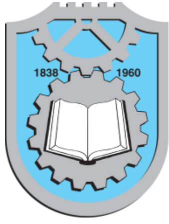
Vol. 07, No. 3 (2025) 1991-2000, doi: 10.24874/PES07.03A.025



Proceedings on Engineering Sciences

www.pesjournal.net

BISEARCHINS-DRIVEN ECO-FRIENDLY HYBRID RIDESHARE SYSTEM FOR SUSTAINABLE AND EFFICIENT URBAN TRANSPORTATION

M. Prasad[1]
Raja Rao PBV
N. Lakshmi
G. L N V S Kumar
B. Jagadeesh
Pokkuluri Kiran Sree
B. Srinivas
K. Ajita Lakshmi

Received 26.03.2025
Revised 30.05.2025
Accepted 05.07.2025

Keywords:

Ridesharing, Sustainable Transportation, Hybrid System, Dynamic User Insertion, Efficient Route Planning, BiSearchIns

A B S T R A C T

Ridesharing is an essential component of sustainable and economical urban transportation, facilitating the sharing of vehicles among multiple users while reducing traffic congestion and carbon emissions. Efficient scheduling remains a significant challenge in ensuring the widespread adoption of such systems. A hybrid ridesharing system has been developed that combines reliable route planning with dynamic user insertion to enhance urban mobility. The system focuses on two critical aspects: providing frequent users with predictable, pre-scheduled routes to minimize individual vehicle usage and enabling seamless integration of new users into ongoing shared trips. This approach reduces the effort required to search for rides, decreases waiting times, and maximizes vehicle occupancy. Group schedules and driver details are communicated promptly, ensuring effective coordination. Furthermore, vacant seats in vehicles during active trips can accommodate real-time user requests without disrupting pre-existing schedules. To achieve this, an efficient insertion method, BiSearchIns, has been implemented to process shared trip queries with high responsiveness. The proposed system demonstrates the formation of ridesharing groups through offline scheduling and the real-time integration of dynamic requests, contributing to sustainable and environmentally friendly transportation practices.

© 2025 Published by Faculty of Engineerin

1. INTRODUCTION

A viable strategy for reducing energy use and easing traffic congestion while meeting commuter needs is ridesharing. Private vehicle ridesharing, sometimes referred to as carpooling, has been researched for years as a solution for people's daily journeys, such as those from home to work (Dastani et al., 2024). In highly

Corresponding author: M. Prasad
Email: prasads.maddula@gmail.com

congested city regions, hailing a cab during rush hour has grown increasingly challenging in recent times. Of course, cab ridesharing is taken into consideration as a possible solution to this new transportation issue.

Recurring ridesharing has been applied widely. It refers to a ridesharing arrangement where individuals share rides on a regular, often daily basis, following consistent routes and schedules. This type of ridesharing is particularly common among commuters who travel to and from work or school. The concept involves setting up a reliable and repetitive schedule for sharing rides, as opposed to one-time or occasional ridesharing. For example, if a group of coworkers live in the same neighborhood and work in the same office, they might set up a recurring ridesharing arrangement where they carpool together every weekly morning and evening. There is no opportunity for users who have random routes and random schedules to use this type of service. Cab ridesharing can meet this demand; however, it is more difficult to operate than recurring ridesharing because the locations of cabs and the requests from riders are unpredictable and extremely dynamic. In particular, queries can be sent at any time and from any location in real-time and cabs are always on the road, collecting and delivering riders. Its final destination is determined by the riders, who may travel anywhere in the city (Ong et al., 2024). This type of research ignores the consideration of regular users with similar routes.

Actual observation shows that there are not enough effective ridesharing systems, even though numerous websites like Uber, Lyft, and Grab developed to provide ridesharing services. Only if a ridesharing system is simple, secure, adaptable, and effective will it be extensively utilized. In a standard ridesharing arrangement, drivers will post their travel ahead of time, and riders will have to look for any schedules and routes that fit their needs. The majority of the time riders must conduct multiple searches to locate a satisfactory match in terms of scheduling and route. It's thought that ridesharing has become less popular due to the escalating intricacy of work schedules.

Thanks to the widespread use of GPS-enabled gadgets, an increasing number of users record their daily mobility and are willing to share it with others. Their mobility patterns are documented in these logs. This gives us the chance to find out more about the riders' everyday lives and make recommendations that are tailored to make their travel experiences better. For instance, possibly able to verify each rider's everyday commute path and promote ridesharing to a group of individuals who have comparable commutes. Ridesharing on a regular schedule and routes can be automatically deployed (Sakthivelu et. al., 2024).

In this research, proposes a hybrid scheduling approach for the dynamic cab ridesharing problem to maximize the number of served passengers, reduce the number of cabs needed, minimizing the travel distance, and efficient response time. Imagine a cab firm that runs a dynamic ridesharing cab service. Regular ridesharing groups are established in advance and assigned to specific drivers in advance. Using a mobile phone, irregular riders book a ridesharing request in real-time. Each query specifies the trip's starting and ending points as well as time periods that limit the riders' availability for pickup and drop-off. When the operation center receives a new request, it will send a suitable cab that can accommodate both the existing riders and the new ones. To urge regular users to join a ride share even for the longest portion of their trips to construct ridesharing groups beforehand by modifying previously proposed algorithm a VC-Growth (Vu, 2023).

A route having road segments that a person commonly travels at consistent predetermined periods is referred to as a frequent route. Secondly, once the trip of a ridesharing group is activated the driver can accept new requests in real time if there is still a vacant seat in the vehicle. It comes to solve the problem of scheduling among cab drivers and real-time requests. A shared trip request consists of pickup, drop-off locations and time window constraints. Scheduling is in charge of finding a series of locations to collect and disembark riders. When a new request is sent to the system, pickup and drop-off locations will be inserted into the schedules of the current cabs to find the cab that satisfies some optimization objective. For large-scale ridesharing platforms, the insertion function is ineffective. In practice, insertion becomes a hindrance when processing a high volume of queries since it requires at least quadratic time. But most current algorithms strive to find the most optimal schedule, thus they still require a significant amount of processing time. According to a survey by Dynatrace, users expect mobile apps to response within two seconds. A delay beyond this threshold can lead to dissatisfaction and app abandonment. To address this problem dynamic programming and approximation strategy are utilized for insertion, which help lower time complexity from quadratic to linear. The contributions of this study can be summarized as follows:

- Devising an efficient mining method named aVC for forming ridesharing groups in an offline fashion based on the maximal frequent routes of users.
- Proposing a dynamic programming approximate algorithm named biSearchIns for insertion a new query in an online fashion with multi objectives mentioned above and linear time complexity.
- Conducting experiments to evaluate the performance of the proposed algorithms in comparison with the current ones and non-ridesharing method in terms of the response time, the number of served people, and the number of cabs needed.

The structure of the paper is as follows: Section 2 provides a review of relevant literature. Section 3 introduces a method for generating valid ridesharing groups in an offline manner and describes a novel dynamic programming-based approximation algorithm for real-time query insertion into a cab's schedule. Section 4 evaluates the performance of the proposed algorithms by comparing them with existing approaches, focusing on execution time, the number of cabs required, and the number of queries or passengers served. Finally, Section 5 presents the concluding remarks.

2. LITERATURE REVIEW

The integration of dynamic ridesharing systems has garnered significant research interest due to their potential to optimize transportation and reduce congestion. Aissat et al. (2015) proposed a novel approach to real-time ridesharing by utilizing meeting locations based on a buckets approach, which helps identify the most efficient meeting points for users. Mitropoulos et al. (2021) conducted a systematic literature review on ride-sharing platforms, focusing on user factors and barriers, which provides valuable insights into the challenges and opportunities in scaling such services. Large-scale dynamic ridesharing services, such as T-Share (Shuo et al., 2013), leverage real-time data to match users efficiently, while Liu et al. (2021) introduced mT-Share, which incorporates mobility-awareness for better dynamic routing in urban areas. The extraction of regular routes from GPS data for recommending ridesharing options was explored by He et al. (2012), highlighting the importance of route mining for personalized recommendations. Vu (2023) proposed a novel framework for ridesharing services, emphasizing the need for improved algorithms and technologies to enhance service efficiency and user satisfaction. Furthermore, efficient mining of group patterns from user movement data has been recognized as a key factor in improving ridesharing performance, as explored by Wang et al. (2005). Huang et al. (2014) and Tong et al. (2018) demonstrated the use of large-scale real-time ridesharing and route planning approaches to enhance service guarantees and optimize mobility in road networks.

Recent advancements in intelligent systems have significantly improved road safety and vehicular communication. For instance, Pokkuluri et al. (2024) developed a deep learning-based method for detecting vehicle crashes on roads, improving real-time traffic monitoring and response strategies. Similarly, Prasad et al. (2024) introduced an emergency message prioritization and scheduling approach in vehicular ad hoc networks (VANETs), ensuring timely communication during critical situations, thus enhancing road safety and system reliability. Additionally, Satti et al. (2022) proposed a unified approach for detecting traffic signs and potholes on Indian roads, utilizing image processing techniques for safer navigation. Furthermore, Prasad et al. (2024a) explored robust authentication strategies for secret key exchange in machine-to-machine communications, contributing to the security of intelligent transportation systems. These studies collectively highlight the growing intersection of machine learning, communication technologies, and safety protocols in enhancing modern transportation systems.

3. METHODOLOGY

3.1 Creation of ridesharing group in offline fashion

This section presents the algorithm named aVC for generating regular ridesharing groups from the riders' frequent routes. A frequent route of a rider ui is presented by a series of points $ms_{ui} = \{\langle p_k, t_k \rangle\}_{k=1}^{n}$ where $p_k=\langle x_k, y_k \rangle$ represents the position at time t_k. Let $DB = \bigcup_u ms_{ui}$ be the set of frequent routes the riders ui (i >1). The requirement is to discover regular ridesharing groups based on similarity of the routes.

If there are little differences in time and space between two movements, two people's emotions are said similar. If the distance between two persons' positions at a given time is not greater than a predetermined value maxDis, they are said to geographically close. Assume there is a time interval $[t_k, t_k+\Delta]$, it is said to a valid interval of a group G if and only if all the riders are close at time t_k, $k \in [1, \Delta]$ but not close at time points right before or after that interval, and the duration of that interval must be at least the duration minDur.

Consider a group G with valid intervals $I_1,..,I_n$ and lengT as the total travel length in the set DB, the weight of group G. If the weight is not less than a threshold minW (i.e., WsG >= minW), the group G is defined as a valid group. The group G makes a ridesharing group <G , minDur, maxDis > if a valid interval exists in G. Denote G_k be a candidate group with k riders and VG_k be valid groups ($k \geq 1$). The algorithm aVG is described by pseudo code in Figure 1. The goal of this approach is to accelerate the execution time by improving upon the previous method VC-Growth in and at the same time achieve the maximal ridesharing groups that fit the capacity of a cab.

```
def generate_ridesharing_groups(riders, max_distance, min_duration, min_wetness):
   """Generates valid ridesharing groups with two riders each.
   Args:
      riders: A list of riders.
      max_distance: Maximum distance between riders in a group.
      min_duration: Minimum duration for a valid group.
      min_wetness: Minimum wetness level for a valid
```

```
group.
   Returns:
        """A list of valid ridesharing groups.
   groups = []
   for i in range(len(riders)):
      for j in range(i + 1, len(riders)):
         group = (riders[i], riders[j])
         group_weight = 0
         for t in range(len(riders[0].positions)):
            if is_close(riders[i].positions[t],
              riders[j].positions[t], max_distance):
               group_weight += 1
         if group_weight >= min_duration:
            group_wetness = group_weight /
                                len(riders[0].positions)
            if group_wetness >= min_wetness:
               groups.append(group)
   return groups

def is_close(position1, position2, max_distance):
   """Checks if two positions are within the maximum distance.
   Args:
      position1: The first position.
      position2: The second position.
      max_distance: The maximum allowed distance.
   Returns:
        """ True if the distance between the positions is less than or equal to max_distance, False otherwise.
    return distance(position1, position2) <=
                                max_distance
```

Figure 1. Pseudo Code for Generation of valid ridesharing groups with quantity 2

The graph VGgraph (V, E) is constructed to generate maximal valid ridesharing groups in consideration of cab's capacity cap. That means, subgroups of a group are not generated. The set of valid 2-groups (VG2) is produced by procedure VG2Gen (). The set of vertices V consists of the individuals present in VG2. A directed link in E between two vertices in V is attached valid time interval of 2-groups. For operation convenience, a directed edge represented by two persons is pointed from the lower index vertex to the higher index vertex. A depth-first approach to traverse the graph to identify all of the maximal valid groups.

The prefix vertices of a vertex u are represented by Pru, and their associated edges are denoted by E(Pru). This pair (Pru, E(Pru)) serves as input for identifying valid groups compatible with vertex u. The process of finding valid groups entails a depth-first traversal of the Ggraph, starting from a vertex and exploring its descendants. This recursive algorithm generates valid groups. Vertices in the set V are sorted in descending order based on their number of neighbors. Consequently, users with higher connectivity are prioritized for inclusion in groups, increasing the likelihood of discovering large valid groups early on. This approach can potentially reduce the overall execution time, as detailed in the pseudocode (Figure 2). Because no rider can appear in more than one ridesharing group, if a ride belongs to a group that has been created, it will be eliminated from consideration for a later new group.

Each time, a valid group Gk is kept in the set sG. After obtaining all the maximal valid groups, Generate the ridesharing groups with cab's capacity constraint and store in the set RG. A ridesharing group Gk has the form <lp,ld,|Gk|,tl>, in which the number of riders in Gk must be less than or equal the cab's capacity cap, the same pickup location lp and same droff-off location ld for every rider in the group and the latest time lt required to arrive at the destination. Finally, the algorithm aVC return the ridesharing groups in sG that the driver is in charge of activating them.

```
def generate_ridesharing_groups(graph, max_distance, max_duration, min_wetness, capacity):
   """Generates ridesharing groups based on the given criteria.
   Args:
      graph: The graph representing the ridesharing network.
      max_distance: Maximum allowed distance between pickup and drop-off locations.
      max_duration: Maximum allowed duration for a ridesharing trip.
      min_wetness: Minimum required wetness level for a ridesharing group.
      capacity: Maximum capacity of a ridesharing vehicle.
   Returns:
      """ A list of ridesharing groups.
      # Sort vertices in decreasing order of number of neighbors
   vertices = sorted(graph.vertices, key=lambda
                    v: len(v.neighbors), reverse=True)
   # Initialize set of valid groups
   valid_groups = set()
   # Iterate over vertices
   for vertex in vertices:
      # Generate conditional group Q
      conditional_group =
             generate_conditional_group(vertex, graph)
      # If conditional group is not empty
      if conditional_group:
         # Create a new valid ridesharing group
         new_group =
            create_ridesharing_group(conditional_group)
         valid_groups.add(new_group)
         # Remove vertex and its edges from graph
         graph.remove_vertex(vertex)
         graph.remove_edges(vertex.edges)
        # Adjust valid segments for remaining vertices
         adjust_valid_segments(graph, vertex)
         # If any edges remain after adjustment, establish Q's conditional cluster
```

```
        if graph.edges:
            conditional_cluster =
                establish_conditional_cluster(graph,
                                              conditional_group)
            # Create conditional VGgraph
            conditional_vggraph =
                 create_conditional_vggraph
                                             (conditional_cluster)
            # Recursively generate ridesharing groups for
the conditional cluster
            recursive_groups =
               generate_ridesharing_groups
                   (conditional_vggraph, max_distance,
                    max_duration, min_wetness, capacity)
            # Add recursive groups to valid groups
            valid_groups.update(recursive_groups)
    # Generate final ridesharing groups based on capacity
constraints
    final_groups = generate_final_groups
                                         (valid_groups, capacity)
    return final_groups
```

Figure 2. Pseudo Code Discovering maximal valid ridesharing groups with quantity greater than two.

3.2 Processing a ridesharing query in online fashion

This section introduces a novel scheduling method that involves in insertion function with linear time complexity, while previous studies require quadric complexity. The proposed approach utilizes dynamic programming insertion with greedy pruning strategy named gpiInsertion algorithm.

Let a cab be represented by tuple ca=<lo, a,ka>, where lo is the current location, a is the number of occupied seats and ka is the capacity of the cab.

A ridesharing query is represented by qi=<lp, ld, a, tl> where i≥1, lp is the pickup location, ld is drop-off location, a is the number of booked seats, and tl is the latest time by which the cab must arrive at the pickup location.

Since a ridesharing group generated in offline manner includes riders who have similar frequent trips, these rides can wait and get off at the same location. As a result, the schedule of a cab is initialized with a special query denoted as qo=<lp, ld, a, tl>.

The schedule of a cab ca is represented by Sca=<ca.lo, l1,...,lm> in which ca.lo is the current point of the cab and the sequence {l1, l2, ..., lm} consists of pickup and drop-off locations of the queries in Q, and any pickup point qi.lp must precede the drop-off point qi.ld in the sequence. The total distance the cab has to travel is the sum of the distance between two consecutive locations in the schedule. Without making misunderstood, when it is refer to the elements of a query, Given a set of active cabs and a set of real time ridesharing queries sorted in ascending order by the time they are sent to the system. Objective is search for a cab that meets the query's requirements with minimum additional distance incurred.

3.3 Algorithm for naive insertion

Basically, processing a real-time query means trying to insert q=<lp, ld, a, tl> into the schedule Scab of a cab to get a new schedule S1cab so that the additional distance is minimal and doesn't violate the constraints regarding the time of other riders. Owing to the consideration of all the possible pair (i,j) for insertion the time complexity of simple insertion function is O(n3).

3.4 Insertion algorithm based on binary search strategy

The algorithm named iIns for insertion that takes linear time complexity is Figure 3. First, still the algorithm checks if the cab has enough seats for the request or not. Then, the idea is that assume there's a fixed position j, the algorithm tries to find only position i<j so that the total distance the cab has to travel is minimum. Denote plj be the position to possibly insert the pickup location lp corresponding to Dj. The algorithm starts with the initializations in which S'cab is the best found schedule and its corresponding minimum additional distance δ and all other arrays and parameters. In case, at the time of receiving the request the number of vacant seats is not enough for the new request, the algorithm terminates. When j=0 it's impossible to insert lo before 0. Otherwise the algorithm keeps searching for the best insertion location. In case, the position to insert pickup point leads to the violation of the latest time at drop off location, then moves on to consider another position.

```
def find_insertion_position(cab, schedule, request):
    """Finds the insertion position for a new request in a
cab's schedule.
    Args:
        cab: The cab object.
        schedule: The current schedule of the cab.
        request: The new request to be added to the
schedule.
    Returns:
          """ The index of the insertion position, or -1 if no
valid position is found.
    # Initialize variables
    start = 0
    end = len(schedule) - 1
    mid = 0
    # Binary search to find the earliest possible drop-off
time
    while start <= end:
        mid = (start + end) // 2
        drop_off_time = schedule[mid]["arrival_time"] +
                                  schedule[mid]["travel_time"]
        if drop_off_time < request["due_time"]:
            start = mid + 1
        else:
            end = mid - 1
```

```
    # Check if the request can be inserted at any position after the binary search result
    for i in range(start, end + 1):
        ifschedule[i]["arrival_time"] +
                        schedule[i]["travel_time"] <
                                request["due_time"]:
            return i
    return -1
```

Figure 3. Procedure for finding the starting point to insert l_d

```
# Input:
#   cab: The cab object
#   schedule: The current schedule of the cab
#   request: The new request to be added to the schedule
# Output:
#   new_schedule: The updated schedule with the new request
def schedule_cab(cab, schedule, request):
    new_schedule = schedule.copy()
    best_insertion_index = -1
    best_insertion_cost = float('inf')
    # Check capacity constraint
    if cab.capacity + request.passengers >
                                cab.max_capacity:
        return new_schedule
    # Find the best insertion point
    for i in range(len(schedule) + 1):
        # Calculate the cost of inserting the request at this position
        cost = calculate_insertion_cost(schedule, request, i)
        # Update best insertion if this position has a lower cost
        if cost < best_insertion_cost:
            best_insertion_cost = cost
            best_insertion_index = i
    # Insert the request at the best position
    if best_insertion_index != -1:
        new_schedule.insert(best_insertion_index, request)
    return new_schedule
def calculate_insertion_cost(schedule, request, insertion_index):
    # Calculate the cost of inserting the request at the given position
    # This function should consider factors like:
    #   - Distance between the request's pickup and drop-off locations
    #   - Waiting time for the request
    #   - Impact on the overall schedule (e.g., delays to other passengers)
```

Figure 4. The Pseudo Code biSearchIns() for insertion

To improve the efficiency of finding the optimal insertion point compared to algorithm *iIns*, the *biSearchIns* algorithm (Figure 4) leverages binary search. This is possible because the locations in the schedule are ordered by their latest arrival times (*duek*), enabling efficient searching. The first searching point is found by the procedure startP(). First, initialize the variables *start* and *end* to represent the range of points in S_{cab}. After that, binary search is performed to find the first position where l_p can be inserted. First, check if inserting l_p at the midpoint *mid* results in the estimated cab arrival time is greater than or equal to the earliest possible pickup time at point l_p. If this condition is not satisfied, the possible insertion interval must be from *mid*+1 to *end*. Conversely, if the condition is met, the interval to consider becomes from *start* to *mid*.

Then from start to end, check if inserting the l_p result in the estimated cab arrival being less than the latest pickup time. In case, there is no position found, -1 is returned.

The biSearchIns algorithm finds the position where the pickup location l_p can be inserted and algorithm works as the previous algorithm iIns(), but starts considering the insertion point l_d from the position *start*. The algorithm has a complexity of O(n) similar to iIns. However, it is faster because it prunes in advance the position that pickup location l_p is impossibly inserted. This greedy pruning strategy might lead to the most optimal schedule, the tradeoff is an improvement in the algorithm's efficiency. It is worth accepting this tradeoff, because users tend to want an immediate response after clicking.

4. RESULTS AND DISCUSSIONS

This section evaluates the newly proposed algorithms aVC and biSearchIns on synthetic datasets and compare with the current algorithms.

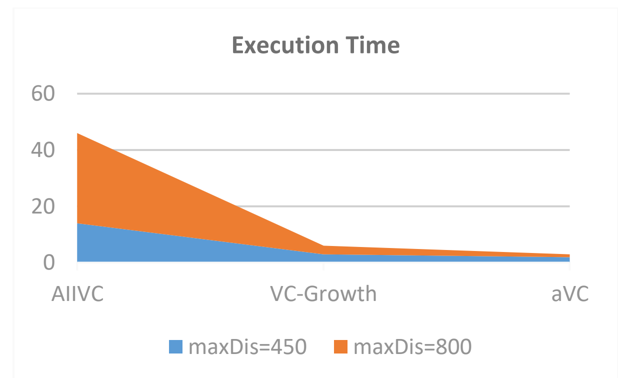


Figure 5(a). Algorithms' performance with respect to Execution time (*second*)

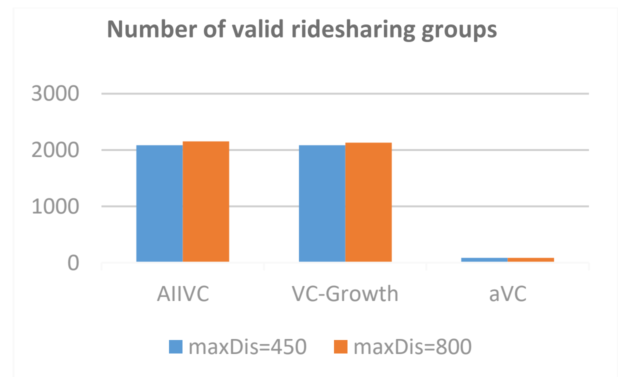


Figure 5(b). Algorithms' performance with respect to Number of valid ridesharing groups

To evaluate aVC frequent routes of riders are generated on a grid space. Denote a dataset, for instance D30_L12, means 25 routes and the maximum total points of a trip is 13. The capacity of a cab is set 4 as default. Set as default minDur= 3, minWei= 0.4 and maxDis varying with values 450 and 800.

As illustrated in Figure 5(a), the execution of the algorithm aVG better than two other ones since the algorithms AllVC and VC-Growth produce all of the subgroups of valid groups. It leads to the explosion of the search space and therefore the number of found groups which are redundant to ridesharing services because specified objectives are to make as big ridesharing group as possible and travel together as long as possible so that the number of served people increase while decreasing the number of cabs needed. Conversely, the algorithm aVC creates as quickly as possible the biggest valid groups, based on which to output the capacity-limited ridesharing groups (Figure 5(b)). When the minDis increases the number of groups decreases. This is because when the common part of riders' routes is not long, the rider is invalid to join the group.

To evaluate all of the insertion algorithms, apply synthetic dataset with 45 online ridesharing requests, available groups being 10. If there is no suitable ridesharing group to serve an online ridesharing request, an unscheduled cab will be used to serve that request. The main goal of ridesharing service is to especially address traffic congestion during the rush hour in the morning when people commute to work and school. Therefore, data for 50 requests mostly includes destinations with times ranging from 07:30 AM to 10:30 AM.

Implement and evaluate the performance of the algorithms nIns, quadIns, iIns, and biSearchIns. The total response time for the 55 requests is used as the criteria for comparing the algorithms as well as non-ridesharing method (i.e., a person uses one cab and 55 requests need 55 cabs).

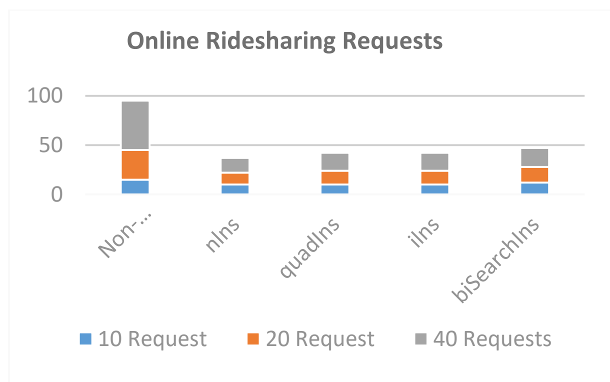


Figure 6. The number of cabs as the function of online ridesharing requests

The first test compares the number of cab needed to serve all online ridesharing requests. The number of available ridesharing groups is increased from 5 to 10. In each case, the number of online ridesharing requests is increased from 10 to 20 and finally 40. In Figure 6, the experimental results show that all the algorithms for inserting online requests into the existing schedule require fewer cabs compare to non-ridesharing service.

The next test compares the total response time for all online ridesharing requests with some number of available ridesharing groups and online request varying from 14, 28, and 56 (Figure 7).

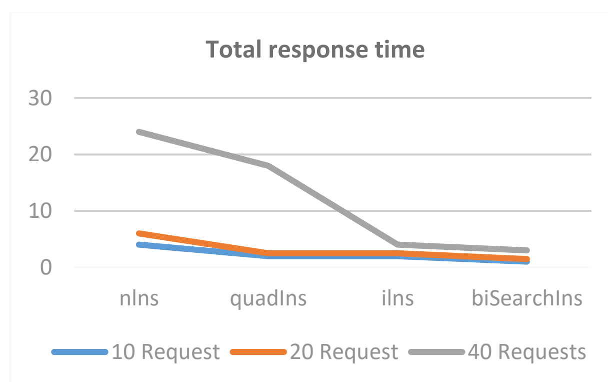


Figure 7. Total response time (*ms*) as the function of the number of online requests

The results show that the naive algorithm nIns takes the most execution time. The algorithms with linear time complexity are far faster than nIns and quadIns especially proposed algorithm is even superior to inIns. When the number of requests increases, the efficiency of biSearchIns becomes more apparent despite requiring more cabs compared to the exiting one since this method prunes some possible insertion positions with expectation of generating an acceptable schedule as quickly as possible. Thereby, the found schedule is not the most optimal one but in turn the performance in terms of time bas been significantly improved. Consequently, the proposed algorithm is efficient when dealing with larger requests.

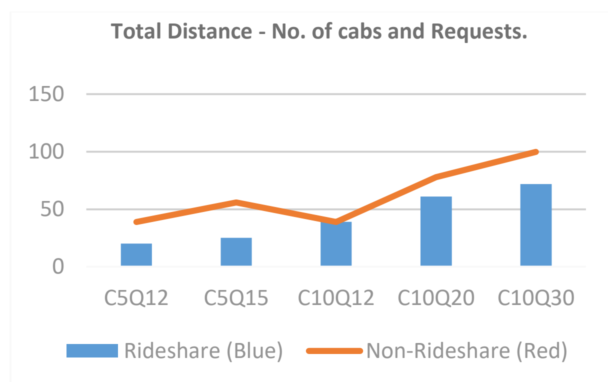


Figure 8. Total distance as a function of the number of cabs and requests

After having the ridesharing groups obtained in offline fashion, try to insert the new queries in the schedule. In terms of calculating the saved distance, when deploying a system in which some cabs are activated with offline ridesharing groups and then receive online queries. For the sake of computation simplicity, assume that originally all regular riders travel alone before sharing a regular ride with others. Figure 8 compares the total distance in cases of having ridesharing (Rideshare) and non-ridesharing (Non-Rideshare).

Number of cabs and the total queries sent to the system. Denote C10Q20 be the dataset with 10 cabs and 20 real-time queries. The result indicates that a significant amount of mileage is saved when riders share their ride. However, the experiment also indicates when the number of queries increases and the number of available cabs is high the difference between ridesharing and non-ridesharing is not much (see the case C10Q12). Try to increase the number of queries three times, more than the number of cabs, ridesharing shows better results than non-ridesharing in terms of the number of served queries and the number of cabs needed.

The BiSearchIns algorithm enhances the efficiency of query insertion by reducing its time complexity from quadratic ($O(n^2)$) to linear ($O(n)$) through the use of a binary search technique combined with a greedy pruning strategy. This approach greatly expedites the identification of optimal positions for integrating new requests into current schedules. Additionally, the system employs the aVC algorithm to establish predefined ridesharing groups offline, thereby minimizing the computational load during real-time operations. These predefined schedules effectively handle recurring ridesharing requests, leaving only a smaller volume of dynamic queries for real-time processing. This hybrid model ensures faster response times for both predefined and dynamic requests. Furthermore, dynamic programming techniques are employed to seamlessly integrate real-time user requests, reducing delays associated with route recalculations. Experimental findings, as illustrated in Figures 6 and 7, reveal that the proposed algorithms considerably surpass conventional methods in terms of response time, highlighting their capability to process real-time requests with remarkable speed.

5. CONCLUSIONS

There are social and environmental benefits to ridesharing in general and private cab ridesharing, such as lower energy use and meeting transportation demands. This research addresses the challenges of dynamic ridesharing by proposing a hybrid scheduling approach. An offline algorithm (aVC) generates ridesharing groups based on rider route similarities and cab capacity constraints. Subsequently, an efficient online query insertion algorithm (BiSearchIns) dynamically integrates new ride requests into existing schedules. Experimental results demonstrate significant improvements in key performance indicators, including reduced response times, minimized cab usage, and shorter travel distances compared to existing methods. This optimized system not only enhances operational efficiency but also contributes to a more sustainable urban transportation system by reducing energy consumption and associated emissions. Future research will focus on integrating a dynamic pricing model to further enhance system adaptability and user satisfaction.

References:

Aissat, K., & Oulamara, A. (2015). Meeting Locations in Real-Time Ridesharing Problem: A Buckets approach. In *Communications in computer and information science* (pp. 71–92). https://doi.org/10.1007/978-3-319-27680-9_5.

Dastani, Z., Koosha, H., Karimi, H., & Moghaddam, A. M. (2024). User preferences in ride-sharing mathematical models for enhanced matching. *Scientific Reports*, *14*(1). https://doi.org/10.1038/s41598-024-78469-1.

Liu, Z., Gong, Z., Li, J., & Wu, K. (2021). MT-Share: a Mobility-Aware dynamic taxi ridesharing system. *IEEE Internet of Things Journal*, *9*(1), 182–198. https://doi.org/10.1109/jiot.2021.3102638.

He, W., Li, D., Zhang, T., An, L., Guo, M., & Chen, G. (2012). Mining regular routes from GPS data for ridesharing recommendations. *Proceedings of the ACM SIGKDD International Workshop on Urban Computing*, 79–86. Beijing China: ACM. https://doi.org/10.1145/2346496.2346510

Huang, Y., Bastani, F., Jin, R., & Wang, X. S. (2014). Large scale real-time ridesharing with service guarantee on road networks. *Proceedings of the VLDB Endowment*, *7*(14), 2017–2028. https://doi.org/10.14778/2733085.2733106.

Shuo, N., Ma, Zheng, N. Y., & Wolfson, O. (2013). T-share: A large-scale dynamic taxi ridesharing service. *2022 IEEE 38th International Conference on Data Engineering (ICDE)*. https://doi.org/10.1109/icde.2013.6544843

Mitropoulos, L., Kortsari, A., & Ayfantopoulou, G. (2021). A systematic literature review of ride-sharing platforms, user factors and barriers. *European Transport Research Review*, *13*(1), 61. https://doi.org/10.1186/s12544-021-00522-1

Ong, Y. C., Protopapas, N., Yazdanpanah, V., Gerding, E. H., & Stein, S. (2024). Fair and efficient ride-scheduling: a preference-driven approach. *Journal of Simulation*, 1–17. https://doi.org/10.1080/17477778.2024.2334826

Pokkuluri, K. S., Sssn Usha Devi, N., Prasad, M., Raja Rao, P., Varma, C. P., & Ramesh Babu, G. (2024). Detection of vehicle crashes on roads using deep learning. *2024 2nd International Conference on Advancement in Computation & Computer Technologies (InCACCT)*, 126–130. Gharuan, India: IEEE. https://doi.org/10.1109/InCACCT61598.2024.10551202

Prasad, M., Teja, A. L. S., Sai, C. M., Gayathri, C., Sriya, G. L., & Pokkuluri, K. S. (2024). Emergency message prioritization and scheduling in vehicular ad hoc networks. *2024 Sixth International Conference on Computational Intelligence and Communication Technologies (CCICT)*, 205–211. Sonepat, India: IEEE. https://doi.org/10.1109/CCICT62777.2024.00042

Prasad, M., Sreenivasu, M., Lakshmi, K. A., Dhulipudi, R., Kumar, G. S. N., Paul, K. J., & Kumar, U. V. (2024). Robust Strategies for Authenticating and Exchanging Secret Keys in Machine-to-Machine Communications with Enhanced Security. In *Lecture notes in networks and systems* (pp. 209–221). https://doi.org/10.1007/978-981-97-4892-1_18

Sakthivelu, & Jayakrishnan, S. (2024). An energy-efficient ride-sharing algorithm using distributed convex optimization. *Stanford Digital Repository*. https://doi.org/10.25740/ss423gm7371

Satti, S. K., K, S. D., Maddula, P., & Ravipati, N. (2021). Unified approach for detecting traffic signs and potholes on Indian roads. *Journal of King Saud University - Computer and Information Sciences*, *34*(10), 9745–9756. https://doi.org/10.1016/j.jksuci.2021.12.006

Tong, Y., Zeng, Y., Zhou, Z., Chen, L., Ye, J., & Xu, K. (2018). A unified approach to route planning for shared mobility. *Proceedings of the VLDB Endowment*, *11*(11), 1633–1646. https://doi.org/10.14778/3236187.3236211

Vu, T. H. N. (2023). A novel framework for ridesharing services. *2023 3rd International Conference on Electrical, Computer, Communications and Mechatronics Engineering (ICECCME)*, 1–5. Tenerife, Canary Islands, Spain: IEEE. https://doi.org/10.1109/ICECCME57830.2023.10252288

Wang, Y., Lim, E., & Hwang, S. (2005). Efficient mining of group patterns from user movement data. *Data & Knowledge Engineering*, *57*(3), 240–282. https://doi.org/10.1016/j.datak.2005.04.006.

M. Prasad
Department of CSE
Shri Vishnu Engineering College for Women (A), Bhimavaram, India.
drmprasadcse@svecw.edu.in
ORCID 0000-0002-5092-9032

G. L N V S Kumar
Department of MCA
B V C Institute of Technology and Science (A), Amalapuram, India.
kumar4248@gmail.com
ORCID 0009-0001-4756-2241

B. Srinivas
Department of CSE
B V C Institute of Technology and Science (A), Amalapuram, India.
borsu136@gmail.com
ORCID 0009-0003-9866-0890

Raja Rao PBV
Department of CSE
Shri Vishnu Engineering College for Women (A), Bhimavaram, India.
rajaraopbv@gmail.com
ORCID 0000-0002-2054-6567

B. Jagadeesh
Department of ECE
B V C College of Engineering (A), Rajahmahendravaram, India.
bjagadeesh2020@gmail.com
ORCID 0000-0002-2038-8378

K. Ajita Lakshmi
Department of ECE
Shri Vishnu Engineering College for Women (A), Bhimavaram, India.
ajita.sansita.m@gmail.com
ORCID 0009-0002-3339-0650

N. Lakshmi
Department of CSE
B V C Engineering College(A), Odalarevu, India.
lakshminalla29@gmail.com
ORCID 0009-0002-0265-799X

Pokkuluri Kiran Sree
Department of CSE
Shri Vishnu Engineering College for Women (A), Bhimavaram, India.
hodcse@svecw.edu.in
ORCID 0000-0001-8601-4304