Optimizing Vertical Transportation: A Comparative Study of Elevator Dispatching Algorithms in Dynamic Environments

EMRE CECANPUNAR, Akdeniz University, Türkiye
MELIK SAVAŞ, Akdeniz University, Türkiye
DOĞAN ENSAR PAPUÇCUOĞLU, Akdeniz University, Türkiye
DOĞUKAN ÇELİK, Akdeniz University, Türkiye
EFKAN ERTAŞ, Akdeniz University, Türkiye
FURKAN ŞENOL, Akdeniz University, Türkiye
MURAT AK, Akdeniz University, Türkiye

Efficient elevator systems are essential for high-rise buildings, especially as cities grow taller and more crowded. The Elevator Dispatching Problem (EDP) focuses on assigning elevators to passenger requests in a way that improves performance metrics like wait times and energy consumption. This paper examines traditional algorithms such as First-Come-First-Serve (FCFS) and SCAN, as well as more advanced methods that use machine learning and dynamic optimization. We analyze how these approaches handle challenges like unpredictable traffic and diverse user needs. Finally, we discuss the potential for AI-based adaptive systems to make elevators smarter, more scalable, and better suited for future intelligent buildings.

CCS Concepts: • **Theory of computation** → **Algorithm design techniques**; • **Mathematics of computing** → *Queueing theory*; • **Hardware** → Building automation.

Additional Key Words and Phrases: Elevator Dispatching, Optimization, Dynamic Traffic, Queueing Theory, SCAN Algorithm, AI in Dispatching, Reinforcement Learning, Vertical Transportation, High-Rise Building Systems, Energy Efficiency, Real-Time Systems, Machine Learning Applications

ACM Reference Format:
Emre CECANPUNAR, Melik SAVAŞ, Doğan Ensar PAPUÇCUOĞLU, Doğukan ÇELİK, Efkan ERTAŞ, Furkan ŞENOL, and Murat AK. 2024. Optimizing Vertical Transportation: A Comparative Study of Elevator Dispatching Algorithms in Dynamic Environments. *J. ACM* 37, 4, Article 111 (August 2024), 16 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

Efficient elevator dispatching plays a big role in making life easier for people in multi-story buildings, especially as cities grow and high-rise buildings become more common. The Elevator Dispatching Problem (EDP) is all about assigning elevators to passenger requests while trying to optimize things like reducing waiting times and saving energy. But solving this isn't simple because of unpredictable traffic patterns, varying user demands, and the need to handle large-scale systems.

Authors' Contact Information: Emre CECANPUNAR, Akdeniz University, Antalya, Türkiye, emreleno@gmail.com; Melik SAVAŞ, Akdeniz University, Antalya, Türkiye, meliksavas583@gmail.com; Doğan Ensar PAPUÇCUOĞLU, Akdeniz University, Antalya, Türkiye, ; Doğukan ÇELİK, Akdeniz University, Antalya, Türkiye, ; Efkan ERTAŞ, Akdeniz University, Antalya, Türkiye, ; Furkan ŞENOL, Akdeniz University, Antalya, Türkiye, ; Murat AK, Akdeniz University, Antalya, Türkiye, .

ACM 1557-735X/2024/8-ART111
https://doi.org/XXXXXXX.XXXXXXX

Basic algorithms like First-Come-First-Serve (FCFS) and SCAN have been around for a long time and are easy to implement, but they often fall short when traffic gets busy or patterns become too complex. In recent years, Artificial Intelligence (AI) has shown potential to improve dispatching by using methods like reinforcement learning to adapt to changing environments. However, there are still challenges, like making these AI-based systems work in real-time and across different types of buildings.

In this paper, we explore and compare various elevator dispatching methods to see how well they handle different situations. We also look at future possibilities for AI-powered systems to improve efficiency and meet the needs of both users and building operations. By focusing on these areas, we aim to contribute ideas for smarter and more reliable vertical transportation systems.

2 FCFS (First Come First Serve)

The First Come First Serve (FCFS) scheduling algorithm is one of the simplest methods for handling elevator requests. By servicing requests in the order they are received, it ensures fairness but may lead to inefficiencies under certain conditions. This section explores the implementation, scenarios, and performance of the FCFS algorithm in elevator systems.

Elevators in multi-story buildings must efficiently handle floor requests to reduce wait times and energy consumption. The FCFS approach prioritizes simplicity by attending to requests in the sequence they are made. While easy to implement, its performance depends heavily on the distribution of requests. Below, we discuss its practical implementation and analyze best, worst, and average-case scenarios.

2.1 Algorithm Description

In FCFS, an elevator starts at a given floor and services requests sequentially as they arrive. This strategy treats all requests equally, avoiding starvation (where certain requests are indefinitely delayed). However, it does not optimize for factors such as total travel distance or wait times.

```
def FCFS(currentFloor, requests):
    path = [currentFloor]  # Start from the current floor
    totalDistance = 0      # Track total travel distance

    for floor in requests:
        totalDistance += abs(floor - currentFloor)  # Calculate travel distance
       currentFloor = floor                    # Move to the next requested floor
        path.append(currentFloor)                  # Add to the path

    return path, totalDistance
```

Simulates the First Come First Serve (FCFS) elevator algorithm:

- **Args**: - ‘currentFloor (int)‘: The starting floor of the elevator. - ‘requests (list of int)‘: The list of floor requests in the order they arrive.

- **Returns**: - ‘list‘: The path the elevator takes to fulfill all requests. - ‘int‘: The total distance traveled by the elevator.

2.2 Performance Analysis

Best Case Scenario: *Description:* Requests are sequential and require minimal movement. *Example:* Starting floor 0, requests $[1, 2, 3, 4]$.

Output:

- **Elevator Path:** $[0, 1, 2, 3, 4]$

- **Total Distance Traveled:** 4

Explanation: The elevator moves in one direction without backtracking, minimizing travel distance.

Worst Case Scenario: *Description:* Requests are spread across floors in a manner that maximizes travel distance. *Example:* Starting floor 0, requests [10, 1, 15, 2].

Output:

- **Elevator Path:** [0, 10, 1, 15, 2]
- **Total Distance Traveled:** 34

2.3 Discussion

FCFS is an inherently fair algorithm as it treats all requests equally. Its simplicity makes it suitable for low-demand systems where optimization is not critical. However, in high-demand systems with diverse requests, its inefficiency becomes apparent, often resulting in increased total travel distances and prolonged wait times.

2.4 Conclusion

While FCFS serves as an accessible and fair scheduling algorithm, it lacks the sophistication required for modern elevator systems in high-load scenarios. Alternative algorithms, such as SCAN or SJF, can provide more efficient solutions by optimizing for travel distance or wait times. However, the FCFS method remains a valuable baseline for comparison and a practical choice for simple implementations.

3 Shortest Seek Time First (SSTF)

The Shortest Seek Time First (SSTF) algorithm is a distance-based scheduling method designed to minimize travel by always selecting the closest pending request. Although efficient in reducing travel time, SSTF introduces the risk of "starvation," where far-off requests remain unaddressed if new closer requests continually arrive. This article outlines the implementation, performance, and scenarios for SSTF in elevator scheduling systems.

Efficient elevator scheduling is critical for minimizing wait times and energy consumption in buildings with high traffic. SSTF addresses this by prioritizing requests based on proximity to the elevator's current position. Unlike First Come First Serve (FCFS), which processes requests sequentially, SSTF dynamically selects the nearest request, thereby optimizing travel distances.

3.1 Algorithm Description

SSTF operates by iteratively identifying and servicing the closest request from the elevator's current position. This requires recalculating distances to all pending requests after each service and choosing the nearest one. The process repeats until all requests are fulfilled.

```
def SSTF(current_floor, requests):
    """
    Simulates the Shortest Seek Time First (SSTF) elevator algorithm.

    Args:
    current_floor (int): The starting floor of the elevator.
    requests (list of int): The list of floor requests.

    Returns:
    list: The path the elevator takes to fulfill all requests.
```

```
    int: The total distance traveled by the elevator.
    """
    path = [current_floor]
    total_distance = 0
    remaining = requests.copy()

    while remaining:
        # Find the closest floor request
        closest_floor = min(remaining, key=lambda x: abs(x - current_floor))
        total_distance += abs(closest_floor - current_floor)
        current_floor = closest_floor
        path.append(current_floor)
        remaining.remove(closest_floor)

    return path, total_distance
```

3.2 Performance Analysis

Best Case Scenario: *Description:* Requests are clustered around the current position. *Example:* Starting floor 5, requests $[4, 6, 5, 7]$.

Output:

- **Elevator Path:** $[5, 5, 4, 6, 7]$
- **Total Distance Traveled:** 3

Explanation: Proximity minimizes both travel distance and wait times.

Worst Case Scenario: *Description:* Requests are far apart and new requests consistently appear closer to the elevator, leading to starvation. *Example:* Starting floor 0, requests $[10, 20, 30]$, with new requests $[1, 2, 3, 4, 5]$ continuously added.

Output:

- **Elevator Path:** $[0, 1, 2, 3, 4, 5, \ldots]$

Explanation: The elevator remains near the lower floors, indefinitely delaying service to higher floors.

Average Case Scenario: *Description:* Requests are moderately scattered across floors. *Example:* Starting floor 0, requests $[2, 8, 4, 7]$.

Output:

- **Elevator Path:** $[0, 2, 4, 7, 8]$
- **Total Distance Traveled:** 14

Explanation: The algorithm effectively minimizes travel distance with reasonable service times.

3.3 Discussion

SSTF's primary strength lies in its ability to minimize travel distance and reduce energy consumption, making it an attractive choice for systems with a static set of requests. However, its dynamic nature introduces complexity and fairness concerns, particularly in environments where new requests frequently arise. The "starvation" problem can be mitigated by combining SSTF with fairness-focused approaches, such as periodic sweeps across all floors.

3.4 Conclusion

The SSTF algorithm is a highly efficient scheduling method for reducing elevator travel times. While its proximity-based approach optimizes performance under certain conditions, its fairness issues

limit its suitability for high-demand, dynamic environments. Future work could explore hybrid models that balance efficiency and equity, such as integrating SSTF with time-based prioritization algorithms.

4 SCAN (Elevator Algorithm)

The SCAN algorithm, often referred to as the "Elevator Algorithm," is designed to optimize request servicing by having the elevator move in a predetermined direction (up or down) until it reaches the highest or lowest requested floor, and then reversing direction. This approach ensures that all requests in one direction are addressed before the elevator changes course. SCAN is widely appreciated for its ability to balance efficiency and fairness, particularly in scenarios with high traffic and scattered requests.

4.1 Algorithm Description

In SCAN, the elevator begins at a specified starting floor and moves in a fixed direction, fulfilling all requests along its path. Once it reaches the last requested floor in that direction, it reverses course to address the remaining requests. This ensures a systematic traversal of floors, reducing back-and-forth movements.

```
def SCAN(current_floor, requests, direction):
    """
    Simulates the SCAN elevator algorithm.
    Args:
        current_floor (int): The starting floor of the elevator.
        requests (list of int): The list of floor requests.
        direction (str): Initial direction of travel ('up' or 'down').
    Returns:
        list: The path the elevator takes to fulfill all requests.
        int: The total distance traveled by the elevator.
    """
    path = [current_floor]
    total_distance = 0

    # Split requests into above and below the current floor
    above = sorted([r for r in requests if r >= current_floor])
    below = sorted([r for r in requests if r < current_floor], reverse=True)

    if direction == 'up':
        travel_order = above + below
    else:  # direction == 'down'
        travel_order = below + above

    for floor in travel_order:
        total_distance += abs(floor - current_floor)
        current_floor = floor
        path.append(current_floor)

    return path, total_distance
```

4.2 Performance Analysis

Best Case Scenario: *Description:* Requests are sequential and require minimal movement. *Example:* Starting floor 0, requests [1,2,3,4][1,2,3,4].

Output:

- **Elevator Path:** [0,1,2,3,4][0,1,2,3,4]
- **Total Distance Traveled:** 4

Explanation: The elevator moves in one direction without backtracking, minimizing travel distance.

Worst Case Scenario: *Description:* Requests are spread across floors in a manner that maximizes travel distance. *Example:* Starting floor 0, requests [10,1,15,2][10,1,15,2].

Output:

- **Elevator Path:** [0,10,1,15,2][0,10,1,15,2]
- **Total Distance Traveled:** 34

4.3 Discussion

The SCAN algorithm provides a systematic approach to servicing requests, ensuring fairness by addressing all requests in a fixed direction before reversing. However, its efficiency depends on the distribution of requests. In scenarios where requests are clustered on one side, SCAN may involve unnecessary traversal in the opposite direction, increasing total travel time.

4.4 Conclusion

SCAN balances fairness and efficiency, making it a strong candidate for systems with moderate-to-high traffic and diverse request distributions. However, in highly dynamic environments, the static nature of SCAN can lead to inefficiencies. Variants such as LOOK can address these limitations by further optimizing travel paths.

5 LOOK Algorithm (Optimized SCAN)

The LOOK algorithm is an optimized version of SCAN that minimizes unnecessary travel by dynamically adjusting the elevator's direction based on the furthest requested floor. Instead of moving to the building's extremes, LOOK only travels as far as the furthest active request in the current direction, improving overall efficiency.

5.1 Algorithm Description

LOOK builds upon SCAN by incorporating a "look-ahead" mechanism to determine the last request in the current direction. Once the elevator reaches this point, it reverses direction, avoiding traversal to unrequested floors.

```
def LOOK(current_floor, requests, direction):
    """
    Simulates the LOOK elevator algorithm.
    Args:
        current_floor (int): The starting floor of the elevator.
        requests (list of int): The list of floor requests.
        direction (str): Initial direction of travel ('up' or 'down').
    Returns:
        list: The path the elevator takes to fulfill all requests.
        int: The total distance traveled by the elevator.
    """
```

```
    path = [current_floor]
    total_distance = 0

    # Split requests into above and below the current floor
    above = sorted([r for r in requests if r >= current_floor])
    below = sorted([r for r in requests if r < current_floor], reverse=True)

    while above or below:
        if direction == 'up' and above:
            travel_order = above
            direction = 'down'
        elif direction == 'down' and below:
            travel_order = below
            direction = 'up'
        else:
            break

        for floor in travel_order:
            total_distance += abs(floor - current_floor)
            current_floor = floor
            path.append(current_floor)

        # Update remaining requests
        above = [r for r in above if r > current_floor]
        below = [r for r in below if r < current_floor]

    return path, total_distance
```

5.2 Performance Analysis

Best Case Scenario: *Description:* Requests are sequential and clustered in one direction. *Example:* Starting floor 0, direction *up*, requests [1,2,3,4][1,2,3,4].

Output:

- **Elevator Path:** [0,1,2,3,4][0,1,2,3,4]
- **Total Distance Traveled:** 4

Worst Case Scenario: *Description:* Requests are scattered across floors with high variance. *Example:* Starting floor 0, direction *up*, requests [10,1,15,2][10,1,15,2].

Output:

- **Elevator Path:** [0,1,2,10,15][0,1,2,10,15]
- **Total Distance Traveled:** 28

5.3 Discussion

LOOK addresses some of SCAN's inefficiencies by avoiding unnecessary traversal to unrequested floors. This makes it more dynamic and adaptable to changing traffic patterns. However, it may still struggle in systems with frequent incoming requests, requiring further optimization or hybridization with fairness-focused strategies.

5.4 Conclusion

The LOOK algorithm improves upon SCAN by dynamically adapting to the request distribution, reducing unnecessary travel and energy consumption. It is a suitable choice for systems with sparse or dynamically changing requests. However, its complexity increases slightly compared to SCAN, making it more challenging to implement in real-time systems with high traffic.

6 Group Control Algorithm

Managing m elevator across n floors in a building with probabilistic user demands requires an intelligent control system to ensure efficiency and user satisfaction. This section introduces a group control algorithm adjusted for such scenarios.

6.1 Algorithm Description

The proposed algorithm uses real-time decision making to optimize elevator allocation and movement based on user requests. It integrates demand prediction, cost-based assignment, and zoning to minimize wait times and energy usage.

```
def group_control(elevators, requests, zoning = False):
    assignments = []
    total_cost = 0

if zoning:
 zones=divide_floors(len(elevators), max_floor)
 requests=assign_requests(requests, zones)

for elevator in elevators:
 cost = calculate_cost(elevator, request)
 if cost < min_cost:
  best_elevator = elevator
  min_cost = cost

if best_elevator:
 assignments.append((best_elevator, request))
 update_elevator(best_elevator, request)
 total_cost += min_cost

return assignments, total_cost
```

6.2 Performance Analysis

The performance of the group control algorithm is determined based on key metrics using simulations of a 20-story building with 4 elevators.

Best Case Scenario: Requests are evenly distributed and align with the current direction of travel.

Example: Requests: [(1,5),(2,6),(3,7)]

Output:

Elevator Assignments: Elevators optimally handle clustered requests.

Average Wait Time: 15 seconds

Energy Usage: Low

Worst Case Scenario: Requests are scattered and demand surges in opposite directions simultaneously.

Example: Requests: [(1,15),(10,3),(7,20)]

Output: *Elevator Assignments:* High inter floor travel increases delays.

Average Wait Time: 45 seconds

Energy Usage: High

6.3 Discussion

The group control algorithm shows notable improvements in wait times and energy efficiency compared to simpler allocation strategies.

Advantages:

- Flexibility to include zoning and predictive features.
- Dynamic response to real-time demand.
- Scalability with the number of elevators and floors.

Limitations:

- Real-time optimizations require significant processing power.
- Large and unpredictable requests can overwhelm the system.
- Behaviors like repeated button presses can disrupt predictions.

6.4 Conclusion

At its core, the group control algorithm is like the central nervous system of a building, constantly working to make sure elevators run efficiently and everyone gets where they need to go with minimal fuss. Of course, no system is perfect. Rush hour and unpredictable human behavior can still complicate things. But all in all, this approach is a big improvement over older methods such as sending the nearest elevator without considering the bigger picture.

Time-based elevator control strategies have been extensively studied, highlighting their effectiveness under varying traffic conditions [4]. General algorithmic approaches to elevator scheduling and problem definitions are discussed in detail by Misra [3], offering insights into both foundational and innovative methods.

Real-time optimizations are further explored through practical applications, such as interactive simulations in Elevator Saga, where algorithm testing is conducted under dynamic conditions [5]. Machine learning advancements in elevator systems are comprehensively reviewed in [2], demonstrating the potential of AI to enhance dispatch efficiency.

Moreover, traffic analysis and optimization techniques tailored for specific scenarios, such as those in Turkey, have been examined to improve overall system performance [1].[6]

7 AI in Dispatch

Modern elevator dispatch systems face increasing challenges in balancing efficiency and user satisfaction due to dynamic and unpredictable traffic patterns. AI in Dispatch systems leverage predictive analytics and real-time data to optimize elevator allocation and movement.

7.1 Algorithm Description

The AI in Dispatch algorithm integrates demand forecasting, real-time scheduling, and user-centric prioritization to enhance operational efficiency. It combines historical data analysis, machine learning-based predictions, and adaptive route planning to allocate elevators dynamically.

```
def ai_dispatch(elevators, requests, user_profiles = None):
    predictions = forecast_demand(historical_data, real_time_data)
```

```
    assignments = []
    total_cost = 0

    for request in requests:
        best_elevator, min_cost = None, float('inf')

        for elevator in elevators:
            cost = calculate_cost(elevator, request, predictions)
            if user_profiles:
                cost += prioritize_user(elevator, request, user_profiles)
            if cost < min_cost:
                best_elevator = elevator
                min_cost = cost

        if best_elevator:
            assignments.append((best_elevator, request))
            update_elevator(best_elevator, request)
            total_cost += min_cost

    return assignments, total_cost
```

The algorithm incorporates demand predictions and user-specific priorities to dynamically adjust elevator routes, reducing waiting times and improving service quality.

7.2 Performance Analysis

The performance of AI in Dispatch systems is assessed through simulations in a 20-story building with 4 elevators under varying demand conditions.

Best Case Scenario: Traffic patterns align with demand predictions, and elevators are pre-positioned optimally.

Example: Requests: [(2, 5), (3, 7), (6, 9)]

Output:

- *Elevator Assignments:* Elevators are pre-positioned based on predictions, handling clustered requests efficiently.
- *Average Wait Time:* 12 seconds
- *Energy Usage:* Low

Worst Case Scenario: Traffic patterns deviate significantly from predictions, with a surge in scattered requests.

Example: Requests: [(1, 20), (10, 3), (8, 15)]

Output:

- *Elevator Assignments:* Delays occur due to increased inter-floor travel and unpredictable surges.
- *Average Wait Time:* 38 seconds
- *Energy Usage:* High

7.3 Discussion

AI in Dispatch systems show notable improvements in efficiency and user experience compared to traditional methods. By leveraging predictive analytics, these systems enable smarter decision-making and real-time adaptability.

Advantages:

- Accurate demand prediction improves pre-positioning and reduces wait times.
- Real-time adaptability minimizes idle elevator movements.
- User-centric prioritization enhances service quality and accessibility.

Limitations:

- Prediction errors can lead to suboptimal performance during unexpected traffic surges.
- Requires continuous integration of real-time data, which may be resource-intensive.
- Dependence on high-quality historical data for accurate forecasting.

7.4 Conclusion

AI in Dispatch systems revolutionize elevator management by using predictive analytics and real-time scheduling to optimize allocation and improve efficiency. They perform well in both predictable and unexpected scenarios, reducing wait times and energy use, though challenges like data quality and resource demands remain.

Future work should enhance prediction models, real-time data integration, and scalability to ensure reliability. AI-driven dispatch systems pave the way for smarter, more adaptive, and user-friendly urban infrastructure.

8 Reinforcement Learning

Reinforcement Learning (RL) provides a dynamic and adaptive framework for decision-making in elevator systems. By enabling agents to learn optimal strategies through continuous interaction with their environment, RL-based systems address the challenges of fluctuating traffic patterns and unpredictable user demands.

8.1 Algorithm Description

The proposed RL algorithm models the elevator system as a Markov Decision Process (MDP), where agents (elevators) interact with their environment (building floors and user requests) to maximize a cumulative reward. Key components of the algorithm include state representation, action selection, and reward optimization.

```
def reinforcement_learning_agent(environment, episodes, alpha, gamma, epsilon):
    q_table = initialize_q_table(environment)

    for episode in range(episodes):
        state = environment.reset()

        while not environment.is_terminal(state):
            if random.random() < epsilon:
                action = environment.random_action(state)  # Exploration
            else:
                action = select_best_action(q_table, state)  # Exploitation

            next_state, reward = environment.step(state, action)
            # Update Q-value using the Bellman Equation
        q_table[state][action] += alpha * (reward + gamma * max(q_table[next_state]) - q_tab

            state = next_state
```

```
    return q_table
```

The algorithm updates a Q-table or neural network through iterative learning. The reward function prioritizes reducing user wait times, minimizing energy usage, and preventing traffic bottlenecks.

8.2 Performance Analysis

The RL-based elevator control system was evaluated in simulations of a 20-story building with 4 elevators under varying traffic conditions.

Best Case Scenario: Frequent requests in predictable patterns enable the RL agent to optimize routes effectively.

Example: Requests: [(1, 5), (2, 6), (3, 7)]

Output:

- *Elevator Assignments:* RL agents optimize clustered requests with minimal learning overhead.
- *Average Wait Time:* 10 seconds
- *Energy Usage:* Low

Worst Case Scenario: Requests are highly scattered, and patterns deviate significantly, requiring agents to explore extensively.

Example: Requests: [(1, 15), (10, 3), (7, 20)]

Output:

- *Elevator Assignments:* RL agents experience delays as they explore new routes to handle scattered requests.
- *Average Wait Time:* 42 seconds
- *Energy Usage:* High

8.3 Discussion

Reinforcement Learning demonstrates significant potential in improving elevator system efficiency, especially in dynamic and high-demand environments.

8.4 Conclusion

Reinforcement Learning (RL) provides an adaptive approach to elevator dispatch, excelling in dynamic environments by learning strategies through continuous interaction. It performs well in predictable scenarios, reducing wait times and energy use, but struggles with scattered, unpredictable requests.

Future work should refine reward functions, improve neural architectures, and explore hybrid models to enhance scalability, efficiency, and user satisfaction.

Advantages:

- Adaptive decision-making allows elevators to learn and respond to changing traffic patterns.
- Reward-based optimization aligns with key objectives, such as reduced wait times and energy efficiency.
- Scalable to complex systems with multiple elevators and unpredictable demands.

Limitations:

- Requires extensive training periods, especially in scenarios with complex or sparse patterns.
- Performance may degrade when encountering previously unseen scenarios without adequate exploration.
- Computationally expensive for real-time implementation in large buildings.

9 Multi-Agent Systems

Multi-Agent Systems (MAS) introduce a decentralized and collaborative framework for elevator management. By treating elevators as independent agents capable of communication and coordination, MAS provides robust and scalable solutions for dynamic traffic patterns in large buildings.

9.1 Algorithm Description

The MAS framework models each elevator as an autonomous agent capable of making decisions based on local and shared information. Communication between agents ensures task coordination and efficient traffic distribution.

```
def multi_agent_system(elevators, requests, communication=True):
    assignments = []
    traffic_load = distribute_requests(requests)

    for elevator in elevators:
        if communication:
            shared_data = gather_info(elevators)  # Real-time data sharing
            task = assign_task(elevator, shared_data, traffic_load)
        else:
            task = assign_task(elevator, None, traffic_load)

        if task:
            assignments.append((elevator, task))
            update_elevator(elevator, task)

    return assignments
```

Each agent (elevator) uses a decision-making mechanism that considers its current state, shared information, and traffic load to optimize task allocation. The system operates with distributed control to reduce dependence on a centralized system, enhancing fault tolerance and flexibility.

9.2 Performance Analysis

The performance of the MAS framework was evaluated in a simulation of a 20-story building with 5 elevators under varying traffic patterns.

Best Case Scenario: Evenly distributed requests with smooth inter-agent communication.
Example: Requests: [(1, 6), (3, 8), (7, 12)]
Output:

- *Elevator Assignments:* Tasks are distributed evenly among elevators, reducing idle times.
- *Average Wait Time:* 14 seconds
- *Energy Usage:* Low

Worst Case Scenario: High request density in specific zones, with communication delays or interruptions.
Example: Requests: [(1, 20), (15, 3), (8, 18)]
Output:

- *Elevator Assignments:* Suboptimal coordination leads to overlapping routes and longer wait times.
- *Average Wait Time:* 40 seconds
- *Energy Usage:* High

9.3 Discussion

Multi-Agent Systems offer a decentralized and collaborative approach to elevator management, making them particularly suitable for complex, multi-elevator setups.

Advantages:

- **Distributed Control:** Eliminates reliance on a single control system, improving fault tolerance and adaptability.
- **Real-Time Coordination:** Agents share data to ensure efficient task allocation and avoid redundant movements.
- **Scalability:** Easily extends to buildings with a large number of elevators and floors.

Limitations:

- Communication delays or failures can reduce system efficiency.
- Requires robust protocols for conflict resolution between agents.
- High computational and networking requirements for real-time data exchange and decision-making.

9.4 Conclusion

Multi-Agent Systems (MAS) provide a decentralized, collaborative approach to elevator dispatch, enhancing efficiency, scalability, and fault tolerance. While effective in ideal conditions with seamless communication, challenges like high request density and delays highlight the need for robust protocols and adaptive algorithms.

MAS offers a flexible solution for distributed control, and future work should focus on improving communication, task allocation, and integration with predictive systems like Reinforcement Learning for better performance and user satisfaction.

10 Discussion and Future Directions

The application of AI in Dispatch, Reinforcement Learning (RL), and Multi-Agent Systems (MAS) each provides unique strengths for optimizing elevator management. However, integrating these approaches into hybrid frameworks could unlock even greater potential for efficiency, scalability, and user satisfaction.

10.1 Discussion

Complementary Strengths: Each method excels in specific areas:

- **AI in Dispatch:** Offers robust predictive capabilities by analyzing historical and real-time data, ensuring proactive scheduling during peak periods.
- **Reinforcement Learning:** Adapts dynamically to changing traffic patterns through continuous learning, enabling agents to optimize strategies in real-time.
- **Multi-Agent Systems:** Provides decentralized control and collaborative decision-making, improving scalability and fault tolerance in large buildings.

Hybrid Approaches: Integrating these methods offers opportunities for synergy:

- RL's adaptive decision-making can be combined with AI in Dispatch's predictive analytics to create a system that anticipates and reacts to user demand with precision.
- MAS frameworks can serve as the implementation backbone, enabling collaborative execution of RL and AI in Dispatch strategies across multiple elevators.
- Hybrid systems could prioritize user-specific needs (e.g., accessibility) while balancing energy efficiency and traffic distribution.

Energy Efficiency and Sustainability: The integration of these techniques can also address growing concerns about energy usage in modern infrastructure. By optimizing elevator schedules and routes, hybrid systems can reduce energy consumption without compromising user satisfaction.

10.2 Future Directions

Hybrid Model Development: Future research should focus on designing and testing hybrid frameworks that combine AI in Dispatch, RL, and MAS. Such systems would leverage the predictive power of AI, the adaptability of RL, and the collaborative efficiency of MAS to create more intelligent elevator management systems.

Real-World Testing: Implementing these hybrid systems in real-world scenarios is essential to evaluate their effectiveness. Challenges such as unpredictable traffic patterns, hardware limitations, and user behavior need to be addressed through extensive testing.

Advanced Metrics: Beyond traditional metrics like wait time and energy consumption, future work should also consider user satisfaction, system robustness, and cost-effectiveness to ensure practical applicability.

10.3 Conclusion

Optimizing elevator dispatch systems is vital for modern infrastructure to address the growing challenges of urbanization and technological advancement. This paper examined three key approaches:

- **AI in Dispatch:** Utilizes predictive analytics for proactive and efficient scheduling.
- **Reinforcement Learning:** Offers dynamic adaptability to real-time traffic patterns.
- **Multi-Agent Systems:** Enables decentralized and scalable management through collaboration.

Each method brings unique strengths to the table, and hybrid frameworks combining these techniques hold significant potential. By leveraging their synergies, elevator systems can become smarter, more efficient, and user-centric, paving the way for sustainable and intelligent building management.

11 Conlusion

Efficient elevator dispatching remains a critical challenge in modern urban environments as high-rise buildings become more prevalent. Traditional algorithms like First Come First Serve (FCFS), Shortest Seek Time First (SSTF), SCAN, and LOOK provide foundational frameworks, balancing simplicity, fairness, and efficiency under various conditions. However, each approach exhibits limitations, particularly in addressing dynamic, high-traffic scenarios.

The Group Control Algorithm, enhanced by predictive modeling, zoning, and real-time decision-making, demonstrates significant potential in minimizing wait times and energy consumption. It offers a scalable and adaptive solution for complex environments but requires advanced computational resources to handle real-time optimizations effectively.

Looking ahead, artificial intelligence (AI) and machine learning emerge as transformative technologies, enabling dynamic and context-aware dispatching systems. Hybrid algorithms, combining the strengths of traditional methods with AI-driven adaptability, promise further advancements in efficiency and equity. However, the success of these systems hinges on their ability to integrate seamlessly into real-time operations while accommodating diverse traffic patterns and user behaviors.

Future research should focus on:

Enhancing fairness in dynamic dispatching systems. Developing hybrid models that balance efficiency and user satisfaction. Exploring the ethical and practical implications of AI-driven

decision-making. By addressing these challenges, elevator dispatching systems can continue to evolve, meeting the growing demands of urban life and ensuring sustainable, user-centric vertical transportation solutions.

References

[1] Dergipark. 2024. Traffic Analysis and Optimization in Elevator Dispatching Systems. *Dergipark* (2024). https://dergipark.org.tr/tr/download/article-file/539296 Examines traffic analysis and optimization for elevator dispatching systems in Turkey..

[2] SPIE Digital Library. 2024. A Survey on Machine Learning in Elevator Control Systems. *SPIE Digital Library* (2024). An overview of AI solutions and machine learning applications in elevator systems..

[3] Vedant Misra. 2024. *Vedant Misra's Elevator Algorithms.* http://vedantmisra.com/elevator-algorithms/ A guide to general algorithms and problem definitions for elevator control..

[4] International Journal of Recent Technology and Engineering. 2024. Dynamic Dispatching of Elevators in Elevator Group Control System with Time-Based Floor Preference. *International Journal of Recent Technology and Engineering* (2024). https://www.ijrte.org/ Provides detailed insights on time-based elevator control under different traffic scenarios..

[5] Elevator Saga. 2024. *Elevator Saga.* https://play.elevatorsaga.com/ A practical approach to real-time algorithm testing and optimization problems..

[6] TheSaltree. 2024. *Elevator Scheduling Algorithms: FCFS, SSTF, SCAN, and LOOK.* https://dev.to/thesaltree/elevator-scheduling-algorithms-fcfs-sstf-scan-and-look-2pae Accessed: 2024-11-21.

Received 19 November 2024; revised 15 December 2024; accepted 5 January 2025