Organ Matching and Donation Networks

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)

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

Binary Tree (AVL Tree)

```
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

```python
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.

```
# Test script demonstrating functionality
user_table = UserHashTable()
product_graph = Graph()
recommendation_queue = RecommendationQueue()

# Adding users
user1 = UserProfile(user_id="user1", preferences=["productA", "productB"])
user2 = UserProfile(user_id="user2", preferences=["productB", "productC"])
user_table.add_user(user1)
user_table.add_user(user2)

# Adding product relationships
product_graph.add_edge("productA", "productB")
product_graph.add_edge("productB", "productC")

# Adding recommendations
recommendation_queue.add_recommendation(Recommendation("productA", 5))
recommendation_queue.add_recommendation(Recommendation("productB", 3))
recommendation_queue.add_recommendation(Recommendation("productC", 8))

# Retrieving the top recommendation
top_recommendation = recommendation_queue.get_top_recommendation()
print(f"Top Recommendation: {top_recommendation.product_id} with score
{top_recommendation.score}")
```

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.

GitHub Repository

The full codebase, including the implementation of data structures and the test script, is available in the following GitHub repository: https://github.com/baralsamrat/MSCS532_Project

Phase 3: Optimization, Scaling, and Final Evaluation

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.

References

1. Igboanusi et al. (2024). BOMS: blockchain-enabled organ matching system. *Scientific Reports*.
2. Al-Thnaibat et al. (2024). Addressing Kidney Transplant Shortage: The Potential of Kidney Paired Exchanges in Jordan. *Journal of Transplantation*.
3. Cloutier et al. (2021). Prediction of donation return rate in young donors using machine-learning models. *ISBT Science Series*.
4. Connor et al. (2018). Outcomes of red blood cell transfusions prescribed in organ donors. *Transfusion*.
5. HealthIT.gov. (2022). ONC’s Cures Act Final Rule.
