2016 17th IEEE International Conference on Mobile Data Management

A Flexible Real-Time Ridesharing System Considering Current Road Conditions

San Yeung
Department of Computer Science
Missouri S&T
Rolla, MO, USA 65409
Email: syq3b@mst.edu

Evan Miller
Department of Computer Science
Missouri S&T
Rolla, MO, USA 65409
Email: eamp68@mst.edu

Sanjay Madria
Department of Computer Science
Missouri S&T
Rolla, MO, USA 65409
Email: madrias@mst.edu

***Abstract*—Recently, ridesharing applications, such as Uber, have gained popularity by helping riders to save money. However, current real-time ridesharing systems with time-window constraints have not yet considered the impact of real-time road conditions. In this work, we propose transferring algorithms and incentive models to enhance the flexibility of an existing large-scale real-time ridesharing system. Transferring algorithms aim to transfer those users who experienced delay to other vehicles by using a pairwise bounding-box pruning technique. The incentive models avoid the use of transferring but instead compensate affected passengers with credits to maintain their satisfaction level in a different way. Extensive experiments were conducted using a real road network. Results show that our proposed algorithms and models are able to boost users' satisfaction level up to 90% under the influence of real-time road conditions.**

I. Introduction

Recently, real-time ridesharing applications such as *Uber* started to gain enormous popularity among common users in many countries [1]. For a nation such as the U.S that hosts more than 250 millions passenger vehicles [2], utilizing ridesharing not only helps alleviate traffic jams and air pollution, but also helps reduce users' traveling costs and time by making use of the empty seats in many vehicles.

Most existing works focus on static ridesharing in which requests were known in advance. Among the few works in real-time ridesharing systems [6]–[8], they did not consider the concept of real-time road conditions, such as traffic incidents and road blockage events. These road conditions received in real-time could induce unexpected delays to the optimal schedules matched to existing ride requests, thus causing violations to some users' pickup or drop-off time constraints.

In order to fill in this gap, we evaluate the impacts current road conditions would have on the schedules and proposed solutions to mitigate them in different ways. Our proposed solutions include an Optimal Transferring Algorithm, a Proactive Transferring Algorithm, and a Global Incentive Model. The transferring algorithms reduce the number of affected passengers in a schedule by transferring them to other vehicles; the proactive version transfers *non-affected* passengers as well. On the other hand, the incentive model compensates affected passengers in the form of incentive without using transferring. For experiment, we evaluate the performance of our algorithms on user satisfaction rate and computational cost.

The structure for the rest of this paper is as followed: section II will discuss related work, followed by the problem definitions and framework overview in sections II and IV. We will discuss the transferring algorithms and the incentive model in sections V and VI. Then, sections VII and VIII will conclude this paper with algorithms evaluation and conclusions.

II. Related Work

A. DARP and Real-Time Ridesharing Systems

The general class of the Dial-A-Ride Problem (DARP) encompasses carpooling and ridesharing problems. It is a NP-hard problem that generalizes the Vehicle Routing Problem with Pickup and Delivery (VRPPD) [3]. Primary works in DARP have a static characteristic, where all ride request are known a priori. For large instances. the problem is usually solved by using a two-phase heuristic [5].

Distinguished from DARP, a real-time ridesharing system (RTRS) is of a highly dynamic nature. Drivers in this model do not have set destination points. Users submit ride requests shortly before their departure and the system searches for the vehicle which minimizes the travel distance. Several existing works attempt to define a comprehensive RTRS by considering various constraints subject to time, capacity, and monetary perspectives [6]–[8]. Ma *et. al.* proposed a mobile-cloud based real-time taxi-sharing system using a heuristic-based dual-side searching algorithm [8]. Restraining from using heuristics, Huang *et. al.* proposed an efficient kinetic tree approach for optimal rideshare matching [7]. Among the proposed RTRS's, their algorithms lack the adaptability to mitigate the effect caused by real-time road conditions.

B. Transfer-Allowed Ridesharing

In transfer-allowed ridesharing (TAR), a user's ride request can be satisfied by multiple vehicles if no feasible rideshare plan exists in the single-vehicle version. The concept of TAR first appears in the work done by Hou *et. al.* [9]. They use a "store-and-forward" strategy to devise a TAR system that allows multiple vehicles to cooperatively serve one ride request. Coltin and Marco gave more detail on the algorithms

2375-0324/16 $31.00 © 2016 IEEE
DOI 10.1109/MDM.2016.37

of TAR in a static setting [10]. They proposed three heuristic-based rideshare plans for transfers: a greedy, an auction, and a graph-search-based approach. Similar with work in RTRS, existing work for TAR did not take real-time road conditions into consideration.

III. Problem Definitions

Let a road network $G_n =< V, E, W >$ represented by a vertex set V and an edge set E. Each edge $\{(u,v)|u,v \in V,(u,v) \in E\}$ is associated with a weight $W(u,v)$ indicating the traveling cost on edge (u,v). Given two nodes s (source) and e (destination) in the road network, a path $p(s,e)$ is a vertex sequence $(v_0, v_1, \cdots, v_k)$ where $(v_i, v_{i+1}) \in E$, v_0 = s and v_k = e. The path cost $W(p(s,e))$ = $\sum W(v_i, v_{i+1})$ is the sum of all edge costs along $p(s,e)$. We denote the shortest path cost from s to e as $d(s,e)$ = $min_P W(p(s,e))$ where P consists of all possible paths from s to e.

Definition 1: (**Ride Request**) A ride request $Q = (s, e, pw, dw)$ is defined by a source s and a destination e $\{s, e \in V\}$, a pickup time pw of when the user wants to be picked up from s, and a drop off time dw of when the user needs to be dropped off at e. The drop off time pw should be bounded by $d(s,e) \leq pw \leq (1+\epsilon)d(s,e)$ since it is impossible to travel faster than the shortest path cost in G_n. In addition, each ride request is associated with $Q.t$, the time when the request is submitted to the system.

For each ride request $Q_i = (s_i, e_i, pw, dw)$, a vehicle (driver), r_i, will be assigned to it using the kinetic tree matching algorithm [7] and its trip schedule will be updated.

Definition 2: (**Vehicle Trip Schedule**) A vehicle's *trip set* $R = Q_1, Q_2, \cdots, Q_n$ consists of all ride requests assigned to it, and the corresponding trip schedule $S = x_1, x_2, \cdots, x_k$ is a sequence of points where each point is either a source point s_i or a destination point e_i. It is further assumed that each vehicle will travel on the shortest path from x_i to x_{i+1}. Hence, the *trip cost* $d_T(x_i, x_j)$ between any two points (x_i, x_j) in a trip schedule is denoted as

$$d_T(x_i, x_j) = d(xi, x_{i+1}) + d(x_{i+1}, x_{i+2}) + \cdots + d(x_{j-1}, x_j). \quad (1)$$

The overall trip cost is then simply $d_T(x_1, x_k)$.

In previous RTRS, once a Q_i is assigned to a vehicle, the resulting schedule remains valid throughout the operation. However, a schedule S might become invalid when real-time road conditions are introduced into the context of RTRS.

Definition 3: (**RTRC**) A real-time road condition RTRC = $(type, t_s, t_e, W(u,v))$ where type can be traffic incident, congestion, road construction, or event. Estimated start time and end time are denoted by t_s and t_e, and W(u,v) is the extra weight (delay) imposed on edge (u,v) due to incident of $type$.

The problem of **Real-time Ridesharing with Real-time Road Conditions** (RTRS-RTRC) *is: Given a vehicle trip schedule S_i invalidated by a set of real-time road conditions, minimize the number of ride requests assigned to r_i that are affected by such set.*

In an invalid schedule S_i, when a RTRC impacts a Q_i, either the pickup or the drop-off time is violated by the delay induced by $W(u,v)$. For the first case, the driver is unable to pickup the user on time, *i.e.,* $d_T(r_i.\ell, s_i) + W(u,v) > Q_i.pw$ and $(u,v) \in p(r_i.\ell, s_i)$. For the second case, the user is already inside the vehicle but cannot be dropped off on time, *i.e.,* $d_T(s_i, e_i) + W(u,v) > Q_i.dw$ and $(u,v) \in p(s_i, e_i)$.

IV. Framework

There are four main components in our real-time ridesharing framework: communication, indexing, RTRC updates, and scheduling. The communication component is responsible for indexing a vehicle's location (gps-coordinates) using spatial grid index, finding an optimal match for a new ride request (submitted to the system via a mobile application) using kinetic tree scheduling algorithm, and resolving existing passengers' time constraints (if affected by RTRC updates) using our proposed transferring algorithms.

A. Spatial Grid Indexing

We decided to employ the spatio grid indexing technique used in [7]. For a given road network, a grid G is constructed by superimposing it on on the map region with a uniform cell size $G.s$. All vehicles in the system will be mapped to corresponding cells according to their current locations. When the system receives a new ride request Q_i in a grid cell g_i, instead of checking feasibility for all vehicles, it only needs to consider vehicles in g_i and within $\lceil \frac{w_i \cdot D}{G.s} \rceil$ cells from g_i where $w_i = Q_i.pw - Q_i.t$ (the maximum waiting time to be picked up) and D is the maximum speed a vehicle can travel in g_i.

B. Optimal Scheduling

Once the set of vehicle candidates is identified for Q_i, the system needs to assign it a vehicle r_i with minimum travel cost. It is accomplished by using the kinetic tree approach proposed in [7]. The system applies the kinetic tree algorithm by maintaining a *dynamic tree* structure to each candidate. The algorithm conducts a feasibility check before inserting s_i or e_i. For optimization, it uses the concept of *slack time* to speed up the process. Let x_j denote a node j in a kinetic tree schedule, then the maximum allowed detour for x_j, δ_j, is equals to:

$$\delta_j = \begin{cases} w_j - d_T(\ell, s_j) & \text{if } x_j = s_j \\ y_j - d_T(s_j, e_j) & \text{if } x_j = e_j \end{cases} \quad (2)$$

where $y_j = Q_j.dw - Q_j.t$ (maximum service time to be dropped off). Then, the slack time of x_j can be associated by $\Delta(x_j) = min(\delta_j, max_{i \in x_j.children}(\Delta(x_i)))$. More details of the kinetic tree insertion algorithm can be read in [7].

C. Real-time Road Conditions Update

The RTRC update component pushes real-time traffic updates to the ridesharing system. Based on the estimated start and end time, and the associated travel weight increase on road segments, the system updates the travel cost of the road network and identifies all vehicles of which their original schedules have become invalid due to the updates. We can easily determine whether S_i has become invalid by computing the new δ_j for each node j in S_i using the new travel costs.

If $\delta_j^{new} < 0$, then S_i is now invalid and we mark node j to be *affected* (unsatisfied). Otherwise, the vehicle will take the updated shortest path and δ_j will also be updated. For every invalid S_i, the system will insert it into a queue and executes transferring algorithm for each schedule.

V. Proposed Transferring Algorithms

The goal of transferring algorithms is to *transfer* an affected on-board passenger, Q_a, in r_i to another vehicle r_j in order to resolve the violation. It is possible because r_j might be able to take Q_a by a different route that r_i cannot due to its ride request assignments.

A. Transfer-Point Search

For the design of transferring algorithms, a critical sub-problem is *how to search for the optimal transfer point* via transfer-point search (TPS). We define an optimal transfer point as follows:

Definition 4: Let C_T be a set of transfer-point candidates along two consecutive nodes (x_j and x_{j+1}) in a schedule S_i for an affected passenger Q_a, the optimal transfer-point (tp_o) is a point $tp \in C_T$ that results in minimum travel cost increased for r_i, *i.e.,* $tp_o = min_{C_T} W(p(x_j, t) + p(x_{j+1}, p))$.

1) BFS-TPS: We propose two variations of TPS. The first one is *Breadth-first-search* based (BFS-TPS). We must ensure two properties for TPS operations: (1) TPS should only be executed when we know that an alternative route indeed exists that will satisfy $Q_a.dw$. (2) The addition of a tp_o into S_i must not induce new violations. Let's consider the illustration in figure 1 that shows the current traveling schedule of r_i, $S_i = x_j, x_{j+1}, x_a$, where x_a is the affected node. To satisfy property (1), the condition of $d(\ell, x_a) \leq x_a.dw$ must hold to indicate that an alternative route (dashed-line in the figure) indeed exists to satisfy x_a. Then we update the new slack time δ_{x_a} to be $y_a - (d_T(s_a, \ell) + d(\ell, e_a))$. To ensure property (2), we must assign the path a slack time $\Delta(\ell, x_j) = min(\Delta(\ell), \delta_{x_a})$ to prevent new violations in S_i.

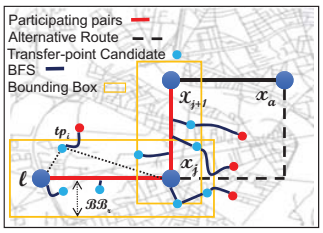

Fig. 1. An illustration of BFS-TPS and BB-TPS for retrieving the optimal transfer point for the affected node x_a.

The BFS-TPS procedure is presented in algorithm 1. For each vertex v, BFS is used to search all adjacent vertexes from v that meet the candidate requirement in line 13 (lines 3-15). The requirement basically says that a vertex, v_{adj}, can only become a valid candidate if the extra travel cost induced by such vertex is within the maximally allowed slack time for $p(x_j, x_{j+1})$ (illustrated by the travel path of $\ell \rightarrow tp_i \rightarrow x_j$ in figure 1).

Algorithm 1 Breadth-first-based Transfer-point Search

Input: Path (x_j, x_{j+1}) and its slack time $\Delta(x_j, x_{j+1})$, Q_a
Output: A tp_o with successful transfer vehicle match or $\emptyset$

1: $R_(x_j, x_{j+1}) = \emptyset$, $C_T = \emptyset$, $Q_{C_T} = \emptyset$, $tp_o = \emptyset$
2: Let $R_(x_j, x_{j+1})$ = all vertexes on $p(x_j, x_{j+1})$ in G_n
3: **for** $v \in R_(x_j, x_{j+1})$ **do**
4: Q_{C_T} = new queue()
5: Q_{C_T}.enqueue(v)
6: v.visited = **true**
7: **while** Q_{C_T}.isNotEmpty() **do**
8: $v_{current}$ = Q_{C_T}.dequeue()
9: **for** $v_{adj} \in v_{current}.adjacentVertices()$ **do**
10: **if** v_{adj}.unvisisted() **then**
11: v_{adj}.visted = **true**
12: ΔT=$d(x_j, v_{adj}) + d(v_{adj}, x_{j+1}) - d(x_j, x_{j+1})$
13: **if** ($\Delta T \leq \Delta(x_j, x_{j+1})$ **then**
14: Q_{C_T}.enqueue(v_{adj})
15: C_T.add(v_{adj})
16: C_T.sortByIncreasingDistance((x_j, x_{j+1}))
17: **while** C_T.hasNext() **and** $tp_o = \emptyset$ **do**
18: tp_i = C_T.next()
19: Q_a = (tp_i, e_a, pw, dw)
20: **if** findTransferVehicle(Q_a) **then**
21: tp_o = tp_i
22: return tp_o

2) BB-TPS: Some adjacent vertexes would still be evaluated even if they do not satisfy the candidate requirement in BFS-TPS (red nodes in figure 1). Therefore, we propose a second TPS method using bounding-box (BB-TPS) to reduce the candidate search space. The underlying concept is to find a bound with radius BB_r from $p(x_j, x_{j+1})$ such that most points within the box will be a valid candidate. We can calculate BB_r from the midpoint of the path as illustrated from figure 2. Since we would like the travel cost of the two hypotenuses

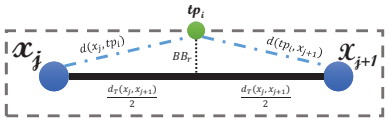

Fig. 2. Calculate BB_r from the midpoint of $p(x_j, x_{j+1})$.

combined to be less than the maximum travel cost allowed on $p(x_j, x_{j+1})$, and we know that both hypotenuses will have equivalent length if tp_i is projected proportionally from the midpoint of the path, then

$$2 \cdot \sqrt{(\frac{d_T(x_j, x_{j+1})}{2})^2 + (BB_r)^2} \leq d_T(x_j, x_{j+1}) + \Delta(x_j, x_{j+1})$$

and BB_r can be derived as followed:

$$BB_r = \sqrt{(\frac{d_T(x_j, x_{j+1}) + \Delta(x_j, x_{j+1})}{2})^2 - (\frac{d_T(x_j, x_{j+1})}{2})^2}.$$

Once the BB_r value has been obtained, a bounding box can be formed for each edge (u, v) in $p(x_j, x_{j+1})$ to retrieve nearby vertexes that are within the BB_r radius.

B. Optimal Transferring Algorithm

The basic transferring algorithm (BTA) can be developed by using either TPS method to mitigate impact caused by RTRC's. However, it is possible that the RTRC's did not affect all branches in a kinetic tree. Hence, we propose an *Optimal Transferring Algorithm* (OTA) that consists of two parts: kinetic tree updates and transfer of affected passengers.

The procedure for OTA is described in algorithm 2. The input to OTA consists of K_i, the kinetic tree maintained for r_i, and the road network with updated travel cost from RTRC's, G_n^U. In line 5, the function getOptimalSchedule(K_i) assigns S_i^{new} the schedule candidate in K_i with the *minimum* number of affected nodes and with minimum trip cost. The procedure for applying transferring algorithm to each schedule candidate is described in lines 8-35. At line 13, we must first determine the type of the affected node. The position of an affected node in S also matters. If x_a is immediately after ℓ, then it is impossible to find another vehicle to transfer because $d_T(\ell, x_a)$ is already the shortest path (line 17).

At lines 18-19, we let X_a be the set of nodes that positioned before x_a in S and let V_o be vertexes along $p(\ell, x_{a-1})$. We must first execute findTransferVehicle(Q_a) for each vertex in V_o because ideally they are already along the current travel path (lines 20-25). Else, we perform a TPS for each consecutive node-pair $p(x_j, x_{j+1})$ in X_a to find the optimal transfer-point (line 33). Before that, we must obey TPS property (1) by ensuring that there must exist an alternative route from either node (x_j or x_{j+1}) to x_a such that $x_a.dw$ is satisfied (line 28). Line 29-31 is for updating the slack time of the node-pair.

Algorithm 2 Optimal Transferring Algorithm

Input: Kinetic tree of r_i, K_i, Updated road network G_n^U
Output: A modified valid schedule S_i^{new}

1: $S_i^{new} = \emptyset$, $Q_N = \emptyset$, $Q_a = \emptyset$, $X_a = \emptyset$, $V_o = \emptyset$, $tp_o = \emptyset$
2: **Kinetic-tree Update**:
3: **for** each schedule candidate $S \in K_i$ **do**
4: updateSchedule(S, G_n^U) // record number of affected nodes
5: S_i^{new} = getOptimalSchedule(K_i)
6: **Optimal Transfer of Affected Passengers**:
7: **if** S_i^{new}.numberAffectedNodes > 0 **then**
8: **for** $S \in K_i$ **do**
9: Q_N = new queue()
10: Q_N.enqueueAll(S.getAffectedNodes())
11: **for** each affected node $x_a \in Q_N$ **do**
12: $tp_o = \emptyset$
13: **if** $x_a.type$ == pickup **then**
14: findAlternativePickupVehicle(x_a)
15: S.remove(x_a)
16: **else**
17: **if** x_a.previousNode != ℓ **then**
18: Let X_a contains all nodes with index $< x_a$ in S
19: $V_o = X_a$.getVertexes(ℓ, x_{a-1})
20: **while** $tp_o == \emptyset$ **and** V_o.hasNext() **do**
21: $x_o = V_o$.nextVertex()
22: **if** $d(x_o, e_a) \leq x_a.dw$ **then**
23: $Q_a = (x_o, e_a, pw, dw)$
24: **if** findTransferVehicle(Q_a) **then**
25: $tp_o = x_o$
26: **while** $tp_o == \emptyset$ **and** X_a.hasNextPair() **do**
27: $p(x_j, x_{j+1}) = X_a$.nextNodePair()
28: **if** $d(x_j, e_a) \leq x_a.dw$ **or** $d(x_{j+1}, e_a) \leq x_a.dw$ **then**
29: **if** $d(x_j, e_a) \leq x_a.dw$ **then**
30: $\delta_{x_a} = y_a - (d_T(s_a, x_j) + d(x_j, e_a))$
31: **else** $\delta_{x_a} = y_a - (d_T(s_a, x_{j+1}) + d(x_{j+1}, e_a))$
32: $\Delta(x_j, x_{j+1}) = \min(\Delta(x_j), \delta_{x_a})$
33: tp_o = TPS($p(x_j, x_{j+1}), \Delta(x_j, x_{j+1}), x_a$)
34: **if** tp_o != $\emptyset$ **then**
35: S.update(tp_o, x_a)
36: S_i^{new} = getOptimalSchedule(K_i)
37: return S_i^{new}

1) Transferring Algorithm with Minimum Affected Nodes: In OTA, only one schedule candidate is selected as the optimal schedule. However, the computation cost might be great if K_i contains many branches. An alternative solution can be derived from OTA by applying transferring algorithm selectively. Hence, we propose *Optimal Transferring Algorithm with Minimum Affected Nodes* (OTA-MAN). The difference lies in line 8: transferring algorithm is only applied to the set with minimum affected nodes.

C. Proactive Transferring Algorithm

In our proposed *Proactive Transferring Algorithm* (PTA), we transfer on-board passengers who have *no* time constraint violations in hope that by doing so more violations can be mitigated. The algorithm is explained in algorithm 3. The input is a schedule S_i that has been through BTA or OTA. For each x_a, let X_{tf} be a list that contains *transferable nodes* (nodes that have no time constraint violations, have not been transferred, and are not previously determined transfer-points) located in between ℓ and x_a. We first let x_j to be the second-to-last node in X_{tf} because we want to transfer x_{j+1} first. At line 10, we must perform a condition check to evaluate whether or not the transfer of any transferable nodes in between x_j and x_a could satisfy $x_a.dw$. The notation $x_{\bar{tf}_i}$ stands for a *nontransferable* node (will be discussed shortly) and the condition check essentially asks that the new route for $p(x_j, x_a)$ must include all nontransferable nodes. If this condition check passes, it means that some nodes in X'_{tf} (a list that contains transferable nodes in between x_j and x_a) could be transferred. From lines 12-31, we iterate through X'_{tf} and transfer each node x_k. If x_k is a pickup node, we can find it an alternative pickup vehicle (lines 14-19). If successful, x_k can be removed from S_i and the function S_i.satisfied(x_a) checks whether or not the updated schedule has now satisfied x_a. If it has, iterations through X'_{tf} can be stopped and we can move onto the next affected node. However, if alternative vehicle cannot be found, then we mark it as nontransferable and it will not be considered in future iterations. We also recheck the condition for x_j at line 24. If it fails, then we can stop iterating through X'_{tf} but iterate through X_{tf} backward instead

Algorithm 3 Proactive Transferring Algorithm

Input: Schedule S_i from previous transferring algorithms
Output: A modified schedule S_i^{new}

1: $Q_N = \emptyset$, $tp_o = \emptyset$, $j = 0$, continue = $\emptyset$
2: Q_N = new queue()
3: Q_N.enqueueAll(S_i.getAffectedNodes())
4: **for** each affected node $x_a \in Q_N$ **do**
5: X_{tf} = S_i.getTransferableNodes(ℓ, x_a)
6: $j = X_{tf}$.lastNodeIndex-1
7: **while** X_{tf}.get(j-1).preNode != ℓ **and** !S_i.satisfied(x_a) **do**
8: x_j = X_{tf}.get(j)
9: continue = **true**
10: **if** $d(x_j, x_{\bar{tf}_1}) + \cdots + d(x_{\bar{tf}_n}, x_a) \leq x_a.dw$ **then**
11: X'_{tf} = S_i.getTransferableNodes(x_j, x_a)
12: **while** X'_{tf}.hasNext() **and** continue **do**
13: x_k = X'_{tf}.getNext()
14: **if** $x_k.type$ == pickup **then**
15: **if** findAlternativePickupVehicle(x_k) **then**
16: S_i.remove(x_k)
17: **if** S_i.satisfied(x_a) **then**
18: continue = **false**
19: **else** x_k.status = **nontransferable**
20: **else**
21: tp_o = findTransferPoint(S_i, x_k)
22: **if** tp_o == $\emptyset$ **then**
23: x_k.status = **nontransferable**
24: **if** $d(x_j, x_{\bar{tf}_1}) + \cdots + d(x_{\bar{tf}_n}, x_a) > x_a.dw$ **then**
25: $j = j - 1$
26: continue = **false**
27: **else**
28: S_i.update(tp_o, x_k)
29: **if** S_i.satisfied(x_a) **then**
30: continue = **false**
31: return S_i^{new} = S_i

in line 7 until x_j is located right after ℓ, which means that no more transfer can be done.

VI. RTRC Incentive Model

The transferring algorithms provide no good solutions when no transfer vehicles can be found. Moreover, some passengers would still oppose to the idea of being transferred. Therefore, we propose the RTRC Incentive Model to address these problems by compensating users that are delayed in their travels. Let **F** be an existing ridesharing pricing scheme, then the final ride fare for a passenger Q_i, f_e^i, can be obtained as:

$$f_e^i = \mathbf{F}(Q_i) + \beta\eta - \gamma(\Delta T_i) \quad (3)$$

where $\mathbf{F}(Q_i)$ is the base fare for servicing Q_i, β is a contribute factor that takes an integer value in $[0, 1]$, η is a system-fixed incentive contribution that each passenger is charged when delay occurs, γ is a adaptive incentive rate (dollars per delay in time), and ΔT_i is the total delay time experienced by Q_i. β will take a zero value when RTRC updates cause no impact on current schedules; otherwise, it will take a value of one.

A. Local versus Global RTRC Incentive Model

From equation 3, the one constraint that needs to hold is $\gamma(\Delta T_i) \leq m(\eta)$, where m is the number of passengers in the vehicle. In the local incentive model, we can calculate the average incentive rate as follows: $\gamma = \frac{m_L(\eta)}{\sum_{i=1}^{n_L} \Delta T_i}$, where m_L and n_L denote the total number of passengers and the number of affected passengers in the vehicle locally, respectively. However, m_L is constrained by c_i, the vehicle capacity. Since m_L is unchangeable, we must have the system to increase η in order to have a reasonable incentive rate. Nevertheless, a larger incentive contribution would not be desired among users. A global incentive model could be a substitute in which we can get the incentive rate as follows: $\gamma = \frac{m_G(\eta)}{\sum_{i=1}^{n_G} \Delta T_i}$, where m_G and n_G denote the total number of passengers and the number of affected passengers in the entire ridesharing system globally.

B. Hybrid Models

GIM or LIM can also be combined with the proposed transferring algorithms to create hybrid models. For example, if an invalid schedule S_i still contains unresolved affected nodes after going through OTP and PTP, then we can apply GIM or LIM to give incentive to those still experiencing delay.

VII. Performance Study

A. Experimental Designs

The road network used is generated via the MNTG traffic generator on mid-Missouri region [11], consisting of 58,600 nodes and 61,300 edges. The algorithms are evaluated using 50,000 generated trip requests for 9,804 vehicles. The trips are generated by randomly selecting a starting location, ending location, and start time. Upon initialization, each vehicle is placed at a random vertex, and a constant speed of 48 km/hr is used for vehicles traveling in the road network when no road conditions are present. Road conditions reduce the speed of vehicles by an amount proportional to the delay it incurs on the affected road segments. Each vehicle is assigned a capacity of four with pickup and service constraints set at 10 minutes each. The framework is written in C++ and experiments are run on an Intel Core i7 (2.40 GHz) processor. The implementation is single threaded and allowed to use up to 12 GBs of memory.

B. TPS Algorithms Comparison

Using OTA as a baseline, performance of the BFS and BB-based TPS methods are compared. There are two choices to make when calculating a transfer: which point to make the transfer and which vehicle to transfer to. Both choices can be made in a greedy or optimal fashion. For the first choice, the optimal choice will consider all possible transfer points while the greedy choice will stop looking once a viable transfer point has been found. For each transfer point, the transfer vehicle can be decided similarly. The first letter of each algorithm corresponds to the choice of transfer point, and the second letter corresponds to the choice of vehicle in figure 3. Figure 3 shows the resulting detour (blue) averaged over 500 successful transfers for all eight algorithms. Note that

the greedy choices cause noticeably longer detours, but BFS and BB with the same optimal/greedy choice combination have very similar length detours. The computational cost (yellow) is

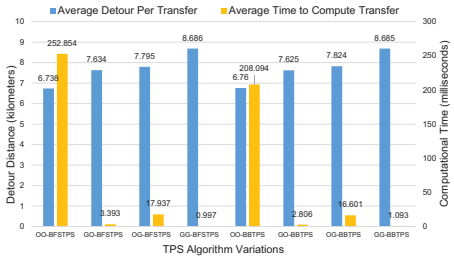

Fig. 3. Performance of various TPS algorithms (optimal/greedy choices).

also compared. Note that the greedy choice results in a much more noticeable speedup in terms of computational time.

C. Transferring Algorithms Comparison

Three versions of both OTA and PTA are compared against BTA in terms of proportion of unsatisfied users ($\frac{final\ \#unsatisfied\ users}{original\ \#unsatisfied\ users}$) and computational cost (averaged over 500 vehicle schedules that had potential transfers after RTRC updates) in figure 4. Each *Proactive Optimal Transferring Algorithm* (POTA) algorithm attempts proactive transfers after the corresponding OTA algorithm has been executed. OTA-TOP-K and POTA-TOP-K take the top "k" branches in the kinetic tree based on minimum number of affected nodes and attempt transfers only on them (for these experiments, k=6). GO-BFSTPS is used to find transfer points for these algorithms. Figure 4(a) compares all seven algorithms for

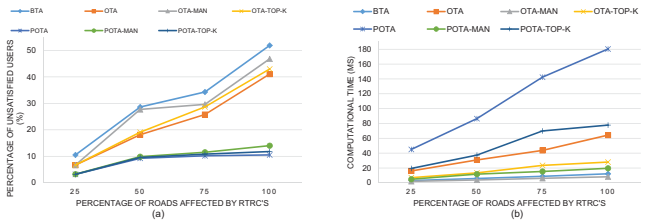

Fig. 4. Performance of various transferring algorithms.

different % of roads affected by RTRC's. Note that each POTA algorithm results in significantly fewer unsatisfied users when compared to its respective OTA algorithm. Figure 4(b) shows the result on the computational costs. For each vehicle, POTA takes a significantly longer time to execute. Although POTA-MAN's unsatisfaction is slightly higher than POTA-TOP-K, it is about 75% faster then POTP-TOP-K in average.

D. Incentive Models Comparison

Figure 5 shows the resulting compensation rate (γ) based on the initial passenger fee (η) for all three POTP algorithms and a baseline algorithm that does not execute any transfer algorithms for the global and local model, respectively. For the global model, compensation rates are calculated based on the entire set of vehicles (9804 for these experiments). In GIM,

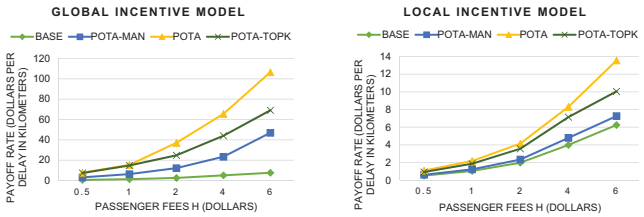

Fig. 5. Performance of Global and Local Incentive Models.

POTA-MAN outperforms the baseline algorithm, followed by a similar performance gap between POTA-MAN and POTA-TOP-K. While the payoffs may seem a bit high, it should be noted that many of the transfers allow passengers to reach their destinations within the initial constraints and thus do not result in necessary compensation. For LIM, the highest payoff rate reached by any algorithm is many times lower than GIM. The only algorithm that performs relatively similar in both models is the baseline algorithm. This is most likely a result of the fact that, in the local model, any vehicles that have no delays simply return the money to the passengers.

VIII. Conclusion and Future Work

In this paper, we fill in the gap of integrating current road conditions into an existing real-time ridesharing framework. POTA can satisfy affected passengers up to 90% within milliseconds, while POTA-MAN could be an ideal candidate with a competitive satisfaction rate when faster response time is desired. The incentive models act as a supplement to the transferring algorithms by compensating affected passengers. For future work, we can further improve the *stability* of the transferring algorithms by considering the *unpredictability* in RTRCs through the use of historical traffic data.

References

[1] "Drive with Uber." Uber. *Uber*. Web. 26 Dec. 2015.

[2] "National Transportation Statistics." *Bureau of Transportation Statistics*.

[3] Savelsbergh, M. W. P. 1985. Local search in routing problems with time windows. *Ann. Oper. Res.* 4 285-305.

[4] J.-F. Cordeau and G. Laporte, "The dial-a-ride problem: Models and algorithms," Ann. Oper. Res., vol. 153, no. 1, 2007.

[5] Z. Xiang, C. Chu, and H. Chen, "A fast heuristic for solving a large-scale static dial-a-ride problem under complex constraints," Eur. J. Oper. Res., vol. 174, no. 2, pp. 1117–1139, 2006.

[6] P. M. d'Orey, R. Fernandes, and M. Ferreira, "Empirical evaluation of a dynamic and distributed taxi-sharing system," in Proc. 15th *Int. IEEE Conf. Intell. Transp. Syst.*, Sep. 2012, pp. 140–146.

[7] Huang, Yan, et al. "Large scale real-time ridesharing with service guarantee on road networks." *Proceedings of the VLDB Endowment* 7.14 (2014): 2017-2028.

[8] Shuo Ma; Yu Zheng; Wolfson, O., "Real-Time City-Scale Taxi Ridesharing," in *Knowledge and Data Engineering, IEEE Transactions on*, vol.27, no.7, pp.1782-1795, July 1 2015

[9] Hou, Yunfei, Xu Li, and Chunming Qiao. "TicTac: From transfer-incapable carpooling to transfer-allowed I carpooling." *Global Communications Conference (GLOBECOM)*, 2012 IEEE. IEEE, 2012.

[10] Coltin, Brian, and Marco Veloso. "Ridesharing with passenger transfers." *Intelligent Robots and Systems (IROS 2014), 2014 IEEE/RSJ International Conference on. IEEE*, 2014.

[11] Mokbel, Mohamed F., et al. "MNTG: an extensible web-based traffic generator." *Advances in Spatial and Temporal Databases*. Springer Berlin Heidelberg, 2013. 38-55.