```diff
+ PHASE                    DATE
- Project Phase 1 : September 08, 2024
- Project Phase 2 : September 23, 2024
- Project Phase 3 : September 29, 2024
- Project Phase 4 : October 6, 2024
+ Project Phase 5 : ... Editing  ...
```

> Samrat Baral
> Algorithm and Data Structures

Organ Matching and Donation Networks

Project Phase 1 Deliverable: Data Structure Design and Implementation

Application Context

Organ donation is a crucial area in healthcare, potentially saving countless lives. The challenge is to efficiently match donors with recipients based on various criteria, including blood type, HLA matching, geographic proximity, and urgency. An effective algorithm can leverage real-time data for optimal matching, ensuring timely transplants as the demand for organs increases.

Design

The architecture of the donation matching system needs to prioritize fast lookups and manage urgency and optimal matches based on location.

Donor and Recipient Requirements

- **Authentication**: Users must verify their identities using medical records, adhering to HIPAA regulations and relevant laws to ensure secure access to health information.

Potential Input Requirements

1. Blood Type
2. Human Leukocyte Antigen (HLA)
3. Geographic Proximity
4. Urgency
5. HCT levels (optional)

Potential Data Structures

1. **Priority Queue (Heap)**

    - **Insertion/Deletion**: O(log n)
    - Consider using a Fibonacci heap for frequent insertions with an amortized time of O(1).

2. **Hash Table/Maps**

- Maps blood types to donor/recipient lists.
- **Time Complexity**: O(1)

3. **Binary Tree (Balanced Tree or Red-Black Tree)**

   - For storing user information and urgency levels.
   - **Time Complexity**: O(log n)

4. **Graphs**

   - Use Dijkstra's algorithm for finding optimal paths based on geographic data.
   - **Time Complexity**: O(log n) + O(V + E log V)

Key Ideas on Data Structures

Priority Queue

```
class Recipient:
    def __init__(self, id, blood_type, urgency):
        self.id = id
        self.blood_type = blood_type
        self.urgency = urgency

    def __lt__(self, other):
        return self.urgency > other.urgency  # Higher urgency gets higher priority

class UrgencyQueue:
    def __init__(self):
        self.heap = []

    def add_recipient(self, recipient):
        heapq.heappush(self.heap, recipient)

    def get_highest_priority(self):
        return heapq.heappop(self.heap) if self.heap else None
```

Hash Table

```
class Donor:
    def __init__(self, id, blood_type, hla_type):
        self.id = id
        self.blood_type = blood_type
        self.hla_type = hla_type

class DonorHashMap:
    def __init__(self):
        self.hash_table = {blood_type: [] for blood_type in ["A", "B", "AB", "O"]}

    def add_donor(self, donor):
        self.hash_table[donor.blood_type].append(donor)
```

```python
    def get_donors_by_blood_type(self, blood_type):
        return self.hash_table.get(blood_type, [])
```

Binary Tree (AVL Tree)

```python
class AVLNode:
    def __init__(self, recipient):
        self.recipient = recipient
        self.height = 1
        self.left = None
        self.right = None

class AVLTree:
    def __init__(self):
        self.root = None

    def insert(self, recipient):
        # Insertion logic goes here, including balancing the tree
        pass
```

Architecture: Estimation and Potential Requirements

1. **Donor and Recipient Entry**: Incorporate privacy measures and smart contracts.
2. **Matching Criteria**: Implement real-time matching, considering unmatched donors.
3. **Network Security**: Leverage blockchain technology for secure data handling.
4. **Kidney Paired Exchange (KPE)**: Utilize KPE algorithms to optimize organ exchanges.
5. **Compliance**: Adhere to ethical and regulatory standards.
6. **Machine Learning**: Implement predictive models for optimal donor selection.

Python Library Implementation

- `heapq`: For priority queues.
- `sortedcontainers`: For sorted data structures.
- `bintree`: For balanced trees.
- `NetworkX`: For graph-related operations.
- `SQLAlchemy`: For database interactions.
- `scikit-learn`: For machine learning implementations.
- `NumPy` and `Pandas`: For data manipulation.
- `schedule`: For scheduling periodic tasks.

Potential Challenges and Limitations

- **Scalability**: Addressing hash table collisions and ensuring efficient lookups.
- **Geographic Limitations**: Potentially utilizing AI models for routing.
- **Real-Time Updates**: Managing updates without excessive computational costs.
- **Healthcare Compliance**: Navigating evolving regulations.

- **Data Sources**: Automating data updates from various APIs while ensuring data integrity.

Python API Use Case

```
import requests
import json

api_endpoint = "https://api.example.com/donors"
headers = {
    "Authorization": "Bearer YOUR_API_KEY",
    "Content-Type": "application/json"
}

def update_data():
    response = requests.get(api_endpoint, headers=headers)
    if response.status_code == 200:
        data = response.json()
        save_to_database(data)
    else:
        print(f"Failed to fetch data: {response.status_code}")

def save_to_database(data):
    # Implement database saving logic here
    pass

if __name__ == "__main__":
    update_data()
    # Schedule the update function to run periodically
```

Demonstration of Key Operations

The provided test script demonstrates how to add users, establish product relationships, and generate recommendations. Running this script will output the top product recommendation based on the scores assigned.

```
import heapq
from collections import defaultdict

# Recipient class representing the recipient details
class Recipient:
    def __init__(self, id, blood_type, urgency):
        self.id = id
        self.blood_type = blood_type
        self.urgency = urgency

    def __lt__(self, other):
        return self.urgency > other.urgency  # Higher urgency gets higher priority

# Priority queue for managing recipient urgency
```

```
class UrgencyQueue:
    def __init__(self):
        self.heap = []

    def add_recipient(self, recipient):
        heapq.heappush(self.heap, recipient)

    def get_highest_priority(self):
        return heapq.heappop(self.heap) if self.heap else None

# Donor class representing the donor details
class Donor:
    def __init__(self, id, blood_type, hla_type):
        self.id = id
        self.blood_type = blood_type
        self.hla_type = hla_type

# Hash table to manage donors by blood type
class DonorHashMap:
    def __init__(self):
        self.hash_table = defaultdict(list)

    def add_donor(self, donor):
        self.hash_table[donor.blood_type].append(donor)

    def get_donors_by_blood_type(self, blood_type):
        return self.hash_table.get(blood_type, [])

# Simple matching algorithm based on blood type and urgency
def match_donors_to_recipients(donor_map, urgency_queue):
    matches = []
    while urgency_queue.heap:
        recipient = urgency_queue.get_highest_priority()
        donors = donor_map.get_donors_by_blood_type(recipient.blood_type)

        if donors:
            matched_donor = donors.pop(0)  # Get the first available donor
            matches.append((recipient.id, matched_donor.id))
        else:
            print(f"No suitable donor found for recipient {recipient.id} with
blood type {recipient.blood_type}.")

    return matches

# Test script demonstrating functionality
if __name__ == "__main__":
    # Initialize data structures
    donor_map = DonorHashMap()
    urgency_queue = UrgencyQueue()

    # Adding donors
    donor1 = Donor(id="donor1", blood_type="A", hla_type="HLA-A1")
    donor2 = Donor(id="donor2", blood_type="B", hla_type="HLA-B1")
    donor3 = Donor(id="donor3", blood_type="A", hla_type="HLA-A2")
```

```
    donor_map.add_donor(donor1)
    donor_map.add_donor(donor2)
    donor_map.add_donor(donor3)

    # Adding recipients with varying urgency
    recipient1 = Recipient(id="recipient1", blood_type="A", urgency=5)
    recipient2 = Recipient(id="recipient2", blood_type="B", urgency=10)
    recipient3 = Recipient(id="recipient3", blood_type="A", urgency=8)
    urgency_queue.add_recipient(recipient1)
    urgency_queue.add_recipient(recipient2)
    urgency_queue.add_recipient(recipient3)

    # Matching donors to recipients
    matches = match_donors_to_recipients(donor_map, urgency_queue)
    for recipient_id, donor_id in matches:
        print(f"Matched recipient {recipient_id} with donor {donor_id}.")
```

Output

```
python3 main.py
Matched recipient recipient2 with donor donor2.
Matched recipient recipient3 with donor donor1.
Matched recipient recipient1 with donor donor3.
```

Challenges Encountered

- **Integration of Data Structures**: Ensuring the user profiles, product relationships, and recommendations work cohesively posed initial challenges. I tackled this by defining clear interfaces for each data structure, allowing easy data flow.

- **Scoring System**: Developing a straightforward and effective scoring mechanism for recommendations required careful thought. I implemented a basic scoring method, which can be refined later.

Solutions Implemented

- **Modular Design**: Each data structure was designed with clear responsibilities, making future expansions easier.
- **Error Handling**: Basic error handling was included to manage cases such as duplicate users or missing product relationships.

Next Steps

1. **Enhance User Interaction Tracking**: Implement a more sophisticated method for tracking user interactions and preferences to improve recommendation accuracy.
2. **Advanced Recommendation Algorithms**: Explore collaborative filtering and content-based filtering techniques for generating more personalized recommendations.

3. **User Interface Development**: Develop a simple UI to facilitate user interaction with the recommendation system.
4. **Testing and Validation**: Conduct extensive testing to validate functionality, performance, and scalability as the application grows.

Project Phase 2 Deliverable 2: Proof of Concept Implementation

1. Recipient Class and Priority Queue

```
import heapq

class Recipient:
    def __init__(self, id, blood_type, urgency, location):
        self.id = id
        self.blood_type = blood_type
        self.urgency = urgency
        self.location = location #new

    def __lt__(self, other):
        return self.urgency > other.urgency  # Higher urgency gets higher priority

class UrgencyQueue:
    def __init__(self):
        self.heap = []

    def add_recipient(self, recipient):
        heapq.heappush(self.heap, recipient)

    def get_highest_priority(self):
        return heapq.heappop(self.heap) if self.heap else None
```

2. Donor Class and Hash Table

```
class Donor:
    def __init__(self, id, blood_type, hla_type, location):
        self.id = id
        self.blood_type = blood_type
        self.hla_type = hla_type
        self.location = location

class DonorHashMap:
    def __init__(self):
        self.hash_table = {blood_type: [] for blood_type in ["A", "B", "AB", "O"]}

    def add_donor(self, donor):
        self.hash_table[donor.blood_type].append(donor)

    def get_donors_by_blood_type(self, blood_type):
        return self.hash_table.get(blood_type, [])
```

3. Simple Geographical Matching Function

Installation: GeoPy

```
pip install geopy
```

```python
from geopy.distance import geodesic

def find_best_match(recipient, donors):
    best_match = None
    best_distance = float('inf')

    for donor in donors:
        if donor.blood_type == recipient.blood_type:  # Check blood type 
compatibility
            distance = geodesic(recipient.location, donor.location).miles
            if distance < best_distance:
                best_distance = distance
                best_match = donor

    return best_match
```

4. Blockchain Setup

Integrating blockchain with machine learning (ML) in an organ matching and donation network can provide enhanced security, data integrity, and improved decision-making through predictive analytics. Below is a structured approach to implement this integration along with relevant source references for your research.

Smart Contract for Organ Donation.

```
contract OrganDonation {
    struct Donor {
        string id;
        string bloodType;
        string hlaType;
        address owner;
    }

    struct Recipient {
        string id;
        string bloodType;
        uint urgency;
        address owner;
    }

    mapping(string => Donor) public donors;
```

```
    mapping(string => Recipient) public recipients;

    event DonorRegistered(string id);
    event RecipientRegistered(string id);

    function registerDonor(string memory _id, string memory _bloodType, string
memory _hlaType) public {
        donors[_id] = Donor(_id, _bloodType, _hlaType, msg.sender);
        emit DonorRegistered(_id);
    }

    function registerRecipient(string memory _id, string memory _bloodType, uint
_urgency) public {
        recipients[_id] = Recipient(_id, _bloodType, _urgency, msg.sender);
        emit RecipientRegistered(_id);
    }
}
```

Deploy the Smart Contract using tools like Truffle or Hardhat.

5. Machine Learning Model Development

Data Preparation: Gather historical data on donors and recipients.

```python
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score

# Load dataset
data = pd.read_csv('donor_recipient_data.csv')
X = data[['blood_type', 'urgency', 'distance']]  # Features
y = data['match_success']  # Target variable

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)

# Train model
model = RandomForestClassifier()
model.fit(X_train, y_train)

# Predictions
y_pred = model.predict(X_test)
print("Accuracy:", accuracy_score(y_test, y_pred))
```

Integration with Blockchain: After matching is predicted, save the results back to the blockchain.

```python
from web3 import Web3
```

```
# Connect to Ethereum network
w3 = Web3(Web3.HTTPProvider('http://localhost:8545'))
contract = w3.eth.contract(address='YOUR_CONTRACT_ADDRESS', abi='YOUR_ABI')

def save_match_result(donor_id, recipient_id, match_success, private_key):
    tx = contract.functions.saveMatchResult(donor_id, recipient_id, 
match_success).buildTransaction({
        'donor': 1,
        'level': 1,
        'urgency': w3.toWei('1', 'RED'),
        'nonce': 
w3.eth.getTransactionCount(w3.eth.account.privateKeyToAccount(private_key).address
),
    })

    signed_tx = w3.eth.account.signTransaction(tx, private_key)
    w3.eth.sendRawTransaction(signed_tx.rawTransaction)
```

6. API Development

Develop a RESTful API (using Flask or FastAPI) to interface with both the blockchain and ML model.

```
from flask import Flask, request, jsonify

app = Flask(__name__)

@app.route('/match', methods=['POST'])
def match():
    data = request.json
    # Predict match using the ML model
    match_success = model.predict([data['blood_type'], data['urgency'], 
data['distance']])

    # Save result to blockchain
    save_match_result(data['donor_id'], data['recipient_id'], match_success)

    return jsonify({'match_success': match_success.tolist()})

if __name__ == '__main__':
    app.run(debug=True)
```

Integrating smart contracts, machine learning, and optimizations into the organ matching system requires a more sophisticated approach. Here's how we can outline the implementation for these components, along with an updated code snippet to illustrate these concepts on next phase 3.

Proof of Concept Usage

```
if __name__ == "__main__":
    # Create a queue for recipients
```

```
    recipient_queue = UrgencyQueue()

    # Add recipients
    recipient1 = Recipient("rec1", "A", 5, (40.7128, -74.0060))  # New York
    recipient2 = Recipient("rec2", "B", 10, (34.0522, -118.2437))  # Los Angeles
    recipient_queue.add_recipient(recipient1)
    recipient_queue.add_recipient(recipient2)

    # Create a hash map for donors
    donor_map = DonorHashMap()

    # Add donors
    donor1 = Donor("don1", "A", "HLA1", (41.8781, -87.6298))  # Chicago
    donor2 = Donor("don2", "B", "HLA2", (34.0522, -118.2437))  # Los Angeles
    donor_map.add_donor(donor1)
    donor_map.add_donor(donor2)

    # Process matching for the highest priority recipient
    highest_priority_recipient = recipient_queue.get_highest_priority()
    donors_of_same_blood_type =
donor_map.get_donors_by_blood_type(highest_priority_recipient.blood_type)
    best_match = find_best_match(highest_priority_recipient,
donors_of_same_blood_type)

    if best_match:
        print(f"Best match for {highest_priority_recipient.id}: Donor ID
{best_match.id} at location {best_match.location}")
    else:
        print(f"No suitable donor found for {highest_priority_recipient.id}.")
```

Explanation

- **Recipient Class**: Holds information about each recipient, including urgency and location.
- **UrgencyQueue Class**: Implements a priority queue to manage recipients based on urgency.
- **Donor Class**: Stores donor information, including blood type and location.
- **DonorHashMap Class**: Uses a hash table to organize donors by blood type.
- **find_best_match Function**: Compares recipients and donors based on blood type and geographic distance, returning the closest suitable donor.

Future Enhancements

- **Integration with Databases**: Store recipient and donor data in a database for persistence.
- **Machine Learning**: Implement predictive matching algorithms based on historical data.
- **Real-time Updates**: Use webhooks or similar methods to update donor/recipient information in real time.
- **Advanced Geolocation**: Enhance the geographical matching function with more sophisticated routing algorithms.

Project Phase 3: Optimization, Scaling, and Final Evaluation

Goal & Objective

1. Optimization of Data Structures Analyze performance and identify inefficiencies. Implement optimizations like caching frequently accessed user profiles.

2. Scaling for Large Datasets Modify implementations to manage larger datasets effectively, ensuring acceptable performance levels.

3. Advanced Testing and Validation Develop comprehensive test cases and perform stress testing to evaluate robustness.

4. Final Evaluation and Performance Analysis Compare the final implementation with the initial proof of concept, discussing strengths, limitations, and areas for improvement.

Implementation Overview

1. Objectives

- **Blockchain**: Ensure secure and transparent storage of donor and recipient data.
- **Machine Learning**: Use historical data to predict successful matches based on various features.

2. System Architecture

- **Blockchain Layer**: Manages donor and recipient data securely.
- **ML Model Layer**: Predicts match success and recommends optimal matches.
- **API Layer**: Interfaces between the blockchain, ML model, and frontend application.

3.1 Integration with Databases

Objective: Store recipient and donor data for persistence, enabling easy retrieval and management.

Implementation Steps:

- **Choose a Database:** Select a relational database (e.g., PostgreSQL, MySQL) or a NoSQL database (e.g., MongoDB) based on your data structure and querying needs.

- **Database Models:** Define models for Donor and Recipient, using an ORM like SQLAlchemy for relational databases or a library like PyMongo for NoSQL.

```
from sqlalchemy import create_engine, Column, String, Integer, Float
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.orm import sessionmaker

Base = declarative_base()

class Donor(Base):
    __tablename__ = 'donors'
    id = Column(String, primary_key=True)
    blood_type = Column(String)
    hla_type = Column(String)
    location_lat = Column(Float)
    location_lon = Column(Float)
```

```python
class Recipient(Base):
    __tablename__ = 'recipients'
    id = Column(String, primary_key=True)
    blood_type = Column(String)
    urgency = Column(Integer)
    location_lat = Column(Float)
    location_lon = Column(Float)

# Set up the database
engine = create_engine('sqlite:///organ_donation.db')
Base.metadata.create_all(engine)
Session = sessionmaker(bind=engine)
```

- **Data Access Methods:** Implement functions to add, retrieve, and update donor/recipient records.

3.2 Machine Learning

Objective: Use historical data to improve matching accuracy through predictive algorithms.

Implementation Steps:

- **Data Collection:** Gather historical donor and recipient data, including successful matches and outcomes.
- **Feature Engineering:** Identify relevant features (e.g., blood type, urgency, location distance) that influence successful matches.
- **Predictive Matching:** Integrate the model into your matching logic to prioritize matches based on predicted success rates.

3.3 Real-time Updates

Objective: Ensure that donor and recipient information is up-to-date without manual intervention.

Implementation Steps:

- **Webhooks:** Set up webhooks to listen for changes in external data sources (e.g., hospitals, health organizations).
- **API Integration:**
  - Use libraries like `requests` to fetch real-time updates from APIs.

```python
import requests

def fetch_donor_data():
    response = requests.get('https://api.example.com/donors')
    if response.status_code == 200:
        data = response.json()
        # Update the database with new data
    else:
        print(f"Error fetching data: {response.status_code}")
```

```
```

- **Scheduled Tasks:** Use a task scheduler (e.g., `schedule` library, Celery) to periodically check for updates.

3.4 Advanced Geolocation

Objective: Enhance geographic matching with sophisticated algorithms.

Implementation Steps:

- **Routing Algorithms:** Implement algorithms like Dijkstra's or A* for more accurate pathfinding based on real traffic data.
- **Geospatial Libraries:** Use libraries like `geopy` or `Shapely` to perform complex geospatial operations.
- **Distance Calculation:**
  - Improve distance calculations by factoring in real-world conditions like traffic patterns.

  ```python
  from geopy.distance import geodesic

  def calculate_distance(location1, location2):
      return geodesic(location1, location2).miles
  ```

1. Smart Contract Implementation

Smart contracts can be implemented on a blockchain to ensure data integrity and automate the matching process. For this example, we'll use Solidity for the smart contract and assume you have a blockchain environment set up (like Ethereum or a test network).

Solidity Smart Contract Example

- SPDX-License-Identifier: MIT : pragma solidity ^0.8.0;

```
contract OrganDonation {
    struct Donor {
        string id;
        string bloodType;
        string hlaType;
        string location;
        bool isAvailable;
    }

    struct Recipient {
        string id;
        string bloodType;
        uint urgency;
        string location;
    }
```

```
    mapping(string => Donor) public donors;
    mapping(string => Recipient) public recipients;

    function addDonor(string memory id, string memory bloodType, string memory
hlaType, string memory location) public {
        donors[id] = Donor(id, bloodType, hlaType, location, true);
    }

    function addRecipient(string memory id, string memory bloodType, uint urgency,
string memory location) public {
        recipients[id] = Recipient(id, bloodType, urgency, location);
    }

    function matchDonor(string memory recipientId) public view returns (string
memory) {
        Recipient memory recipient = recipients[recipientId];
        // Add logic to find the best match based on blood type and urgency.
        // This is simplified; implement a proper search algorithm.
        for (string memory donorId in donors) {
            if (donors[donorId].isAvailable &&
compareBloodType(donors[donorId].bloodType, recipient.bloodType)) {
                return donorId; // Return matching donor ID
            }
        }
        return "No suitable donor found";
    }

    function compareBloodType(string memory bloodType1, string memory bloodType2)
private pure returns (bool) {
        // Simple blood type comparison logic
        return keccak256(abi.encodePacked(bloodType1)) ==
keccak256(abi.encodePacked(bloodType2));
    }
}
```

2. Machine Learning Integration

Machine learning can be used to predict the best donor-recipient matches based on historical data. For this, you can utilize libraries such as `scikit-learn` in Python.

Example: Predictive Model for Matching

Here’s an outline of how you might implement a simple predictive model:

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier

# Sample data preparation
data = pd.read_csv('organ_donation_data.csv')  # Assume this CSV contains
historical match data
```

```
features = data[['blood_type_donor', 'blood_type_recipient', 'urgency']]
labels = data['match_success']  # 1 if matched successfully, 0 otherwise

# Train-test split
X_train, X_test, y_train, y_test = train_test_split(features, labels,
test_size=0.2)

# Model training
model = RandomForestClassifier()
model.fit(X_train, y_train)

# Example of making predictions
def predict_match(donor_info):
    donor_df = pd.DataFrame([donor_info])  # Convert donor_info to DataFrame
    return model.predict(donor_df)

# Use the model in the matching process
recipient_info = {
    'blood_type_donor': 'A',
    'blood_type_recipient': 'A',
    'urgency': 5
}

match_prediction = predict_match(recipient_info)
print(f"Match prediction: {match_prediction[0]}")
```

3. Optimizations

Code Optimizations

- **Database Indexing**: Ensure the SQLite database has indexes on commonly queried columns like `blood_type`.
- **Efficient Data Structures**: Use more efficient data structures (like a balanced binary tree) for handling recipient and donor lists, allowing for faster lookups.

Python Code Enhancements

Here's an updated Python code snippet that incorporates some of these changes, including a more structured approach and the machine learning integration:

```
import heapq
from geopy.distance import geodesic
import pandas as pd
from sklearn.ensemble import RandomForestClassifier
from sqlalchemy import create_engine, Column, String, Integer, Float
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.orm import sessionmaker

# Database setup
Base = declarative_base()
```

```
engine = create_engine('sqlite:///organ_donation.db')
Session = sessionmaker(bind=engine)
session = Session()

class Recipient(Base):
    __tablename__ = 'recipients'
    id = Column(String, primary_key=True)
    blood_type = Column(String)
    urgency = Column(Integer)
    location = Column(String)

class Donor(Base):
    __tablename__ = 'donors'
    id = Column(String, primary_key=True)
    blood_type = Column(String)
    hla_type = Column(String)
    location = Column(String)

Base.metadata.create_all(engine)

class UrgencyQueue:
    def __init__(self):
        self.heap = []

    def add_recipient(self, recipient):
        heapq.heappush(self.heap, recipient)

    def get_highest_priority(self):
        return heapq.heappop(self.heap) if self.heap else None

def find_best_match(recipient, donors):
    best_match = None
    best_distance = float('inf')

    for donor in donors:
        if donor.blood_type == recipient.blood_type:  # Check blood type
compatibility
            distance = geodesic(recipient.location, donor.location).miles
            if distance < best_distance:
                best_distance = distance
                best_match = donor

    return best_match

# Machine Learning Model Preparation
data = pd.read_csv('organ_donation_data.csv')
features = data[['blood_type_donor', 'blood_type_recipient', 'urgency']]
labels = data['match_success']

# Train the model
model = RandomForestClassifier()
model.fit(features, labels)

# Example Matching Process
```

```
def match_recipients_to_donors(recipient_queue):
    while recipient_queue.heap:
        recipient = recipient_queue.get_highest_priority()
        donors = session.query(Donor).filter(Donor.blood_type ==
recipient.blood_type).all()

        if donors:
            best_match = find_best_match(recipient, donors)
            if best_match:
                # Use machine learning model to predict match success
                prediction = model.predict([[best_match.blood_type,
recipient.blood_type, recipient.urgency]])
                if prediction[0] == 1:  # If the prediction is successful
                    print(f"Matched recipient {recipient.id} with donor
{best_match.id}.")
                else:
                    print(f"No suitable donor found for recipient
{recipient.id}.")
        else:
            print(f"No donors available for recipient {recipient.id}.")

# Main Execution Flow
if __name__ == "__main__":
    recipient_queue = UrgencyQueue()

    # Add recipients and donors to the database (omitted for brevity)
    # Call match_recipients_to_donors(recipient_queue)
```

Project Phase 4: Final Report and Presentation

Requirements

Run as Adminstrator

```
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Try the new cross-platform PowerShell https://aka.ms/pscore6

PS C:\WINDOWS\system32> Set-ExecutionPolicy RemoteSigned
>>

Execution Policy Change
The execution policy helps protect you from scripts that you do not trust.
Changing the execution policy might expose
you to the security risks described in the about_Execution_Policies help topic at
https:/go.microsoft.com/fwlink/?LinkID=135170. Do you want to change the execution
policy?
[Y] Yes  [A] Yes to All  [N] No  [L] No to All  [S] Suspend  [?] Help (default is
"N"): y
```

```
 python -m venv python-env  # activate the enviroment
 .\python-env\Scripts\Activate.ps1
(python-env) PS C:\\MSCS532_Project>
```

```
pip install -r requirements.txt #requirment packages
```

```
pip install -r requirements.txt
Collecting geopy (from -r requirements.txt (line 1))
  Using cached geopy-2.4.1-py3-none-any.whl.metadata (6.8 kB)
Collecting pandas (from -r requirements.txt (line 2))
  Using cached pandas-2.2.3-cp312-cp312-win_amd64.whl.metadata (19 kB)
Collecting scikit-learn (from -r requirements.txt (line 3))
  Downloading scikit_learn-1.5.2-cp312-cp312-win_amd64.whl.metadata (13 kB)
Collecting SQLAlchemy (from -r requirements.txt (line 4))
  Downloading SQLAlchemy-2.0.35-cp312-cp312-win_amd64.whl.metadata (9.9 kB)
Collecting geographiclib<3,>=1.52 (from geopy->-r requirements.txt (line 1))
  Using cached geographiclib-2.0-py3-none-any.whl.metadata (1.4 kB)
Collecting numpy>=1.26.0 (from pandas->-r requirements.txt (line 2))
  Using cached numpy-2.1.1-cp312-cp312-win_amd64.whl.metadata (59 kB)
Collecting python-dateutil>=2.8.2 (from pandas->-r requirements.txt (line 2))
  Using cached python_dateutil-2.9.0.post0-py2.py3-none-any.whl.metadata (8.4 kB)
Collecting pytz>=2020.1 (from pandas->-r requirements.txt (line 2))
  Using cached pytz-2024.2-py2.py3-none-any.whl.metadata (22 kB)
Collecting tzdata>=2022.7 (from pandas->-r requirements.txt (line 2))
  Using cached tzdata-2024.2-py2.py3-none-any.whl.metadata (1.4 kB)
Collecting scipy>=1.6.0 (from scikit-learn->-r requirements.txt (line 3))
  Downloading scipy-1.14.1-cp312-cp312-win_amd64.whl.metadata (60 kB)
Collecting joblib>=1.2.0 (from scikit-learn->-r requirements.txt (line 3))
  Downloading joblib-1.4.2-py3-none-any.whl.metadata (5.4 kB)
Collecting threadpoolctl>=3.1.0 (from scikit-learn->-r requirements.txt (line 3))
  Downloading threadpoolctl-3.5.0-py3-none-any.whl.metadata (13 kB)
Collecting typing-extensions>=4.6.0 (from SQLAlchemy->-r requirements.txt (line
4))
  Downloading typing_extensions-4.12.2-py3-none-any.whl.metadata (3.0 kB)
Collecting greenlet!=0.4.17 (from SQLAlchemy->-r requirements.txt (line 4))
  Downloading greenlet-3.1.1-cp312-cp312-win_amd64.whl.metadata (3.9 kB)
Collecting six>=1.5 (from python-dateutil>=2.8.2->pandas->-r requirements.txt
(line 2))
  Using cached six-1.16.0-py2.py3-none-any.whl.metadata (1.8 kB)
Using cached geopy-2.4.1-py3-none-any.whl (125 kB)
Using cached pandas-2.2.3-cp312-cp312-win_amd64.whl (11.5 MB)
Downloading scikit_learn-1.5.2-cp312-cp312-win_amd64.whl (11.0 MB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 11.0/11.0 MB 14.6 MB/s eta 0:00:00
Downloading SQLAlchemy-2.0.35-cp312-cp312-win_amd64.whl (2.1 MB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 2.1/2.1 MB 14.6 MB/s eta 0:00:00
Using cached geographiclib-2.0-py3-none-any.whl (40 kB)
Downloading greenlet-3.1.1-cp312-cp312-win_amd64.whl (299 kB)
Downloading joblib-1.4.2-py3-none-any.whl (301 kB)
```

```
Using cached numpy-2.1.1-cp312-cp312-win_amd64.whl (12.6 MB)
Using cached python_dateutil-2.9.0.post0-py2.py3-none-any.whl (229 kB)
Using cached pytz-2024.2-py2.py3-none-any.whl (508 kB)
Downloading scipy-1.14.1-cp312-cp312-win_amd64.whl (44.5 MB)
   ---------------------------------------- 44.5/44.5 MB 18.8 MB/s eta 0:00:00
Downloading threadpoolctl-3.5.0-py3-none-any.whl (18 kB)
Downloading typing_extensions-4.12.2-py3-none-any.whl (37 kB)
Using cached tzdata-2024.2-py2.py3-none-any.whl (346 kB)
Using cached six-1.16.0-py2.py3-none-any.whl (11 kB)
Installing collected packages: pytz, tzdata, typing-extensions, threadpoolctl,
six, numpy, joblib, greenlet, geographiclib, SQLAlchemy, scipy, python-dateutil,
geopy, scikit-learn, pandas
Successfully installed SQLAlchemy-2.0.35 geographiclib-2.0 geopy-2.4.1 greenlet-
3.1.1 joblib-1.4.2 numpy-2.1.1 pandas-2.2.3 python-dateutil-2.9.0.post0 pytz-
2024.2 scikit-learn-1.5.2 scipy-1.14.1 six-1.16.0 threadpoolctl-3.5.0 typing-
extensions-4.12.2 tzdata-2024.2
```

```
geographiclib==2.0
geopy==2.4.1
greenlet==3.1.1
joblib==1.4.2
numpy==2.1.1
pandas==2.2.3
python-dateutil==2.9.0.post0
pytz==2024.2
scikit-learn==1.5.2
scipy==1.14.1
six==1.16.0
SQLAlchemy==2.0.35
threadpoolctl==3.5.0
typing_extensions==4.12.2
tzdata==2024.2
```

```
python -m pip freeze # Check if it meet all requirements
```

```
deactivate # deactivate the enviroment
```

Acknowledgment

"This work was supported in part by Health Resources and Services Administration contract HHSH250-2019-00001C. The content is the responsibility of the authors alone and does not necessarily reflect the views or policies of the Department of Health and Human Services, nor does mention of trade names, commercial products, or organizations imply endorsement by the U.S. Government."

> 103,223 Number of men, women, and children on the national transplant waiting list.

> Every donor can save 8 lives and enhance over 75 more.

> 17 people die each day waiting for an organ transplant.

2004 Annual Report of the U.S. Organ Procurement and Transplantation Network and the Scientific Registry of Transplant Recipients: Transplant Data 1994-2003. Department of Health and Human Services, Health Resources and Services Administration, Healthcare Systems Bureau, Division of Transplantation, Rockville, MD; United Network for Organ Sharing, Richmond, VA; University Renal Research and Education Association, Ann Arbor, MI.

Dataset: HRSA.gov

The OPTN Web site offers a wealth of information about transplantatio as of October 4, 2024

Google Collab: Use cloud CPU & GPU performance

> Python Notebook: OrganDonation.ipynb

Deploy Smart Contract: example

Conclusion

Incorporating blockchain, smart contracts, machine learning, and optimization techniques significantly enhances the functionality and robustness of the organ matching system. This approach not only ensures data integrity and security but also leverages predictive analytics for better matching outcomes. Future work can focus on refining these components and expanding the system's capabilities.

GitHub Repository

The full codebase, including the implementation of data structures and the test script, is available in the following GitHub repository: MSCS532_Project

References

1. Igboanusi, I. S., Nnadiekwe, C. A., Ogbede, J. U., Kim, D.-S., & Lensky, A. (2024). BOMS:blockchain-enabled organ matching system. Scientific Reports, 14(1), 1–13. https://doi.org/10.1038/s41598-024-66375-5
2. Al-Thnaibat, M. H., Balaw, M. K., Al-Aquily, M. K., Ghannam, R. A., Mohd, O. B., Alabidi, F., Alabidi, S., Hussein, F., & Rawashdeh, B. (2024). Addressing Kidney Transplant Shortage: The Potential of Kidney Paired Exchanges in Jordan. Journal of Transplantation, 2024, 1–8.https://doi.org/10.1155/2024/4538034
3. Cloutier, M., Grégoire, Y., Choucha, K., Amja, A.-M., & Lewin, A. (2021). Prediction of donation return rate in young donors using machine-learning models. ISBT Science Series, 16(1), 119–126. https://doi.org/10.1111/voxs.12618
4. Connor, J. P., Raife, T., & Medow, J. E. (2018). Outcomes of red blood cell transfusions prescribed in organ donors by the Digital Intern, an electronic decision support algorithm. Transfusion, 58(2), 366–371. https://doi.org/10.1111/trf.14424 Other References

5. ONC’s Cures Act Final Rule | HealthIT.gov. (2022, August 31). Www.healthit.gov. https://www.healthit.gov/topic/oncs-cures-act-final-rule
6. PacktPublishing. “Packtpublishing/Mastering-Geospatial-Analysis-with-Python: Mastering Geospatial Analysis with Python, Published by Packt.” GitHub, https://github.com/PacktPublishing/Mastering-Geospatial-Analysis-with-Python?tab=readme-ov-file. Accessed 23 Sept. 2024.
7. “Pattern Recognition and Machine Learning - Free Computer, Programming, Mathematics, Technical Books, Lecture Notes and Tutorials.” FreeComputerBooks, https://freecomputerbooks.com/Pattern-Recognition-and-Machine-Learning.html. Accessed 23 Sept. 2024.
8. Pattern Recognition and Machine Learning, https://www.microsoft.com/en-us/research/uploads/prod/2006/01/Bishop-Pattern-Recognition-and-Machine-Learning-2006.pdf. Accessed 24 Sept. 2024.
9. “Sqlalchemy.” PyPI, https://pypi.org/project/SQLAlchemy/. Accessed 23 Sept. 2024.
10. Sundriyal, Vaibhav, and Masha Sosonkina. “Runtime Energy Savings Based on Machine Learning Models for Multicore Applications.” SCIRP, Scientific Research Publishing, 9 June 2022, https://www.scirp.org/journal/paperinformation?paperid=118212.
11. “Welcome to GeoPy’s Documentation!□.” Welcome to GeoPy’s Documentation! - GeoPy 2.4.1 Documentation, https://geopy.readthedocs.io/en/stable/#installation. Accessed 23 Sept. 2024.
12. Agbo CC, Mahmoud QH, Eklund JM. Blockchain Technology in Healthcare: A Systematic Review. Healthcare. 2019; 7(2):56. https://doi.org/10.3390/healthcare7020056
13. Author links open overlay panelS.M Campbell, et al. “Defining Quality of Care.” Social Science & Medicine, Pergamon, 5 Oct. 2000, https://www.sciencedirect.com/science/article/abs/pii/S0277953600000575.
14. Learn How Organ Allocation Works - OPTN. Organ Procurement and Transplantation Network. (2024). https://optn.transplant.hrsa.gov/patients/about-transplantation/how-organ-allocation-works/
15. Arashtad. “How to Deploy a Smart Contract Using Python Web3 Tools: A Full Coverage.” Medium, Medium, 19 Feb. 2024, https://medium.com/@arashtad/how-to-deploy-a-smart-contract-using-python-web3-tools-a-full-coverage-59e6c2ad3f9f.
