nodes.
   - **y-coords:** It is an array that stores the y-coordinates of the nodes.
   - **cost-matrix:** It is a 2-D array that stores the cost to travel from one node to another.

b) **Binary Quadratic Model:** It is an imported datatype that is used represent a BQM instance.

c) **Quadratic Program:** It is an imported datatype that is used represent a QP instance. It has the following three attributes:
   - **constraint:** It is an attribute that stores a constraint that binds the QP.
   - **objective function:** It is an attribute that stores the objective function of the QP.

d) **solution:** It is an attribute that stores the solution obtained from the solver. It is in the form of a 3-D array.

Object Model of a Shuttle: The Shuttle is a user-defined datatype that is used to represent the state of a shuttle at any instant. It has the following components:

a) **Number:** A unique number identifying the shuttle.

b) **Passengers:** A list containing the details of the current passengers.

c) **Current Position:** Denotes the current position of the shuttle.

d) **Route:** The current route that the shuttle is following.
e) **Drop order:** Denotes the order in which the shuttle will service its passengers.
f) **Occupancy:** The current occupancy of the shuttle.
g) **Service:** Denotes whether the shuttle is servicing a pickup request or not.
h) **Pickup:** The request of the passenger whom the shuttle is going to pick up.

Object Model of Simulation: The Simulation is a user-defined datatype that is used to represent the state of the simulation at any instant. It has the following components:

a) **m:** No. of vehicles available for service.
b) **n:** No. of initial passengers.
c) **Center:** Index of the hub.
d) **Total Nodes:** Total number of nodes in the problem.
e) **Clusters:** The clusters generated from the initial requests generated by the QUBO solver.
f) **Requests:** The list of initial requests.
g) **Shuttles:** The list of shuttle objects.

4.4 Procedural Design

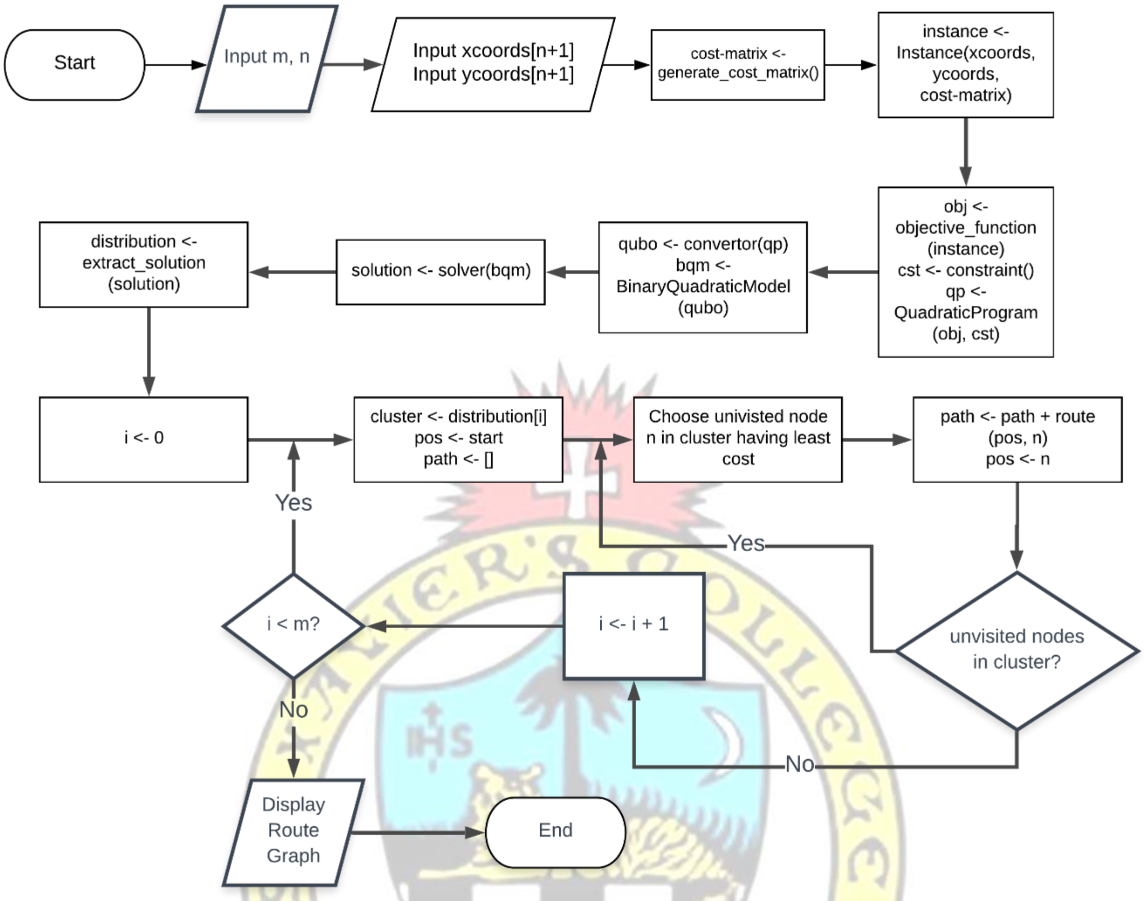

Figure 3. QUBO Solver Algorithm Flowchart

The process starts by taking as input the values m and n which are the number of shuttles and the number of commuters respectively. For each commuter, the coordinates of their destination are taken as input. The x-coordinates and y-coordinates of all the nodes including that of the hub are stored in two arrays. A cost matrix is generated corresponding to the inputs. Next, an instance is created using the coordinates and the cost matrix. The instance is fed to the objective function generator to create an objective function. The necessary constraints are also generated. The objective function and constraints are used to create the QP which is then converted to a QUBO instance. The QUBO is transformed into a BQM. The BQM is then fed to the solver to obtain the distribution of commuters amongst the m shuttles. For each cluster in the distribution, the following steps are performed to generate the route for that cluster:

i. Choose the node having the least cost amongst the unvisited nodes.
ii. Create a route between the current position and the chosen node.
iii. Append the route to the existing path.
iv. If there are any unvisited nodes left in the cluster, go to step 1.

After all the routes have been created for all the clusters, they are displayed in the form of a graph.

A simulation is created based on the above-obtained initial state. The initial state is shown in pictorial format by drawing the drop locations and the routes on the map. The simulation is run for a fixed number of iterations. For each iteration, a random request is generated and a request handler is used to determine whether the request is serviceable or not. If the request is not serviceable, the request is declined. Otherwise, suitable shuttles which can service the request are determined. If a shuttle is currently idle, and the pickup location is within a predefined radius of the current position of the shuttle, then it qualifies as a suitable shuttle. On the other hand, if a shuttle currently has passengers but its occupancy is not full, it is suitable for the request if the pickup location is within a predefined radius of the current position of the shuttle and the drop location of the request falls within the predefined radius of any node along its untraveled route. Among the suitable shuttles, the best fit is selected. The chosen shuttle is rerouted to accommodate the pickup and drop locations of the new passenger. It is set to service mode till the passenger is onboard. While in service mode, the shuttle does not entertain any further requests. The state of the simulation in every iteration is documented using logs as well as images of the map representing the current routes of the shuttles, the various drop locations, and the current positions of the shuttles.

4.5 Output Prototype

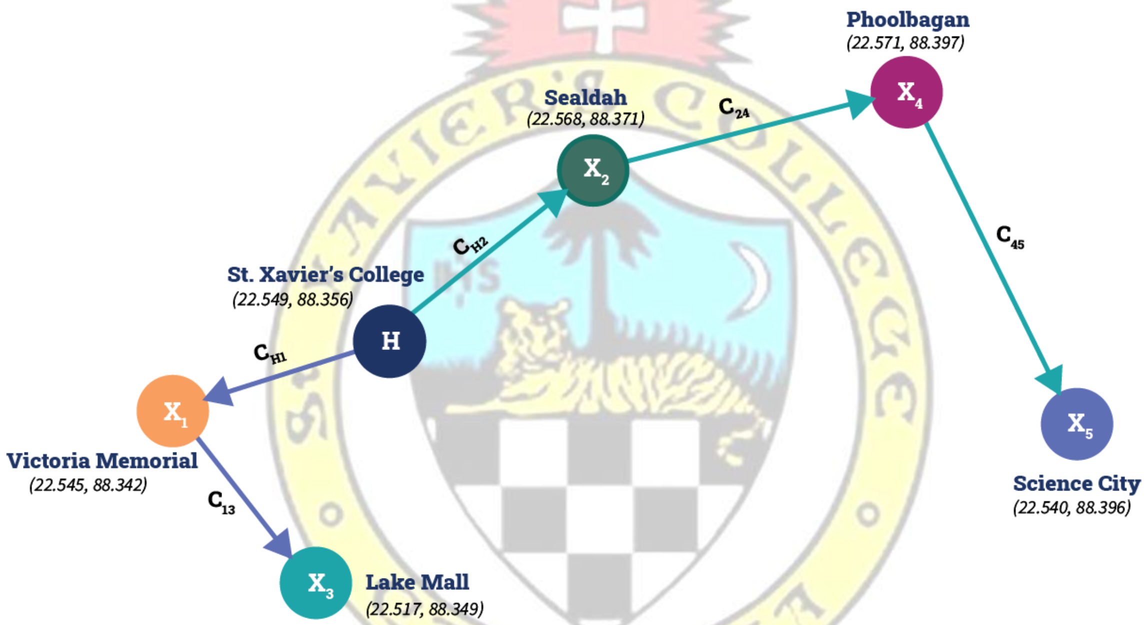

Figure 4. Cluster graph

Figure 4 shows a diagrammatical representation of the output as displayed by the visualizer. The nodes are distributed amongst the available shuttles (here, 2) and the routes are generated accordingly. Routes for each shuttle are represented by their respective colors. The total cost of journey for each shuttle is the sum of the costs of travelling to the nodes in accordance with their routes.

5. IMPLEMENTATION AND TESTING

5.1 Source Code

constants.py

Contains the constant data used in the implementation of the application.

```
import numpy as np

dist_matrix = np.array([
    [0, 294, -1, -1, ……………………………………………………………., -1, -1, -1, -1],
    [294, 0, 348, -1, …………………………………………………………..., -1, -1, -1, -1],
    [-1, 348, 0, 333, ..............................................................................................., -1, -1, -1, -1],
    [-1, -1, 333, 0, ..................................................................................................., -1, -1, -1, -1],
                                        .
                                        .
                                        .
                                        .
                                        .
    [-1, -1, -1, -1, ...........................................................................…………..... 0, 373, -1, -1],
    [-1, -1, -1, -1, ...........................................................................………….., 373, 0, 175, -1],
    [-1, -1, -1, -1, …………………………………………………………………., 175, 0, 436],
    [-1, -1, -1, -1, .........................................................................…………......., -1, -1, 436, 0],
])
```

[*Dimension: 122 rows x 122 columns*]

```
coordinates = [
                {'latitude': 22.59257785, 'longitude': 88.39495508},
                {'latitude': 22.59380613, 'longitude': 88.39813082},
                {'latitude': 22.59582682, 'longitude': 88.40139238},
                {'latitude': 22.59687855, 'longitude': 88.40372944},
                                        .
                                        .
                                        .
                                        .
                                        .
                                        .
                                        .
                {'latitude': 22.56493018, 'longitude': 88.41030496},
                {'latitude': 22.56208484, 'longitude': 88.40822695},
                {'latitude': 22.56069259, 'longitude': 88.40911035},
                {'latitude': 22.55804088, 'longitude': 88.41162285}]
```

[*Dimension: 122 items*]

```
image_coordinates = [(143, 109), (180, 109), (220, 109), ……………………….
                    ……………………………..., (58, 538), (58, 569), (59, 614)]
```

[*Dimension: 122 items*]

utility.py

Contains functions to generate initial requests, to generate a VRP instance from the obtained requests, and to generate the optimal route for a particular cluster along with the service ordering.

```
import numpy as np
import random
from constants import coordinates
from routes import Route, join_route


def generate_initial_requests(n, center, total_nodes):
    """Generate the initial set of requests.
    Args:
        n: No. of nodes excluding the hub.
        center: Index of the node considered as the hub.
        total_nodes: Total number of nodes
    Returns:
        A numpy array of size n containing the initial requests.
    """
    requests = np.zeros(n, dtype=np.int8)
    i = 0
    while i < n:
        val = random.randrange(total_nodes)
        if val != center:
            requests[i] = val
            i += 1
    return requests


def generate_vrp_instance(n, center, requests):
    """Generate a random VRP instance.
    Args:
        n: No. of nodes excluding the hub.
        center: Index of the node considered as the hub.
        requests: The initial batch of drop requests.
    Returns:
        A list of (n + 1) x coordinates, a list of (n + 1) y coordinates and an (n + 1) x (n + 1)
numpy array as the cost matrix.
    """

    # Generate VRP instance
    xs = np.zeros(6)
    ys = np.zeros(6)
    ys[0] = 0
    xs[0] = 0

    for i in range(1, 6):
        ys[i] = (coordinates[requests[i - 1]]['latitude'] - coordinates[center]['latitude']) * 500
        xs[i] = (coordinates[requests[i - 1]]['longitude'] - coordinates[center]['longitude']) * 500
```

```
    instance = np.zeros((n + 1, n + 1))
    for ii in range(n + 1):
        for jj in range(ii + 1, n + 1):
            instance[ii, jj] = (xs[ii] - xs[jj]) ** 2 + (ys[ii] - ys[jj]) ** 2
            instance[jj, ii] = instance[ii, jj]

    # Return output
    return instance, xs, ys


def generate_cluster_route(center, requests, cluster):
    """Generate the route for a particular cluster.
    Args:
        center: Index of the node considered as the hub.
        requests: The initial batch of drop requests.
        cluster: The cluster for which the route is to be generated.
    Returns:
        A path for the cluster, and a list containing the order in which the requests belonging to
the cluster are serviced.
    """

    path = []
    edge = []
    drop_order = []
    dist = 0
    nodes = [requests[i - 1] for i in cluster]
    closest = nodes[0]
    start = center
    route = Route()
    while len(nodes) > 0:
        closest_distance = 99999999  # Earth's cicumference: 40,075,000 m.
        for node in nodes:
            temp = route.generate(start, node)
            if route.distance < closest_distance:
                closest_distance = route.distance
                closest = node
                edge = temp
        join_route(path, edge[:-1])
        dist = dist + closest_distance
        start = closest
        nodes.remove(closest)
        drop_order.append((center, closest))
    path.append(closest)

    return path, drop_order
```

backend.py

Contains solver that is used to solve the Vehicle Routing Problem

```
from dwave.system import LeapHybridSampler


class Backend:
    """Class containing all backend solvers that may be used to solve the Vehicle Routing
Problem."""

    def __init__(self, vrp):
        """Initializes required variables and stores the supplied instance of the VehicleRouter
object."""

        # Store relevant data
        self.vrp = vrp

        # Initialize necessary variables
        self.result_dict = None

    def solve(self):
        """Takes the solver as input and redirects control to the corresponding solver."""

        # Call Leap Solver
        self.solve_leap()

    def solve_leap(self):
        """Solve using Leap Hybrid Sampler."""

        # Solve
        sampler = LeapHybridSampler()
        self.vrp.result = sampler.sample(self.vrp.bqm)

        # Extract solution
        self.vrp.timing.update(self.vrp.result.info)
        self.result_dict = self.vrp.result.first.sample
        self.vrp.extract_solution(self.result_dict)
```

vehicle_routing.py

Contains an abstract class to build and solve the VRP, a function to convert the VRP into a QUBO and eventually builds a BQM to be solved by a D-Wave solver

```
import numpy as np
import dimod
import time

from functools import partial
from backend import Backend
```

```
from dwave.embedding.chain_strength import uniform_torque_compensation
from qiskit_optimization.converters import QuadraticProgramToQubo
from qiskit_optimization.algorithms import OptimizationResult


class VehicleRouter:
    """Abstract Class for solving the Vehicle Routing Problem. To build a VRP solver, simply
inherit from this class and override the build_quadratic_program function in this class."""

    def __init__(self, n_clients, n_vehicles, cost_matrix, **params):

        """Initializes the VRP by storing all inputs, initializing variables for storing the
quadratic structures and results and calls the rebuild function to build all quadratic structures.
        Args:
            n_clients: No. of nodes in the problem (excluding the hub).
            n_vehicles: No. of vehicles available for service.
            cost_matrix: (n_clients + 1) x (n_clients + 1) matrix describing the cost of moving
from node i to node j.
            penalty: Penalty value to use for constraints in the QUBO. Defaults to automatic
calculation by qiskit converters.
            chain_strength: Chain strength to be used for D-Wave sampler. Defaults to automatic
chain strength calculation via uniform torque compensation.
            num_reads: Number of samples to read. Defaults to 1000.
        """

        # Store critical inputs
        self.n = n_clients
        self.m = n_vehicles
        self.cost = np.array(cost_matrix)

        # Extract parameters
        self.penalty = params.setdefault('constraint_penalty', None)
        self.chain_strength = params.setdefault('chain_strength',
partial(uniform_torque_compensation, prefactor=2))
        self.num_reads = params.setdefault('num_reads', 1000)

        # Initialize quadratic structures
        self.qp = None
        self.qubo = None
        self.bqm = None
        self.variables = None

        # Initialize result containers
        self.result = None
        self.solution = None

        # Initialize timer
        self.clock = None
        self.timing = {}
```

```
        # Initialize backend
        self.backend = Backend(self)

        # Build quadratic models
        self.rebuild()

    def build_quadratic_program(self):

        """Dummy function to be overridden in child class. Required to set self.variables to contain the names of all variables in the form of a numpy array and self.qp to contain the quadratic program to be solved."""

        # Dummy. Override in child class.
        pass

    def build_bqm(self):

        """Converts the quadratic program in self.qp to a QUBO by appending all constraints to the objective function in the form of penalties and then builds a BQM from the QUBO for solving by D-Wave."""

        # Convert to QUBO
        converter = QuadraticProgramToQubo(penalty=self.penalty)
        self.qubo = converter.convert(self.qp)

        # Extract qubo data
        Q = self.qubo.objective.quadratic.to_dict(use_name=True)
        g = self.qubo.objective.linear.to_dict(use_name=True)
        c = self.qubo.objective.constant

        # Build BQM
        self.bqm = dimod.BQM(g, Q, c, dimod.BINARY)

    def rebuild(self):

        """Builds the quadratic program by calling build_quadratic_program and then the QUBO and BQM by calling build_bqm."""

        # Begin stopwatch
        self.clock = time.time()

        # Rebuild quadratic models
        self.build_quadratic_program()
        self.build_bqm()

        # Record build time
        self.timing['qubo_build_time'] = (time.time() - self.clock) * 1e6

    def extract_solution(self, result_dict):
```

```
        """Uses a result dictionary mapping variable names to the solved solution to build the self.solution variable in the same shape as self.variables and containing the corresponding solutions.
        Args:
            result_dict: Dictionary mapping variable names to solved values for these variables.
        """

        # Extract solution from result dictionary
        var_list = self.variables.reshape(-1)
        self.solution = np.zeros(var_list.shape)
        for i in range(len(var_list)):
            self.solution[i] = result_dict[var_list[i]]

        # Reshape result
        self.solution = self.solution.reshape(self.variables.shape)

    def evaluate_vrp_cost(self):

        """Evaluate the optimized VRP cost under the optimized solution stored in self.solution.
        Returns:
            Optimized VRP cost as a float value.
        """

        # Return optimized energy
        if type(self.result) == OptimizationResult:
            return self.result.fval
        else:
            return self.result.first.energy

    def evaluate_qubo_feasibility(self, data=None):

        """Evaluates whether the QUBO is feasible under the supplied data as inputs. If this data is not supplied, the self.solution variable is used instead.
        Args:
            data: Values of the variables in the solution to be tested. Defaults to self.solution.
        Returns:
            A 3-tuple containing a boolean value indicating whether the QUBO is feasible or not, a list of variables that violate constraints, and the list of violated constraints. If feasible, (True, [], []) is returned.
        """

        # Resolve data
        if data is None:
            data = self.solution.reshape(-1)
        else:
            data = np.array(data).reshape(-1)

        # Get constraint violation data
        return self.qp.get_feasibility_info(data)
```

```
    def solve(self):

        """Solve the QUBO using the selected solver."""

        # Solve
        self.backend.solve()

    def get_clusters(self):

        """Retrieve the clusters from the solution.
        Returns:
            A 2-D list of lists where each list contains the indices of the requests in that cluster.
        """

        clusters = []
        for i in range(self.solution.shape[0]):
            var_list = np.transpose(self.variables[i]).reshape(-1)
            sol_list = np.transpose(self.solution[i]).reshape(-1)
            active_vars = [var_list[k] for k in range(len(var_list)) if sol_list[k] == 1]
            cluster = [int(var.split('.')[2]) for var in active_vars]
            cluster = list(filter(lambda x: x != 0, cluster))
            clusters.append(cluster)
        return clusters
```

qubo_solver.py

Contains the class QuboSolver which inherits the VehicleRouter class and builds the quadratic program for capacitated QUBO solver.

```
import numpy as np
import networkx as nx
import matplotlib.pyplot as plt

from collections import Counter
from matplotlib.colors import rgb2hex
from vehicle_routing import VehicleRouter
from qiskit_optimization import QuadraticProgram


class QuboSolver(VehicleRouter):
    """Capacitated Qubo Solver implementation."""

    def __init__(self, n_clients, n_vehicles, cost_matrix, **params):

        """Initializes any required variables and calls init of super class."""

        # Call parent initializer
        super().__init__(n_clients, n_vehicles, cost_matrix, **params)
```

```
    def build_quadratic_program(self):

        """Builds the required quadratic program and sets the names of variables in
self.variables."""

        # Initialization
        self.qp = QuadraticProgram(name='Vehicle Routing Problem')

        # Designate variable names
        self.variables = np.array([[['x.{}.{}.{}'.format(i, j, k) for k in range(1, self.n + 1)]
                                    for j in range(self.n + 1)] for i in range(1, self.m + 1)])

        # Add variables to quadratic program
        for var in self.variables.reshape(-1):
            self.qp.binary_var(name=var)

        # Build objective function
        obj_linear_a = {self.variables[m, n, 0]: self.cost[0, n] for m in range(self.m) for n in
range(1, self.n + 1)}
        obj_linear_b = {self.variables[m, n, -1]: self.cost[n, 0] for m in range(self.m) for n in
range(1, self.n + 1)}
        obj_quadratic = {(self.variables[m, i, n], self.variables[m, j, n + 1]): self.cost[i, j] for m
in range(self.m)
                         for n in range(self.n - 1) for i in range(self.n + 1) for j in range(self.n + 1)}

        # Add objective to quadratic program
        self.qp.minimize(linear=dict(Counter(obj_linear_a) + Counter(obj_linear_b)),
quadratic=obj_quadratic)

        # Add constraints - single client service
        for k in range(1, self.n + 1):
            constraint_linear = {self.variables[i, k, j]: 1 for i in range(self.m) for j in
range(self.n)}
            self.qp.linear_constraint(linear=constraint_linear, sense='==', rhs=1,
name=f'single_service_{k}')

        # Add constraints - vehicle at one place at one time
        for m in range(self.m):
            for n in range(self.n):
                constraint_linear = {self.variables[m, k, n]: 1 for k in range(self.n + 1)}
                self.qp.linear_constraint(linear=constraint_linear, sense='==', rhs=1,
                                          name=f'single_location_{m + 1}_{n + 1}')

        # Add capacity constraints
        for i in range(self.m):
            constraint_linear = {self.variables[i, j + 1, k]: 1 for j in range(self.n) for k in
range(self.n)}
            self.qp.linear_constraint(linear=constraint_linear, sense='<=', rhs=4,
name=f'capacity_{i}')
```

```
    def visualize(self, xc, yc):

        """Visualizes solution.
        Args:
            xc: x coordinates of nodes.
            yc: y coordinates of nodes.
        """

        labels = {0: "O", 1: "A", 2: "B", 3: "C", 4: "D", 5: "E"}

        # Initialize figure
        plt.figure()
        ax = plt.gca()
        ax.set_title(f'Vehicle Routing Problem - {self.n} Clients & {self.m} Cars')
        cmap = plt.cm.get_cmap('Accent')

        # Build graph
        G = nx.MultiGraph()
        G.add_nodes_from(range(self.n + 1))

        # Plot nodes
        pos = {i: (xc[i], yc[i]) for i in range(self.n + 1)}
        #         labels = {i: str(i) for i in range(self.n + 1)}
        nx.draw_networkx_nodes(G, pos=pos, ax=ax, node_color='b', node_size=500,
alpha=0.8)
        nx.draw_networkx_labels(G, pos=pos, labels=labels, font_size=16)

        # Loop over cars
        for i in range(self.solution.shape[0]):
            # Get route
            var_list = np.transpose(self.variables[i]).reshape(-1)
            sol_list = np.transpose(self.solution[i]).reshape(-1)
            active_vars = [var_list[k] for k in range(len(var_list)) if sol_list[k] == 1]
            route = [int(var.split('.')[2]) for var in active_vars]

            # Plot edges
            edgelist = [(0, route[0])] + [(route[j], route[j + 1]) for j in range(len(route) - 1)] + [
                (route[-1], 0)]
            G.add_edges_from(edgelist)
            nx.draw_networkx_edges(G, pos=pos, edgelist=edgelist, width=2,
edge_color=rgb2hex(cmap(i)))

        # Show plot
        plt.grid(True)
        plt.show()
```

passenger.py

Contains a class passenger that defines an instance of a passenger.

```
import random


class Passenger:
    def __init__(self, request, shuttle_number):
        self.id = None
        self.request = request
        self.shuttle_number = shuttle_number
        self.generate_passenger_id()

    def details(self):
        print('ID: {}\tRequest Info: {}\t Shuttle Number: {}'.format(self.id, self.request,
self.shuttle_number))

    def generate_passenger_id(self):
        """ This function generates a random passenger id."""
        pid = 'P'
        for i in range(3):
            pid += str(random.randrange(10))
        for i in range(2):
            pid += chr(random.randrange(65, 91))
        for i in range(4):
            pid += str(random.randrange(10))
        self.id = pid
```

shuttle.py

Contains a class Shuttle that defines an instance of a shuttle.

```
import random

class Shuttle:
    def __init__(self, route, drop_order=None):
        """Initializes a shuttle.
        Args:
            route: The initial route for the shuttle.
            drop_order: A list containing the passenger requests in the order they will be serviced.
        """
        if drop_order is None:
            drop_order = []
        self.number = None
        self.passengers = None
        self.current_position = route[0]
        self.index = 0
```

```
        self.route = route
        self.drop_order = drop_order
        self.occupancy = 0
        self.service = False
        self.pickup = None
        self.generate_shuttle_number()

    def set_passengers(self, passengers):

        """Set the current passengers of a shuttle.
        Args:
            passengers: A list containing passenger objects.
        """

        self.passengers = passengers
        self.occupancy = len(passengers)

    def passenger_details(self):

        """Display the details of a passenger."""

        for passenger in self.passengers:
            passenger.details()

    def details(self):

        """Display the details of a shuttle."""

        print('Number: {}\tNumber of Passengers: {}\t\tCurrent Position: {}\t Route : {} \tDrop
order: {}'.format(
            self.number, self.occupancy, self.current_position, self.route, self.drop_order))

    def generate_shuttle_number(self):

        """This function randomly generates a car number.
        Returns:
            A string id of length 9.
        """

        number = 'WB'
        for i in range(2):
            number += str(random.randrange(10))
        number += chr(random.randrange(65, 91))
        for i in range(4):
            number += str(random.randrange(10))
        self.number = number
```

routes.py

Contains methods to generate the most optimal route between start and end points.

```
import numpy as np
from constants import dist_matrix, coordinates


class Node:
    def __init__(self, parent=None, position=None):

        """Initializing a node.
        Args:
            parent: Parent of the node.
            position: Index of the node.
        """

        self.parent = parent
        self.position = position

        self.g = 0
        self.h = 0
        self.f = 0

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


def return_path(current_node):

    """Recreates the path from the current node to the starting node.
    Args:
        current_node: The current node under consideration.
    Returns:
         A list containing the path from the starting node to the current node.
    """

    path = []
    current = current_node
    while current is not None:
        path.append(current.position)
        current = current.parent
    path = path[::-1]
    return path


def join_route(route1, route2):

    """Add the contents of the second list to the first list.
    Args:
        route1: List denoting the 1st route.
```

```
        route2: List denoting the route to be merged with the 1st route.
    Returns:
        A list with the 2nd route appended to the 1st route.
    """

    for i in route2:
        route1.append(i)
    return route1


def route_distance(route):

    """Calculate the total distance of a route.
    Args:
        route: A list containing the route.
    Returns:
        An integer denoting the total distance of the route.
    """

    distance = 0
    for i in range(1, len(route)):
        distance += dist_matrix[route[i - 1]][route[i]]
    return distance


class Route:

    def __init__(self):

        """Initialize the route object."""

        self.total_nodes = dist_matrix.shape[0]
        self.start = None
        self.end = None
        self.distance = None

    def neighbour(self, node):

        """List of neighbours of a node.
        Args:
            node: The index of the node in consideration.
        Returns:
            List containing the indices of the neighbours of the node.
        """

        nb = np.array([], dtype=np.int8)
        for i in range(self.total_nodes):
            if dist_matrix[node][i] > 0:
                nb = np.append(nb, i)
        return nb
```

```
def astar(self):

    """This function finds the optimal route based on distance."""

    start = self.start
    end = self.end
    start_node = Node(None, start)
    start_node.g = start_node.h = start_node.f = 0
    end_node = Node(None, end)
    end_node.g = end_node.h = end_node.f = 0

    end_node_coordinates = (coordinates[end]['longitude'], coordinates[end]['latitude'])

    # Initialize the open set and closed set
    open_set = []
    closed_set = []

    # Add the start node to the open set
    open_set.append(start_node)

    x = 0
    # Loop till you reach the end node
    while len(open_set) > 0:
        x += 1
        if x > 100000:
            print('Exceeded 100000 loops. Exiting.')
            break
        current_node = open_set.pop(0)
        closed_set.append(current_node)

        # Return the path if end point is reached
        if current_node == end_node:
            path = return_path(current_node)
            return path

        neighbours = self.neighbour(current_node.position)

        for nb in neighbours:

            new_node = Node(current_node, nb)

            # Checking if child is in closed list
            flag = False
            for closed_child in closed_set:
                if new_node == closed_child:
                    flag = True
                    break

            if flag:
```

```
                continue

            new_node_coordinates = (
                coordinates[new_node.position]['longitude'],
coordinates[new_node.position]['latitude'])
            new_node.g = current_node.g + dist_matrix[nb][current_node.position] / 100
            new_node.h = (new_node_coordinates[0] - end_node_coordinates[0]) ** 2 + (
                    new_node_coordinates[1] - end_node_coordinates[1]) ** 2
            new_node.f = new_node.g + new_node.h

            # Checking if child is already in open list and the new path is larger
            flag = False
            for open_child in open_set:
                if new_node == open_child and new_node.g > open_child.g:
                    flag = True
                    break

            if flag:
                continue

            i = 0
            if len(open_set) > 0:
                while i < len(open_set) and open_set[i].f < new_node.f:
                    i += 1
            open_set.insert(i, new_node)
        return False

    def generate(self, start, end):

        """Generate a new route between the start and end points.
        Args:
            start: The starting position of the route.
            end: The ending position of the route.
        Returns:
            A list containing the generated route.
        """

        self.start = start
        self.end = end
        route = self.astar()
        self.distance = route_distance(route)
        return route
```

visual.py

Contains methods to plot shuttle positions, pick-up and drop positions, and routes using different colors on the map.

```
from constants import image_coordinates
```

```
from PIL import Image, ImageDraw, ImageFont


class Visual:
    # Color constants used for drawing routes.
    start_color = (255, 255, 255)
    branch_color = [(0, 255, 0), (0, 255, 255)]
    current_route_color = [(173, 255, 47), (30, 144, 255)]
    old_route_color = [(8, 105, 114), (135, 206, 250)]
    drop_node_color = [(0, 100, 0), (180, 0, 180)]
    current_position_color = [(0, 250, 154), (123, 104, 238)]
    denied_request_color = [(255, 0, 0), (255, 0, 255)]

    def __init__(self, center, node_size=5, folder=None):

        """Initialize the Map Drawer.
        Args:
            center: Index of the hub.
            node_size: Size of the node to be drawn.
            folder: The name of the folder where the images are to be saved.
        """

        if folder is None:
            folder = 'random'
        self.center = center
        self.node_size = node_size
        self.folder = folder
        self.im = None
        self.draw = None
        self.initialize()

    def draw_denied_request(self, request):

        """Draw the nodes for a denied request.
        Args:
            request: A tuple containing the pickup and drop location.
        """

        node_size = 10
        for i in range(2):
            self.draw.ellipse((
                image_coordinates[request[i]][0] - node_size, image_coordinates[request[i]][1] -
node_size,
                image_coordinates[request[i]][0] + node_size,
                image_coordinates[request[i]][1] + node_size), fill=self.denied_request_color[i],
                outline=(0, 0, 0))

    def draw_initial_requests(self, requests):
```

```
        """Draw the nodes for the initial requests.
        Args:
            requests: The initial batch of drop requests.
        """

        for req in requests:
            self.draw.ellipse(
                (image_coordinates[req][0] - self.node_size, image_coordinates[req][1] -
self.node_size,
                 image_coordinates[req][0] + self.node_size, image_coordinates[req][1] +
self.node_size),
                fill=(255, 0, 0), outline=(0, 0, 0))
        self.draw.ellipse(
            (image_coordinates[self.center][0] - self.node_size, image_coordinates[self.center][1]
- self.node_size,
             image_coordinates[self.center][0] + self.node_size, image_coordinates[self.center][1]
+ self.node_size),
            fill=(255, 255, 255), outline=(0, 0, 0))

    def draw_nodes(self, shuttles):

        """Draw the nodes for a particular state of the shuttle.
        Args:
            shuttles: A list of objects containing the current shuttle objects.
        """

        for i in range(len(shuttles)):
            for req in shuttles[i].drop_order:
                self.draw.ellipse(
                    (image_coordinates[req[1]][0] - self.node_size, image_coordinates[req[1]][1] -
self.node_size,
                     image_coordinates[req[1]][0] + self.node_size, image_coordinates[req[1]][1] +
self.node_size),
                    fill=self.drop_node_color[i], outline=(0, 0, 0))
            self.draw.ellipse((image_coordinates[shuttles[i].current_position][0] - self.node_size,
                               image_coordinates[shuttles[i].current_position][1] - self.node_size,
                               image_coordinates[shuttles[i].current_position][0] + self.node_size,
                               image_coordinates[shuttles[i].current_position][1] + self.node_size),
                              fill=self.current_position_color[i], outline=(0, 0, 0))

        self.draw.ellipse(
            (image_coordinates[self.center][0] - self.node_size, image_coordinates[self.center][1]
- self.node_size,
             image_coordinates[self.center][0] + self.node_size, image_coordinates[self.center][1]
+ self.node_size),
            fill=(255, 255, 255), outline=(0, 0, 0))

    def draw_request_text(self, request):

        """Draw the current request text.
```

```
        Args:
            request: A tuple containing the pickup and drop location.
        """

        self.draw.text((30, 20), 'Request: ({}, {})'.format(request[0]+1, request[1]+1),
                       font=ImageFont.truetype("arial.ttf", 12), fill=(0, 0, 0, 255))

    def draw_route(self, route, color):

        """Draw a route on the map.
        Args:
            route: A list containing the route.
            color: Color of the route.
        """

        for i in range(1, len(route)):
            self.draw.line((image_coordinates[route[i - 1]][0], image_coordinates[route[i - 1]][1],
                            image_coordinates[route[i]][0], image_coordinates[route[i]][1]), color,
width=5)

    def draw_state(self, shuttles):

        """Draw the nodes for a particular state of the shuttle along with the current routes of the
shuttles.
        Args:
            shuttles: A list of objects containing the current shuttle objects.
        """

        for i in range(len(shuttles)):
            for j in range(1, len(shuttles[i].route)):
                self.draw.line((image_coordinates[shuttles[i].route[j - 1]][0],
                                image_coordinates[shuttles[i].route[j - 1]][1],
                                image_coordinates[shuttles[i].route[j]][0],
                                image_coordinates[shuttles[i].route[j]][1]), self.current_route_color[i],
width=5)
            for req in shuttles[i].drop_order:
                self.draw.ellipse(
                    (image_coordinates[req[1]][0] - self.node_size, image_coordinates[req[1]][1] -
self.node_size,
                     image_coordinates[req[1]][0] + self.node_size, image_coordinates[req[1]][1] +
self.node_size),
                    fill=self.drop_node_color[i], outline=(0, 0, 0))
            self.draw.ellipse((image_coordinates[shuttles[i].current_position][0] - self.node_size,
                               image_coordinates[shuttles[i].current_position][1] - self.node_size,
                               image_coordinates[shuttles[i].current_position][0] + self.node_size,
                               image_coordinates[shuttles[i].current_position][1] + self.node_size),
                              fill=self.current_position_color[i], outline=(0, 0, 0))

        self.draw.ellipse(
```

```
        (image_coordinates[self.center][0] - self.node_size, image_coordinates[self.center][1]
- self.node_size,
         image_coordinates[self.center][0] + self.node_size, image_coordinates[self.center][1]
+ self.node_size),
        fill=(255, 255, 255), outline=(0, 0, 0))

    def initialize(self):

        """Load the image and create the draw object."""

        self.im = Image.open("test-img.png")
        self.draw = ImageDraw.Draw(self.im)

    def save(self, title):

        """Save the image in a folder.
        Args:
            title: Name of the saved file.
        """

        self.im.save('{}/{}.png'.format(self.folder, title))

    def show(self):

        """Create a temporary image file and show the image."""

        self.im.show()
```

simulation.py

Contains methods to simulate a dynamic shared shuttle service for 30 iterations, along with a request handler.

```
from shuttle import Shuttle
from passenger import Passenger
from utility import generate_cluster_route
from visual import Visual
from routes import Route, join_route

import random


class Simulation:
    def __init__(self, m, n, center, total_nodes, clusters, requests, folder=None):

        """Initializing the simulation. The simulation will create an initial state and then run 30
iterations. In each iteration, the state of the shuttles will be displayed. A random request will
```

```
be generated for each iteration. The request will be checked for being serviceable. If
serviceable, the state of the shuttles is modified to accommodate the request.
        Args:
            m: No. of vehicles available for service.
            n: No. of initial passengers.
            center: Index of the hub.
            total_nodes: Total number of nodes in the problem.
            clusters: The clusters generated from the initial requests generated by the QUBO
solver.
            requests: The list of initial requests.
            folder: The name of the folder where the images are to be saved.
        """

        if folder is None:
            folder = 'random'
        self.m = m
        self.n = n
        self.center = center
        self.total_nodes = total_nodes
        self.clusters = clusters
        self.requests = requests
        self.folder = folder
        self.shuttles = []
        self.shuttle_index = {}
        self.initialization()

    def initialization(self):

        """Generating the initial state. In the initial state, the initial passengers are assigned to
the shuttles available and the corresponding routes are generated.
        """

        for i in range(self.m):
            # Initial shuttle and passenger initiation
            current_passengers = []
            path, drop_order = generate_cluster_route(self.center, self.requests, self.clusters[i])
            shuttle = Shuttle(path, drop_order)
            for drop_request in drop_order:
                current_passengers.append(Passenger(drop_request, shuttle.number))
            shuttle.set_passengers(current_passengers)
            self.shuttle_index[shuttle.number] = i
            self.shuttles.append(shuttle)
            shuttle.details()
            for passenger in current_passengers:
                passenger.details()

        # Displaying the initial state on map
        initial_state = Visual(self.center, folder=self.folder)
        initial_state.draw_state(self.shuttles)
        # initial_state.show()
```

```
        initial_state.save('Initial state')

    def simulate(self):

        """A simulation is run for thirty iterations. At each iteration, every shuttle is considered
to move to the next node in its route irrespective of the distance between the nodes. A single
request is generated per iteration, which is handled accordingly.
Args: iterations: The number of iterations the simulation will run for.
        """

        print('\n\t###Simulation Starting###\n')
        for i in range(iterations):
            print('\nIteration ', i + 1)
            for shuttle in self.shuttles:
                if shuttle.route != [] and shuttle.index < len(shuttle.route) - 1:
                    shuttle.index += 1
                    shuttle.current_position = shuttle.route[shuttle.index]
                for drop_request in shuttle.drop_order:
                    if drop_request == shuttle.pickup:
                        continue
                    if drop_request[1] == shuttle.current_position:
                        shuttle.drop_order.remove(drop_request)
                        shuttle.passengers.pop(0)
                        shuttle.occupancy -= 1
                        print('Passenger dropped off at {} by car {}'.format(shuttle.current_position,
shuttle.number))
                if not shuttle.drop_order:
                    shuttle.route = []
                    shuttle.index = -1
                if shuttle.service is True and shuttle.current_position == shuttle.pickup[0]:
                    shuttle.passenger_details()
                    shuttle.passengers.insert(shuttle.drop_order.index(shuttle.pickup),
                                              Passenger(shuttle.pickup, shuttle.number))
                    shuttle.occupancy += 1
                    print('Passenger picked up at {} by car {}'.format(shuttle.pickup[0],
shuttle.number))
                    shuttle.service = False
                    shuttle.pickup = None
            request = self.request_generator()
            self.request_handler(request, i)
            print('\n')
            for shuttle in self.shuttles:
                print(
                    'Number: {}\tNumber of Passengers: {}\t\tCurrent Position: {}\tRoute : {}\tDrop
order: {}\t'
                    'Service: {}\tPickup: {}'.format(
                        shuttle.number, shuttle.occupancy, shuttle.current_position, shuttle.route,
shuttle.drop_order,
                        shuttle.service, shuttle.pickup))
```

```
    def request_generator(self):

        """Generate a random request."""

        src = random.randrange(self.total_nodes)
        dest = random.randrange(self.total_nodes)
        while src == dest:
            dest = random.randrange(self.total_nodes)
        return src, dest

    def request_handler(self, request, iteration):

        """Handle the servicing of a generated request. If a suitable shuttle is available, the request is assigned to the most suitable shuttle, and it's route is recalculated to accommodate the pickup and drop of the passenger. If no suitable shuttles are available, the request is denied.
        Args:
            request: A tuple containing the pickup and drop location.
            iteration: The current iteration of the simulation.
        """

        iteration_state = Visual(self.center, folder=self.folder)
        iteration_state.draw_state(self.shuttles)
        print('Request generated: ', request)
        iteration_state.draw_request_text(request)

        suitable_shuttles = []
        route = Route()
        for shuttle in self.shuttles:
            occupancy = shuttle.occupancy
            src_pt = -1
            dest_pt = -1
            detour = 0
            pos = 0
            # Checking whether occupancy is not full and if the shuttle is already servicing another passenger.
            if shuttle.occupancy < 4 and shuttle.service is False:
                if shuttle.occupancy != 0:
                    pos = shuttle.route.index(shuttle.current_position)
                    shortest_distance = 999999
                    for i in range(pos, pos + 3):
                        if i >= len(shuttle.route):
                            break
                        point = shuttle.route[i]
                        route.generate(point, request[0])
                        print('{}\t{}\t{}\t{}'.format(i, point, request[0], route.distance))
                        if route.distance < 1000:
                            if route.distance > shortest_distance:
                                continue
                            src_pt = i
```

```
                shortest_distance = route.distance
            detour += shortest_distance

            if src_pt != -1:
                shortest_distance = 999999
                for j in range(pos, len(shuttle.route)):
                    route.generate(shuttle.route[j], request[1])
                    if route.distance < 1000:
                        if route.distance > shortest_distance:
                            continue
                        shortest_distance = route.distance
                        dest_pt = j
                detour += shortest_distance

            if src_pt != -1 and dest_pt != -1 and detour < 2000:
                suitable_shuttles.append((shuttle, (src_pt, dest_pt)))
        else:
            route.generate(shuttle.current_position, request[0])
            if route.distance <= 3000:
                suitable_shuttles.append((shuttle, (0, 1)))
    if len(suitable_shuttles) == 0:
        print('\nNo suitable shuttles')
        iteration_state.draw_denied_request(request)
        # iteration_state.show()
        iteration_state.save('Iteration_{}.png'.format(iteration + 1))
        return
    print('\nSuitable Shuttles: ')
    for shuttle in suitable_shuttles:
        route.generate(shuttle[0].current_position, request[0])
        print(
            'Distance: {}\t Number: {}\tNumber of Passengers: {}\t\tCurrent Position: {}\t
Route : {} '
            '\tDrop order: {}'.format(
                route.distance, shuttle[0].number, shuttle[0].occupancy,
shuttle[0].current_position,
                shuttle[0].route, shuttle[0].drop_order))
    route.generate(suitable_shuttles[0][0].current_position, request[0])
    closest_dist = route.distance
    closest = 0
    # selecting the closest shuttle from the list of suitable shuttles.
    for i in range(1, len(suitable_shuttles)):
        route.generate(suitable_shuttles[i][0].current_position, request[0])
        if route.distance < closest_dist:
            closest_dist = route.distance
            closest = i
    selected_shuttle = suitable_shuttles[closest]
    print('Selected Shuttle: ', suitable_shuttles[closest][0].number)
    if len(selected_shuttle[0].passengers) == 0:
        # Use spanning tree algo
        selected_shuttle[0].drop_order.append(request)
```

```
            temp = (route.generate(selected_shuttle[0].current_position, request[0]),
route.generate(*request))
            selected_shuttle[0].route = join_route(temp[0][:-1], temp[1])
            selected_shuttle[0].index = 0
            if selected_shuttle[0].current_position == request[0]:
                selected_shuttle[0].passengers.append(
                    Passenger(request, selected_shuttle[0].number))
                selected_shuttle[0].occupancy += 1
                print('Passenger picked up at {} by shuttle {}'.format(request[0],
selected_shuttle[0].number))

                # draw new route
                iteration_state.draw_route(temp[1],
                                           iteration_state.current_route_color[self.shuttle_index[
                                               selected_shuttle[0].number]])
            else:
                selected_shuttle[0].pickup = request
                selected_shuttle[0].service = True
                iteration_state.draw_route(temp[0],
                                           iteration_state.branch_color[self.shuttle_index[selected_shuttle[0].
number]])
                iteration_state.draw_route(temp[1],
                                           iteration_state.current_route_color[self.shuttle_index[
                                               selected_shuttle[0].number]])
        else:
            src_pt = selected_shuttle[1][0]
            dest_pt = selected_shuttle[1][1]
            drop_list = [node for node in selected_shuttle[0].drop_order]
            flag = False
            insert_index = 0
            for i in range(len(drop_list)):
                if dest_pt < selected_shuttle[0].route.index(drop_list[i][1]):
                    drop_list.insert(i, request)
                    insert_index = i
                    flag = True
                    break
            if not flag:
                insert_index = len(drop_list)
                drop_list.append(request)
            selected_shuttle[0].drop_order = [order for order in drop_list]
            new_route = selected_shuttle[0].route[:src_pt]

            # draw old route
            iteration_state.draw_route(selected_shuttle[0].route,
                                       iteration_state.old_route_color[self.shuttle_index[selected_shuttle[0].
number]])

            branch_route = route.generate(selected_shuttle[0].route[src_pt], request[0])
            new_route = join_route(new_route, branch_route[:-1])
```

```
            drop_first = False
            for i in range(insert_index):
                if selected_shuttle[0].route.index(drop_list[i][1]) > src_pt:
                    drop_first = True
                    break
            drop_list = drop_list[i if drop_first else insert_index:]
            join_index = len(new_route)
            new_route = join_route(new_route, route.generate(request[0], drop_list[0][1])[:-1])
            for i in range(1, len(drop_list)):
                new_route = join_route(new_route, route.generate(drop_list[i - 1][1],
drop_list[i][1])[:-1])
            new_route.append(drop_list[-1][1])
            selected_shuttle[0].route = new_route
            # draw new route
            iteration_state.draw_route(new_route[join_index:],
                                       iteration_state.current_route_color[self.shuttle_index[
                                           selected_shuttle[0].number]])

            if selected_shuttle[0].current_position == request[0]:
                selected_shuttle[0].passengers.insert(insert_index, Passenger(request,
                                                              selected_shuttle[0].number))
                selected_shuttle[0].occupancy += 1
                print('Passenger picked up at {} by shuttle {}'.format(request[0],
selected_shuttle[0].number))
            else:
                selected_shuttle[0].pickup = request
                selected_shuttle[0].service = True

                # draw branch route
                iteration_state.draw_route(branch_route,
                                           iteration_state.branch_color[self.shuttle_index[selected_shuttle[0].
number]])
        iteration_state.draw_nodes(self.shuttles)
        # iteration_state.show()
        iteration_state.save('Iteration_{}.png'.format(iteration + 1))
```

main.py

The main program from which the different modules are called to execute the application.

```
import numpy as np
import utility
import random
```

```
from qubo_solver import QuboSolver
from routes import Route
from visual import Visual
from PIL import Image, ImageDraw
from simulation import Simulation


"""Definining the number of shuttles, the initial number of passengers, the central node, and
the total number of nodes.
    n: Initial number of passengers.
    m: Number of shuttles.
    center: Index of central node.
    total_nodes: Total number of nodes.
"""

n = 5
m = 2
center = 86
total_nodes=122

# Generate the initial set of requests.
requests = utility.generate_initial_requests(n, center, total_nodes)
print(requests)
# Show the requests generated on the map.
initial_request_view = Visual(center, node_size=10)
initial_request_view.draw_initial_requests(requests)
initial_request_view.show()

# Generate the VRP instance
instance, xc, yc = utility.generate_vrp_instance(n, center, requests)

# Initialize the QUBO instance and solve
qs = QuboSolver(n, m, instance)
qs.solve()

# Visualize the solution as a graph
qs.visualize(xc, yc)

# Retrieve the clusters from the solution
clusters = qs.get_clusters()

# Create the simulation instance
sim = Simulation(m, n, center, total_nodes, clusters, requests, folder='solution')

# Run the simulation for 30 iterations
sim.simulate(30)
```

5.2 Unit Testing

5.2.1. Test Strategy

The test strategy for this project involves manual testing during various stages of the development cycle. Since the iterative model of software development has been followed, where a relatively basic implementation of a subset of the requirement specification progresses through iterative improvisations until the entire system is completed, both unit testing and integration testing (functional testing) are required after each improvisation. Non-functional testing has been performed after the final deployment.

5.2.2 Assumptions

The capacitated shuttle routing problem is solved by keeping in mind the following constraints:

- The initial number of clients in the depot = 5
- Number of shuttles = 2
- The capacity of each shuttle = 4
- Only nodes marked in the map are considered.
- In a unit time instance, a shuttle moves from one node to the next node irrespective of the distance between the two nodes.
- While a shuttle is en route to service a passenger, it will not accept other service requests even if the boarding location is nearby.

5.2.3 Unit Testing

A brief description of different modules/components of the implementation is mentioned below, along with their input and output instances, as part of the project's unit testing after the final iteration.

<u>VRP instance generation</u>

It generates an instance of the shuttle routing problem, in agreement with the constraints.

Function signature -

generate_vrp_instance(n, center, requests) : instance, xc, yc

where

n: No. of nodes excluding the hub.

center: Index of the node considered as the hub.

requests: The initial batch of drop requests.

Returns:

xc: a list of (n + 1) x coordinates,

yc: a list of (n + 1) y coordinates,

instance: an (n + 1) x (n + 1) NumPy array as the cost matrix.

Input:
n = 5
center = 86
requests = [24, 10, 40, 111, 121]

Output:
instance = [
[0.   50.0393134   92.0705667   50.1057419   60.81787326
  179.56707486]
 [50.0393134   0.   44.18660808 28.01566795 217.90432711
  412.62742124]
 [92.0705667   44.18660808   0.   6.33445004 282.73393999
  404.26432554]
 [50.1057419   28.01566795   6.33445004   0.   206.62493985
  327.25651104]
 [60.81787326 217.90432711 282.73393999 206.62493985   0.
   57.82263815]
 [179.56707486 412.62742124 404.26432554 327.25651104 57.82263815
    0.]]
xc= [0.   1.90355   8.262975 6.1013   -3.85519 -0.11506]
yc= [0.   6.812915   4.87789   3.588855 -6.77904 -13.39977]

<u>Qubo instance generation</u>

It generates the QUBO form of a particular instance of the shuttle routing problem in order to make it solvable by a quantum annealer.

Function signature -
qs = QuboSolver(n, m, instance)
where
   n: No. of nodes in the problem (excluding the hub).
   m: No. of vehicles available for service.
   instance: (n_clients + 1) x (n_clients + 1) matrix describing the cost of moving from node i to node j.

Input:
n = 5
m = 2
instance = [
[0.   50.0393134   92.0705667   50.1057419   60.81787326
  179.56707486]

```
[ 50.0393134    0.          44.18660808 28.01566795 217.90432711
  412.62742124]
[ 92.0705667   44.18660808   0.           6.33445004 282.73393999
  404.26432554]
[ 50.1057419   28.01566795   6.33445004   0.         206.62493985
  327.25651104]
[ 60.81787326 217.90432711 282.73393999 206.62493985   0.
   57.82263815]
[179.56707486 412.62742124 404.26432554 327.25651104 57.82263815
    0.]]
```

Output: Creates a QuboSolver object

<u>Qubo Solver</u>

Solves the QUBO problem corresponding to a particular instance of the shuttle routing problem.

Function signature -

qs.solve()

This function redirects the control to the solver where it is solved using Leap Hybrid Sampler.

Input: Not Applicable

Output: A solution to the QUBO problem created.

```
[[[0. 0. 0. 1. 1.]
  [0. 0. 1. 0. 0.]
  [0. 1. 0. 0. 0.]
  [1. 0. 0. 0. 0.]
  [0. 0. 0. 0. 0.]
  [0. 0. 0. 0. 0.]]

 [[1. 1. 1. 0. 0.]
  [0. 0. 0. 0. 0.]
  [0. 0. 0. 0. 0.]
  [0. 0. 0. 0. 0.]
  [0. 0. 0. 0. 1.]
  [0. 0. 0. 1. 0.]]]
```

For cluster 1, nodes A, B, and C are chosen.
For cluster 2, nodes D, and E are chosen.

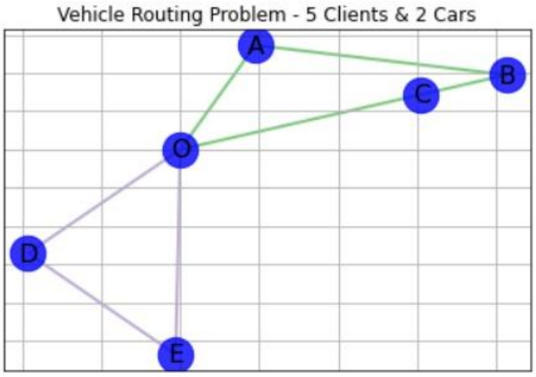

Figure 5. QUBO solution as Cluster graph

Route generation

Generates the optimal route for a car instance given a starting point and ending point using A* path search algorithm.

Function signature -

generate(self, start, end): route

where

start: the starting point of the journey
end: the ending point of the journey
route: the path between the start and end nodes

Input:
start = 13, end = 53

Output:
route = [13, 18, 43, 52, 53]

Initial service ordering

Determines the route and the drop order by which the commuters of each shuttle will be serviced from the central hub, given a set of destinations.

Function signature -

generate_cluster_route(center, requests, cluster): path, drop_order

where

center: the central hub
cluster: the set of destinations that will be serviced by a particular shuttle
requests: The initial batch of drop requests.
path: the sequence of nodes along which the shuttle moves
drop_order: the ordered pair denoting the boarding point and alighting point of each passenger in the shuttle

Input:

center = 86,

cluster = [2, 3, 5, 4]

Output:

path = [86, 85, 90, 93, 103, 108, 112, 113, 114, 106, 105, 82, 74, 51, 52, 43, 18, 13, 12, 11, 10]

drop_order = [(86, 93), (86, 113), (86, 13), (86, 10)]

<u>Request handler</u>

Assigns the most suitable shuttle amongst the available shuttles whenever a request is generated, and the state of the shuttle is changed to pick-up. If there is no such suitable shuttle, the request is denied.

Function signature -

request_handler(request, iteration)

where

request: A tuple containing the pickup and drop location.
iteration: The current iteration of the simulation.

Case I:

Input:

request = (66, 28)

iteration = 2

Output:

Suitable Shuttles:

Distance: 420

Number: WB24S1026

Number of Passengers: 2

Current Position: 97

Route: [86, 87, 97, 100, 98, 99, 98, 65, 64, 31, 30, 29]

Drop order: [(86, 99), (86, 29)]
Selected Shuttle: WB24S1026

Number: WB78C2237
Number of Passengers: 4
Current Position: 90
Route: [86, 85, 90, 85, 84, 71, 55, 41, 20, 21, 10, 11, 12, 13]
Drop order: [(86, 20), (86, 21), (90, 10), (86, 13)]
Service: False   Pickup: None

Number: WB24S1026
Number of Passengers: 2
Current Position: 97
Route: [86, 87, 97, 66, 65, 98, 99, 98, 65, 64, 31, 30, 29, 28]
Drop order: [(86, 99), (86, 29), (66, 28)]
Service: True Pickup: (66, 28)

Figure 6. Accepted request Map Representation

Case II:
Input:
request = (77, 28)
iteration = 3

Output:
No suitable shuttles

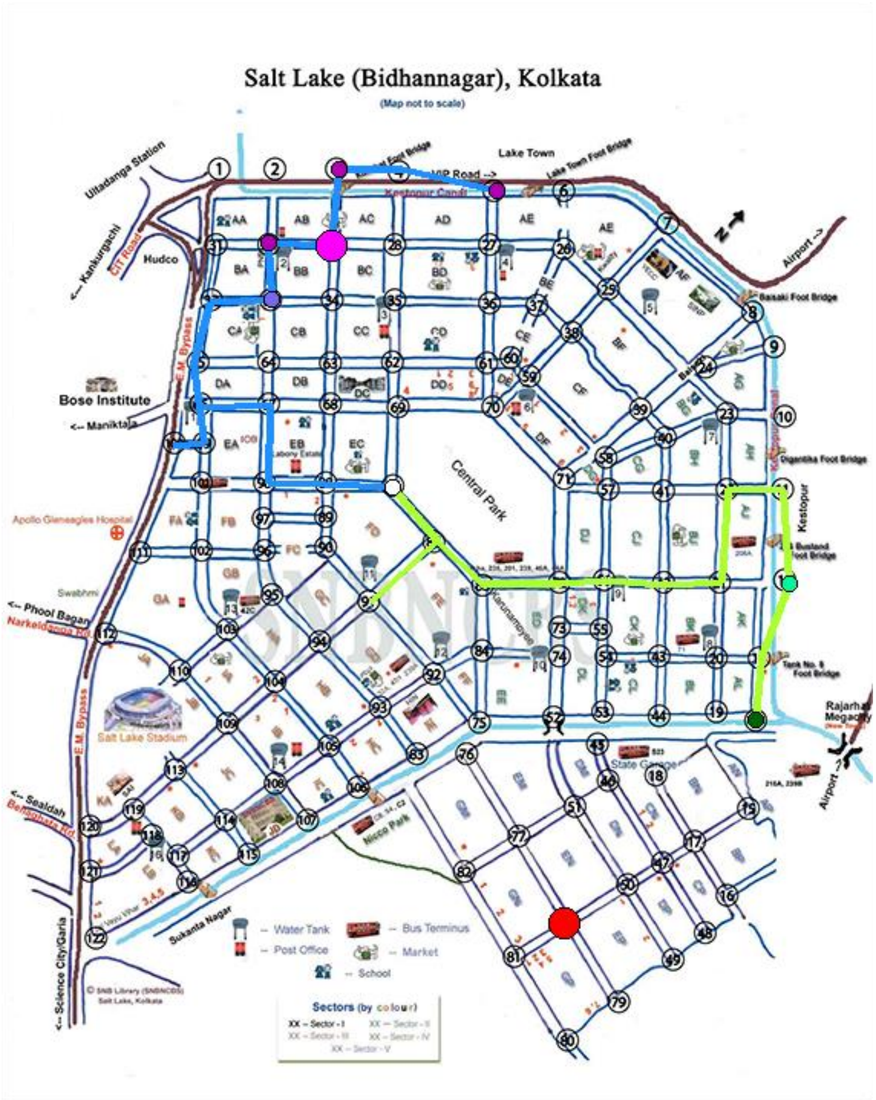

Figure 7. Declined request Map Representation

Initial car state generator

Generates the shuttle numbers and assigns passengers to shuttles.

Input: Not Applicable.

Output:

Number: WB95S4330
Number of Passengers: 3
Current Position: 86
Route: [86, 68, 69, 58, 37, 24, 37, 38, 39, 40, 21, 10]
Drop order: [(86, 24), (86, 40), (86, 10)]

ID: P759XV3663 Request Info: (86, 24) Shuttle Number: WB95S4330
ID: P852IO8829 Request Info: (86, 40) Shuttle Number: WB95S4330
ID: P881CY6956 Request Info: (86, 10) Shuttle Number: WB95S4330

Number: WB75Y6796
Number of Passengers: 2
Current Position: 86
Route: [86, 87, 97, 96, 95, 101, 110, 111, 119, 120, 121]
Drop order: [(86, 111), (86, 121)]

ID: P257TB1978 Request Info: (86, 111) Shuttle Number: WB75Y6796
ID: P958ZK9651 Request Info: (86, 121) Shuttle Number: WB75Y6796

5.3 System Testing

Test Data

This project uses the map of the Saltlake area to test the functional correctness of the proposed car routing algorithm. Some specific points on the map have been selected as drop locations, and these points are highlighted in the map given below.

Figure 8. Map

Legend

The legend of the colors used to represent details on the map.

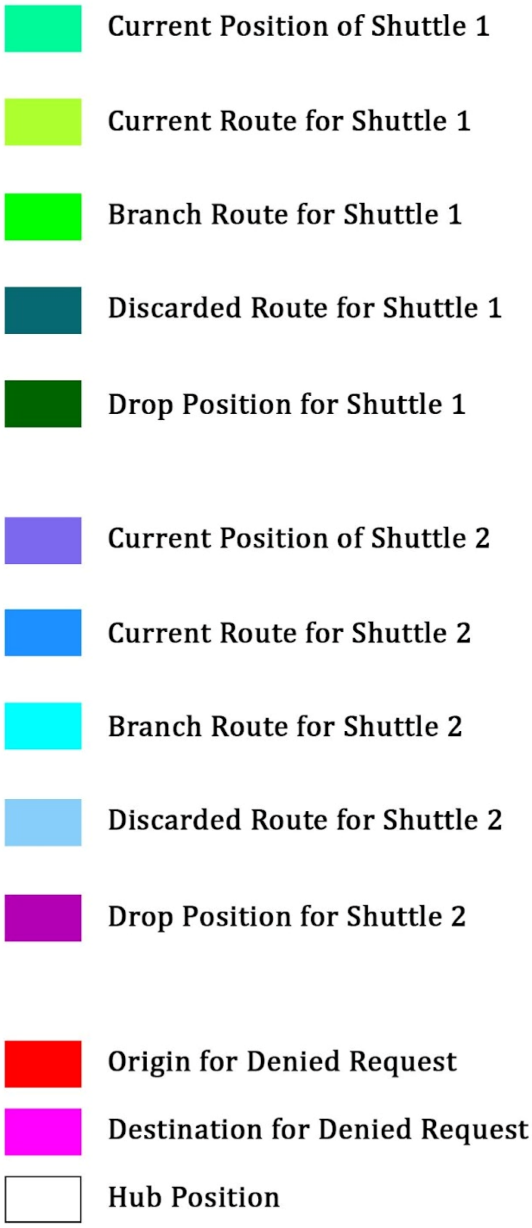

Figure 9. Legend

Test Case

Initial requests: 30, 21, 14, 100, 22

Clusters: (21, 14, 22) and (30, 100)

Figure 10. Initial State

Initial State

Number: WB78C2237

Number of Passengers: 3

Current Position: 86

Route: [86, 85, 84, 71, 55, 41, 20, 21, 20, 19, 18, 13]

Drop order: [(86, 20), (86, 21), (86, 13)]

Passengers:

ID: P019KP9860 Request Info: (86, 20) Shuttle Number: WB78C2237

ID: P504HM8618 Request Info: (86, 21) Shuttle Number: WB78C2237

ID: P009CS6541 Request Info: (86, 13) Shuttle Number: WB78C2237

Number: WB24S1026

Number of Passengers: 2

Current Position: 86

Route: [86, 87, 97, 100, 98, 99, 98, 65, 64, 31, 30, 29]

Drop order: [(86, 99), (86, 29)]

Passengers:

ID: P921TZ7006 Request Info: (86, 99) Shuttle Number: WB24S1026

ID: P376VI3525 Request Info: (86, 29) Shuttle Number: WB24S1026

Figure 11. Iteration 2 (Passenger pickup)

Iteration 2

Passenger picked up at 90 by car WB78C2237

Request generated: (66, 28)

Suitable Shuttles:

Distance: 420 Number: WB24S1026

Number of Passengers: 2 Current Position: 97

Route: [86, 87, 97, 100, 98, 99, 98, 65, 64, 31, 30, 29]

Drop order: [(86, 99), (86, 29)]

Selected Shuttle: WB24S1026

Number: WB78C2237

Number of Passengers: 4

Current Position: 90

Route: [86, 85, 90, 85, 84, 71, 55, 41, 20, 21, 10, 11, 12, 13]

Drop order: [(86, 20), (86, 21), (90, 10), (86, 13)]

Service: False Pickup: None

Number: WB24S1026

Number of Passengers: 2

Current Position: 97

Route: [86, 87, 97, 66, 65, 98, 99, 98, 65, 64, 31, 30, 29, 28]

Drop order: [(86, 99), (86, 29), (66, 28)]

Service: True Pickup: (66, 28)

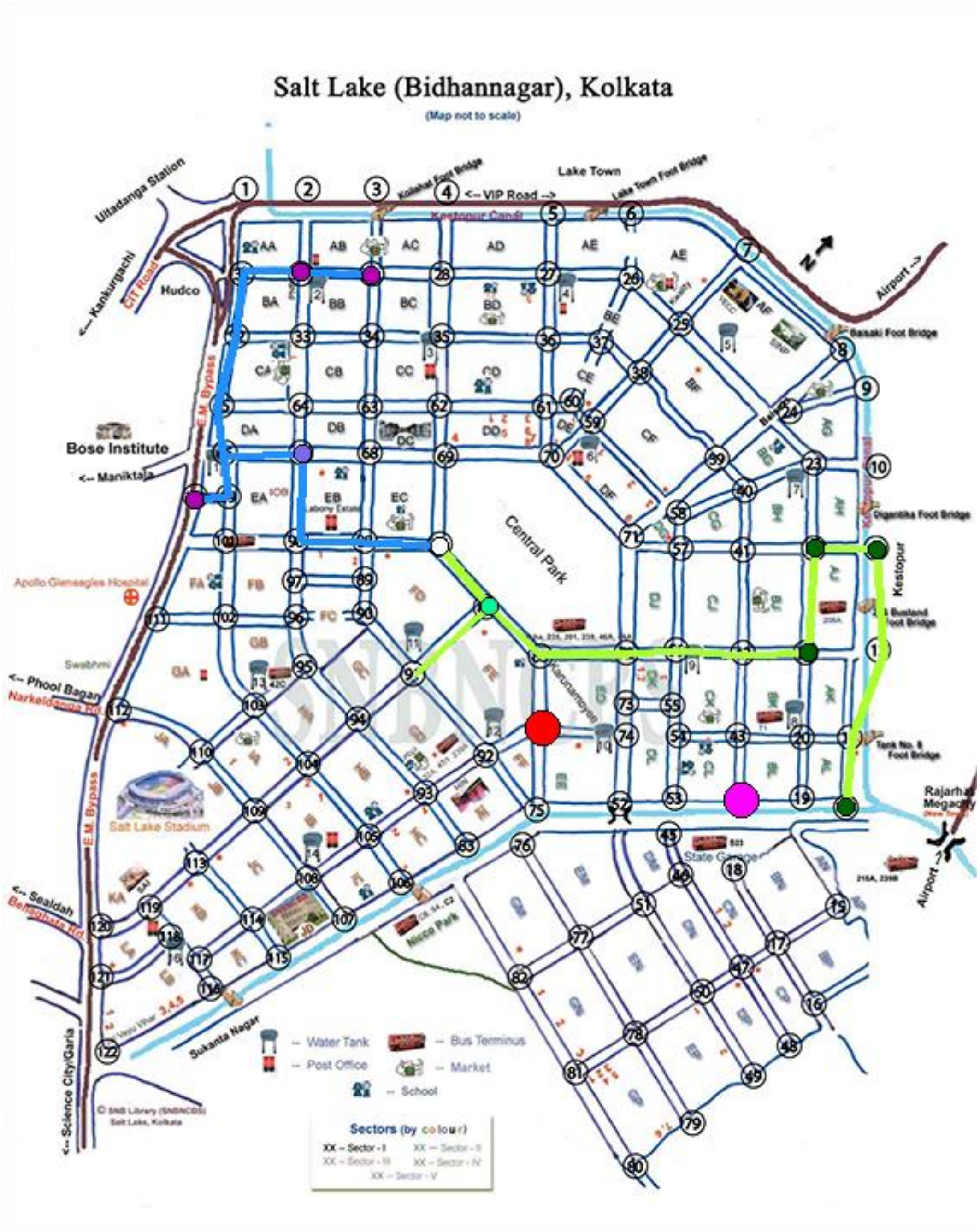

Figure 12. Iteration 3 (Denied Request)

Iteration 3

Passenger picked up at 66 by car WB24S1026

Request generated: (83, 43)

No suitable shuttles

Number: WB78C2237

Number of Passengers: 4

Current Position: 85

Route: [86, 85, 90, 85, 84, 71, 55, 41, 20, 21, 10, 11, 12, 13]

Drop order: [(86, 20), (86, 21), (90, 10), (86, 13)]

Service: False   Pickup: None

Number: WB24S1026

Number of Passengers: 3

Current Position: 66

Route: [86, 87, 97, 66, 65, 98, 99, 98, 65, 64, 31, 30, 29, 28]

Drop order: [(86, 99), (86, 29), (66, 28)]

Service: False   Pickup: None

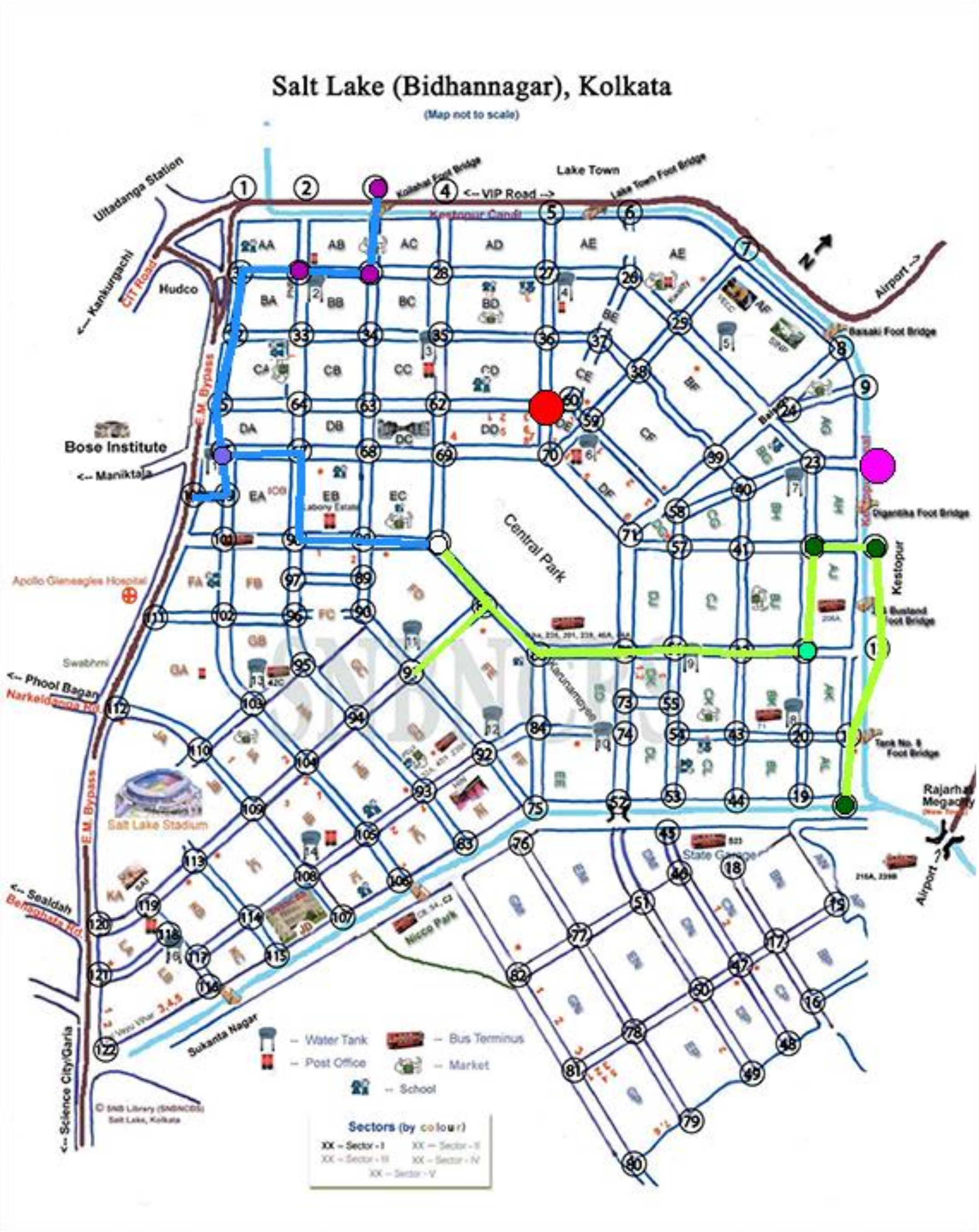

Figure 13. Iteration 8 (Passenger Drop)

Iteration 8

Passenger dropped off at 20 by car WB78C2237

Request generated: (60, 9)

No suitable shuttles

Number: WB78C2237

Number of Passengers: 3

Current Position: 20

Route: [86, 85, 90, 85, 84, 71, 55, 41, 20, 21, 10, 11, 12, 13]

Drop order: [(86, 21), (90, 10), (86, 13)]

Service: False   Pickup: None

Number: WB24S1026

Number of Passengers: 3

Current Position: 65

Route: [86, 87, 97, 66, 65, 98, 99, 98, 65, 64, 31, 30, 29, 28, 2]

Drop order: [(86, 29), (66, 28), (99, 2)]

Service: False   Pickup: None

Figure 14. Iteration 19 (Passenger pickup from idle state)

Iteration 19

Request generated: (26, 70)

Suitable Shuttles:

Distance: 308

Number: WB24S1026

Number of Passengers: 0

Current Position: 4

Route: []

Drop order: []

Selected Shuttle: WB24S1026

Number: WB78C2237

Number of Passengers: 0

Current Position: 21

Route: [40, 21, 22, 23, 24, 6]

Drop order: [(22, 6)]

Service: True   Pickup: (22, 6)

Number: WB24S1026

Number of Passengers: 0

Current Position: 4

Route: [4, 26, 35, 60, 69, 70]

Drop order: [(26, 70)]

Service: True   Pickup: (26, 70)

Test Report Summary

Project Details	
Project Name	Quantum Ridesharing
Project Description	A capacitated vehicle routing algorithm implemented in a quantum system.
Project Duration	Start Date – 12/08/2021 End Date – 05/05/2022
Test Summary	
Test Cycles	Unit Testing ✓ Integration Testing ✓ System Testing ✓
Number of test cases executed	30
Number of test cases passed	30
Number of test cases failed	0

6. RESULTS AND DISCUSSION

The capacitated vehicle routing problem (CVRP) is a VRP in which vehicles with restricted carrying capacity must pick up or deliver commuters/things at many locations. The objective is to pick up or transport the things at the lowest possible cost while never exceeding the capacity of the vehicles. This project tries to simulate a CVRP with a variable number of passengers travelling to different locations from a central hub with a fixed number of shuttles, using Quadratic Unconstrained Binary Optimization (QUBO) implemented via Quantum Annealing on D-Wave systems.

Given a CVRP problem instance, the proposed algorithm will help construct suitable clusters of passengers in a very short period of time. The algorithm further works on the clusters created to generate the most optimal routes. The clusters thus obtained are analyzed using a greedy approach to determine the service ordering. A simulation is initialized, and the service ordering is assigned to the respective shuttle. The simulation is executed for thirty iterations. For each iteration, a request is generated. A request handler is called to determine whether the request is serviceable or not. If the request is not serviceable, the request is declined. Otherwise, shuttles which can service the request are determined. Out of the suitable shuttles, the best-suited shuttle is chosen to service the request. The chosen shuttle's path is modified to pickup the new passenger and the shuttle is set to pickup mode. While a shuttle is in pickup mode, it will not service any other incoming requests, whether it is suitable or not. The pickup mode is turned off once the passenger is picked up. The shuttle then continues its journey using the new route generated to accommodate the picked up passenger. For each iteration, the details of the corresponding state are displayed using logs as well as by drawing on the map.

The strength of the proposed algorithm is the utilization of QUBO for clustering the initial batch of requests. Although the simulation is run on limited capacity due to hardware limitations, with appropriate hardware, it would be possible to drastically reduce the time for clustering an overwhelming amount of data as compared to any classical approach.

7. CONCLUSION

Using a quantum approach from clustering is an efficient approach due to the capabilities of fasters calculations on overwhelming amounts of data. The project described is this paper is still in a preliminary research state due to limitations of quantum hardware. The true potential of this project can be measured when quantum computers are able to handle large amounts of data similar to modern classical computers. Till then, the effectiveness of the results proposed by this project will remain much abstract.

7.1 Limitations

- The project currently faces a hardware limitation in the form of the number of qubits available in contemporary gate-based quantum computers. They can only answer the simplest forms of the VRP. According to the model utilized by this project, the number of commuters should not exceed 5, and the number of shuttles should not exceed 2.
- Quantum run time is currently dominated by embedding time due to hardware constraints.
- Clustering as done by QUBO may not always be the most optimal one and may need to be further optimized.

7.2 Future Scope

- As quantum computers become mainstream in the upcoming few years, the project can be applied in practical scenarios to cater to thousands of service requests per instance of time, generating optimal routes and minimizing cost as well as waiting time of the commuters as well as drivers.
- The algorithm can be improved to handle multiple dynamic service requests in a particular instance of time.
- Improvisations can be made to make it possible to accept service requests from other passengers while in the process of picking up a passenger.

REFERENCES

[1] Laporte, G., and Semet, F. (2002). “Classical heuristics for the capacitated VRP,” in The Vehicle Routing Problem, eds P. Toth and D. Vigo (Philadelphia, PA: Society for Industrial and Applied Mathematics), 109–128. doi: 10.1137/1.9780898718515.ch5

[2] Groër C., Golden B., and Wasil E. (2010). A library of local search heuristics for the vehicle routing problem. Math. Prog. Comput. 2, 79–101. doi: 10.1007/s12532-010-0013-5

[3] A. Crispin and A. Syrichas, "Quantum Annealing Algorithm for Vehicle Scheduling," 2013 IEEE International Conference on Systems, Man, and Cybernetics, 2013, pp. 3523-3528, doi: 10.1109/SMC.2013.601.

[4] Billy E. Gillett, Leland R. Miller, (1974) A Heuristic Algorithm for the Vehicle-Dispatch Problem. Operations Research 22(2):340-349. https://doi.org/10.1287/opre.22.2.340

[5] Fisher, M. L., and Jaikumar, R. (1981). A generalized assignment heuristic for vehicle routing. Networks 11, 109–124. doi: 10.1002/net.3230110205

[6] Bramel, J., and Simchi-Levi, D. (1995). A location based heuristic for general routing problems. Operat. Res. 43, 649–660. doi: 10.1287/opre.43.4.649

[7] Rieffel, E. G., Venturelli, D., O’Gorman, B., Do, M. B., Prystay, E. M., and Smelyanskiy, V. N. (2015). A case study in programming a quantum annealer for hard operational planning problems. Quant. Inform. Process. 14, 1–36. doi: 10.1007/s11128-014- 0892-x

[8] Tran, T. T., Do, M., Rieffel, E. G., Frank, J., Wang, Z., O’Gorman, B., et al. (2016). “A hybrid quantum-classical approach to solving scheduling problems,” in Ninth Annual Symposium on Combinatorial Search (Tarrytown, NY).

[9] Haddar, B., Khemakhem, M., Hanafi, S., and Wilbaut, C. (2016). A hybrid quantum particle swarm optimization for the multidimensional knapsack problem. Eng. Appl. Artif. Intell. 55, 1–13. doi: 10.1016/j.engappai.2016.05.006

[10] Chancellor, N. (2017). Modernizing quantum annealing using local searches. N. J. Phys. 19:023024. doi: 10.1088/1367-2630/aa59c4

[11] Karp, R. M. (1972). “Reducibility among combinatorial problems,” in Complexity of Computer Computations, eds R. E. Miller, J. W. Thatcher, and J. D. Bohlinger (Boston, MA: Springer), 85–103.

[12] Feld, Sebastian & Roch, Christoph & Gabor, Thomas & Seidel, Christian & Neukart, Florian & Galter, Isabella & Mauerer, Wolfgang & Linnhoff-Popien, Claudia. (2019). A Hybrid Solution Method for the Capacitated Vehicle Routing Problem Using a Quantum AnnealerData_Sheet_1.pdf. Frontiers in ICT. 6. 10.3389/fict.2019.00013.

[13] Gueorguiev V. et al., An Exploration of the Vehicle Routing Problem, https://github.com/VGGatGitHub/QOSF-cohort3, last accessed: 20/12/2021

GLOSSARY

API	Application Programming Interface
BQM	Binary Quadratic Model
CVRP	Capacitated Vehicle Routing Problem
GAP	General Assignment Problem
IDE	Integrated Development Environment
KP	Knapsack Problem
QP	Quadratic Program
QUBO	Quadratic Unconstrained Binary Optimization
SDK	Software Development Kit
VRP	Vehicle Routing Problem