Snapcab: Urban Scale Context-Aware Smart Transport Using Adaptive Context Tries

Alistair Morris[1](✉), Constantinos Patsakis[2], Vinny Cahill[1], and Mélanie Bouroche[1]

[1] Distributed Systems Group, Trinity College Dublin, Dublin, Ireland
morrisa5@tcd.ie

[2] Department of Informatics, University of Piraeus, Piraeus, Greece

Abstract. Traffic gridlock has become a very familiar scene in cities due to the inefficiencies of existing transport systems. Context-aware dispatch has the potential to solve such congestion problems. Thus, this paper addresses the problem of realising large scale real-time taxi dispatch with service guarantees on road networks. Such a system requires the dynamic matching of travel requests made by passengers with appropriate taxis. Crucially this must occur while also ensuring the satisfaction of all waiting or travel times constraints. Results gained from simulations show that a novel approach, based on Adaptive Context Tries (ACT), provides fast response times, bounded complexity and thus scalability.

1 Introduction

A real-time taxi dispatch system attempts to solve the problem of matching taxis with passengers, hence avoiding taxis having to drive around looking for fares [1,2]. Potential passengers send travel requests, which include a *start* and an *end*, that respectively denote where a passenger wants to be picked up and dropped off. Recent research has extended the problem to support the concept of *ride sharing* between passengers to further increase efficiency of taxis [3,4].

Each request made by a passenger can contain two constraints: a *waiting time*, that defines the latest time a passenger wants to picked up by, and a *travel time*, that establishes the acceptable extra diversion time from the shortest duration for a given journey between a *start* and an *end*. In order to accept a request a taxi must satisfy all constraints; not only those of the newly encountered request, but also the requests it has committed to for all previous passengers [1].

However, providing such a dynamic taxi dispatch system at an urban scale presents a non-trivial problem. It involves a real-time matching algorithm that can quickly determine the best taxi that can satisfy an incoming request from a large set of choices [5]. Context-awareness offers one potential solution [1], but traditional context-aware middleware solutions using flooding, gossip or overlay based dissemination algorithms cannot scale due to their overhead [6,7].

In this paper, we show that Adaptive Context Tries (ACT) [6] can efficiently disseminate context and enable the distribution of previously-accepted travel

P.C. Vinh and V. Alagar (Eds.): ICCASA 2015, LNICST 165, pp. 31–40, 2016.
DOI: 10.1007/978-3-319-29236-6_4

requests, organised as a itinerary for each taxi. When a new request arrives, our system searches the trie to determine the best match, if any, and assign the request to the nearest taxi that can honour all its constraints. Note that this means the itineraries obtained are not guaranteed to be optimal but workable.

In the following: Sect. 2 provides an overview of traditional optimisation algorithms that solve related problems and argues for an alternative approach. Section 3 provides first a more formal problem definition and introduces our method, with some optimisations presented in Sect. 4. In Sect. 5 we experimentally compare our method to its points and the paper concludes with some ideas for future work in Sect. 6.

2 Related Work

Previous research mainly focuses on a single vehicle and a static scenario where a system knows the set of requests ahead of time [4,5,8]. However, this cannot provide a realistic approach in context-aware taxi dispatch problem at an urban-scale. Notably, earlier work highlights that this constitutes an NP-hard problem and therefore can only be solved for small sizes [9]. To address this, recent work proposes the use of context driven dynamic programming algorithms [10]. Hence, the processing of travel requests in real time becomes the main issue [11], as for any new request, the travel itinerary of each taxi, based on previously-assigned travel requests, needs to be processed. Crucially this means that only taxis at a distance smaller than a threshold w from the start can satisfy the waiting time constraint. This limits the amount of taxis considered. At this point a system could naively use brute-force to find the taxis which could accommodate the request. This requires the enumeration for all permutations and then checking whether the constraints are met. However, the complexity of this approach is exponential which means that it does not provide a scalable solution.

The *branch-and-bound* algorithm provides a more efficient method that systematically enumerates all candidate itineraries and organises them in an itinerary tree. It then estimates a lower bound of each partially constructed itinerary and stops building candidate itineraries with lower bounds greater than the known best solution [12]. Again, this approach can only solve small-scale problems as it also has exponential complexity in terms of response time [13].

Mixed integer programming presents an alternative approach [10,14]. This reduces the problem to finding the maximum/minimum of a linear function of non-negative variables subject to constraints expressed as linear equalities or inequalities. Although it is conceptually simple, many researchers state that the NP-hard set also contains this approach [15] so it also cannot scale.

3 Adaptive Context Tries (ACT)

This section first formalises the Real-Time Context-Aware Taxi Dispatch problem and it introduces the ACT structure that can maintain as well as update any calculations performed up-to-now and use them effectively when passengers issue new travel requests [6]. To deal with the highlighted challenges, our idea is based on a simple observation: A new legal itinerary accommodating a new request can be derived by extending any already existing current travel itinerary.

Based on this observation, the ACT-based solution to the taxi dispatch problem uses the following method: Firstly, it stores a legal travel itinerary for each taxi in a prefix tree (trie) structure at all times [6]. When a new request is received, the system checks if it can extend any travel itinerary to handle the new request. This method provides a promising approach because its incremental nature removes many redundant computations. Therefore, a system does not need to fully recompute an optimal legal travel itinerary for each new request, providing a non-optimal yet workable real-time approach. This outperforms current state of the art methods based on traditional unfeasible optimisation algorithms.

3.1 Formal Problem Definition

We consider a road network $G = \{V, E, W\}$ consisting of a vertex set V and an edge set E. Each edge $(u, v) \in E$ $(u, v \in V)$ has a weight $W(u, v)$ which indicates the travel time from (u) to (v), which is assumed to be a constant value.

Given two points s and e in the road network, a route π between them forms a vertex sequence $(v_0, v_1, \cdots, v_k)$, where (v_i, v_{i+1}) denotes an edge in E, $v_0 = s$, and $v_k = e$. The route cost $W(\pi) = \sum W(v_i, v_{i+1})$ denotes the sum of each edge cost $W(v_i, v_{i+1})$ along the route. Thus, the shortest route cost $\delta(s, e)$ describes the minimal cost for routes available from s to e, this gives $\delta(s, e) = \min_\pi W(\pi)$.

Definition 1 (Travel Request). *A travel request tr across a road network $G = \{V, E, W\}$ takes the form of a quadruplet (s, e, t, τ), where $s \in V$ is the start, $e \in V$ the end, $t \in \mathbb{R}^*_+$ is the maximal waiting time and $\tau \in \mathbb{R}^*_+$ denotes a travel time for any extra diversion time in a travel. This bounds the overall distance from s to e to $(1 + \tau)\delta(s, e)$.*

For each travel request $tr_i = \{s_i, e_i, t, \tau\}$ and a given taxi, r_i denotes the taxi's location. A sequence of $3x$ points can describe a general *travel itinerary* for a taxi with x travel requests: $(p_1, p_2, \cdots, x_{3x})$ as an point p_j in the sequence denotes either a *start* (s_i), an *end* (e_i), or travel request point (r_i). Furthermore, this paper assumes that a taxi takes shortest route when moving between any two consecutive points in the travel itinerary p_i and p_{i+1}.

Thus, the *travel cost* between any two points (p_i, p_j) in the travel itinerary $\delta_T(p_i, p_j)$ becomes $\delta_T(p_i, p_j) = \delta(p_i, p_{i+1}) + \delta(p_{i+1,i+2}) + \cdots + \delta(p_{j-1}, p_j)$ and the overall travel cost $\delta_T(p_1, p_{3x})$. Figure 1 illustrates this for four travel requests.

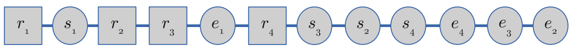


Fig. 1. A travel itinerary that contains the travel starting point s_i, a travel ending point e_i and taxi location r_i when request of travel tr_i arrives.

However, not all travel itineraries can meet the service quality guarantees for each individual travel request. Hence, we must define a *legal travel itinerary*:

Definition 2 (Legal Travel Itinerary). *A legal travel itinerary I for a travel request set $T_R = \{tr_1, tr_2, \ldots, tr_m\}$ must satisfy three conditions for any tr_i:*

1. **Order of events**: *Let $p_{i1} = r_i$, $p_{i2} = s_i$, and $p_{i3} = e_i$, then, $i_1 < i_2 < i_3$, i.e., the requesting point must happen before the pickup point, etc.*
2. **Waiting time**: *The time to travel from the taxi's location to the start can never exceed the waiting time constraint, i.e., $\delta_T(r_i, s_i) \leq t$*
3. **Travel time**: *The actual travel time from the start to the end $\delta_T(s_i, e_i)$ should be at most $(1 + \tau)\delta(s_i, e_i)$.*

To finally define the *Real-Time Context-Aware Taxi Dispatch* problem we need to provide one more definition to take into account multiple travel requests:

Definition 3 (Aggregated Legal Travel Itinerary). *Assuming at time t, there are x travel requests allocated to a given taxi; let $(p_1, p_2, \cdots, p_{3x})$ denote the current legal travel itinerary. For a new travel request tr_{m+1}, the aggregated legal travel itinerary contains any legal travel itineraries $(p'_1, p'_2, \cdots, p'_{3x+3})$ that satisfy $p'_j = p_j$ for $j \leq i$, and $p'_{i+1} = r_{x+1}$.*

Thus, we can define the **Real-Time Context-Aware Taxi Dispatch** as:

Definition 4 (Problem Definition). *Given a set of taxis, a set of previously-allocated requests and a new request $tr = (s, e, t, \tau)$, find the taxi that minimises the travel cost from s to e at that time, in its aggregated legal travel itinerary.*

Note that due to subsequent request allocation, a taxi may not minimise the travel cost of those already accepted, but it will *always* satisfy all constraints.

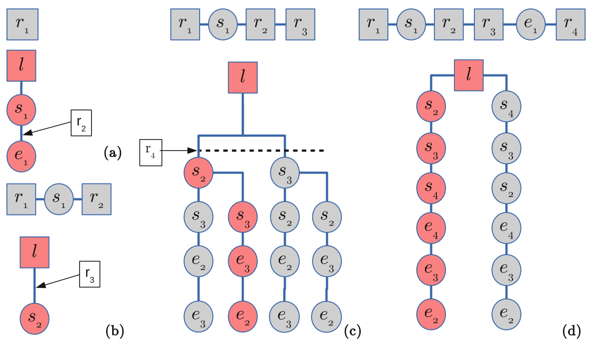


Fig. 2. Adaptive Context Trie (ACT) for travel itineraries. Darkened/red route indicates the selected itinerary to be executed. The dark circled/red travels denoted completed travel requests (Color figure online).

3.2 Trie Structure

ACT stores a travel itinerary for each taxi by associating a key to each itinerary point p. That key is used to store the next itinerary point. A virtual network structure orchestrates the storage of key-context value pairs by assigning keys to different points; the point will store the values for all the keys for which it is responsible. Thus, ACT specifies how keys are assigned to points, and how a point can access context for a given key by first locating the point responsible for that key. In short, this enables ACT to preform urban scale context dissemination method across all points, both for passengers and taxis [6]. Specifically a system uses ACT to maintain all legal travel requests with respect to the taxi's location. Eventually, as the taxi moves, a part of the itinerary becomes obsolete. Thus we need ℓ to track the current position of a taxi and root of its trie.

For a given t and τ, Fig. 2 illustrates the ACT structure corresponding to the complete travel itinerary and context stored across peers in Fig. 1. The darkened/red route represents the selected itinerary that the taxi will execute. Initially, for the first travel request, only one legal travel itinerary exists as shown in Fig. 2(a). When the second request arrives, the taxi has finished with the first passenger. If the taxi accepts the new request it can only perform one option, as it will first pick up the second passenger, but it has the flexibility to accept other travels if needed. This means, for now at least, that the taxi must take the route (ℓ, e_1, s_2, e_2), to drop off the first passenger and pick up the second.

However, on its way to pick up the second passenger, the third request arrives. The taxi now may either pick up the second passenger or the third one. Assuming that the taxi decides to move along the shortest route $(\ell, s_2, s_3, e_3, e_2)$. It then drives to pick up the second passenger and the fourth request arrives the entire right sub-trie of r_3 in Fig. 1(c) becomes inactive. Thus advantageously:

Corollary 1 (Legal Itineraries during Mobility). *When a taxi reaches a new pickup location or drop off location in the travel itinerary, then the taxi only follows legal itineraries which contain unfinished travels and share the same prefix in ACT. The taxi can safely ignore and exclude all the other itineraries.*

3.3 Processing a New Travel Request

When processing a new travel request containing (r_k, s_k, e_k) we assume that the taxi has already an ACT containing all legal itineraries of unfinished travels. Now, it needs to extend all legal itineraries in the prefix trie to a new legal itinerary to include it, if possible. To deal with the new request, it first focuses on the start s_k and then the end e_k. Essentially, a system needs to scan this trie to determine where s_k can be inserted.

All itineraries that share the same prefix from ℓ, the *root* of a trie with respect to the current location of the taxi, to the inserted edge will be added into the trie. Then, we append e_k after s_k in the new trie. Furthermore, if the system can append s_k or e_k at a given location forming an edge in this trie, then the system must find out which travel itineraries containing that edge with an additional point will be invalid and will be excluded. This introduces two problems: (a) How to determine at which edge the system can insert s_k or e_k, in addition to: (b) How to quickly delete any invalid travel itineraries that now exist.

Inserting Start Location: Here a system must know whether it should first insert s_k and then e_k afterwards. To insert s_k in a trie edge, e.g. (p_i, p_{i+1}), it must handle the following cases: (a) only when the distance from the current location to the pickup location s_i satisfies $\delta_T(l, s_i) = \delta(l, p_1) + \delta(p_1, p_2) + \cdots + \delta(p_i, s_k) \leq w$, then it can insert s_k; (b) the additional travel time introduced by the diversion to s_k may make some existing travel itinerary invalid in the sub-trie containing this trie edge (p_i, p_{i+1}), i.e. $\delta(p_i, s_k) + \delta(s_k, p_{i+1}) - \delta(p_i, p_{i+1})$ should not be too large. It should now exclude these from the sub-trie to ensure: $\delta_T(\ell, s_k) \leq w$. Thus, the shortest distance from the current location to the pickup location s_k has a value less than w given a itinerary from ℓ to p_j.

As ACT enables a search across peers starting from the root ℓ to generate all the candidate edges (p_i, p_{i+1}) to insert s_k it can handle condition (b): the explicit maintaining and checking of constraints for each travel request in the sub-trie of the point p_i provides a straightforward way to erase itineraries.

Furthermore, only a single criterion needs to be tested: if the taxi has not picked up the passenger, then ACT can test the pickup waiting time constraint $[r_j, s_j, w]$; once the taxi picks up the passenger, ACT can check the travel constraint $[s_j, e_j, \tau]$. Thus, at any given point, ACT can enable a system to simply partition the "active" passengers into two sets: A that records those passengers who need to be picked up and B that records the on-board passengers who need to be dropped off. When a new location is reached, it moves passengers from A to B and/or remove passengers from B. For travel j in A, the system tests the first criteria: $[r_j, s_j, w]$ and in B, tests the second one: $[s_j, e_j, \tau]$. Therefore, for the sub-trie rooted at p_i, a system can first generate these two sets A and B and then, when it inserts s_k, however it also needs to test each condition associated with the sets A and B.

Algorithm 1. Insert points (from a travel request) pseudo-code

```
Require: root (taxi location) $\ell$, request points $P = (p_1, x_2, ...)$, at current depth $depth$
  if $feasible(l, x_1, depth + \delta(\ell, x_1))$ then
    $fail = 0$, $n = create(\ell, x_1)$ {Copy feasible child branches beneath $n$}
    for all $c$ such that edge $(\ell, c)$ exists do
      $copy(n, \{c\}, \delta(\ell, n) + \delta(n, c) - \delta(\ell, c))$ {If copy fails, $fail = 1$}
    end for{Insert remaining request points to $n$}
    if $fail = 0$ and $|P| > 1$ then {Detour now begins negative as no $p_2$ yet}
      $insert(n, \{x_2, ...\}, -\delta(p_1, x_2))$ {If insert fails, $fail = 1$}
    end if{Now insert request points into children}
    for all $c$ such that edge $(\ell, c)$ exists do
      $insert(c, P, diversion + \delta(\ell, c))$ {If insert fails, exclude $(\ell, c)$}
    end for
    if $fail = 0$ then
      Add edge $(\ell, n)$
    else if No points $c$ with edge $(\ell, c)$ exist then
      Insert fails! {We have an unfeasible sub-trie}
    end if
  else
    Insert fails! {We have an unfeasible sub-trie}
  end if
```

Algorithm 1 implements this described recursive insertion of a new request $tr_k = (s_k, e_k)$ into ACT. This insertion occurs using a call, $insert(l, \{s_k, e_k\}, 0)$. The call to $feasible(parent, point, diversion)$ determines if it can feasibly insert a *point* as a child under a *parent* leaf in the discovered itinerary to always ensure an legal aggregate travel itinerary.

The $copy(to, from, diversion)$ function recursively copies points from a set of leaves in the trie, $from$, to the target, to. Here, tolerance of the root's (ℓ's) children in *insert* is implemented through calling $feasible$ with a $diversion$.

Figure 3 shows how to insert the pickup location s_4 into an existing trie. First s_4 will be inserted directly below ℓ. Then, the branch with root at s_3 will be copied underneath this new s_4 point, forming a trie of $(\ell, s_4, s_3, ((e_2, e_3), (e_3, e_2)))$. Assuming the unfeasible route $(\ell, s_4, s_3, e_3, e_2)$; then, a system should exclude the branch from this trie until we reach s_3, when it has an alternate feasible route $(\ell, s_3, s_4, e_3, e_2)$. This deletion occurs in the *copy* function, which will succeed because s_4 falls along at least one feasible route as shown in Fig. 3(b).

Then, the insertion algorithm moves down to s_3 and attempts to insert the pickup location. This forms two routes: $(\ell, s_3, s_4, e_2, e_3)$ and $(\ell, s_3, s_4, e_3, e_2)$, as a result of the insertion between s_3 and e_3 and between s_3 and e_2 as Fig. 3(c) shows. Suppose inserting s_4 between e_2 and e_3 or between e_3 and e_2 is unfeasible, then, this case is shown in Fig. 3(d). To complete the insert, a system now tries to insert e_4 in the sub-tries that start at s_4 following the insert operation.

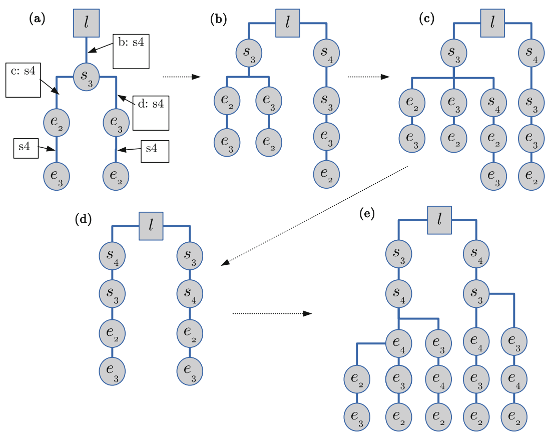


Fig. 3. Trie Insertion. The insertion of s_3 into each edge using ACT.

4 Cluster-Based Optimisation

Although the ACT approach is promising, the exponential explosion of the size of the trie when there are multiple start or end locations close to each other is not avoided. For example, if a taxi has 6 starting points in spatial proximity around similar time e.g. a large park or university campus, any permutation of the starts may result in a legal itinerary. So $6! = 720$ possibilities exist.

The following **clustering algorithm** deals with this situation. When the system inserts a starting point s_k to an edge (p_i, p_{i+1}), we check if $\delta(p_{p+i}, s_k) \leq \mu$, where μ denotes a small number. If so, ACT inserts s_k into the point of p_{i+1}. s_k and the system can treat p_{i+1} as one *cluster* in the trie and it can choose an arbitrary itinerary among the points in a cluster. When the cluster contains more than one point, the newly inserted point needs to be within μ of all the other points in the cluster. A similar procedure can be done for the end points and the mixture of starting and ending points. Once a system combines the cluster with any point, it will stop trying to insert it to any other edges using ACT (Fig. 4).

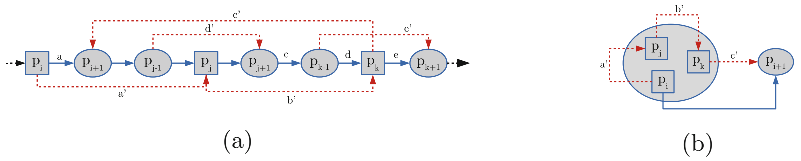


Fig. 4. p_i, p_j, and p_k in one cluster. Black lines: optimal itinerary S_o. We can convert S_o by connecting p_i, p_j, p_k consecutively first and then thread the other locations (represented by circles). The itinerary has a bounded cost.

Assuming sufficiently large travel times with all possible itineraries: For a travel request set T_R, let S_o define the optimal itinerary. Suppose there exists a cluster c among the start and end locations of T_R. This cluster-based method chooses an arbitrary itinerary S_c that goes through the points of the cluster in a consecutive manner. The following can then crucially prove that the bounded cost of S_c, which clearly indicates a feasible approach:

Theorem 1. *$cost(S_c) \leq cost(S_o) + 2(x+1) \times \mu$ where x denotes the number of points in the cluster without considering constraints.*

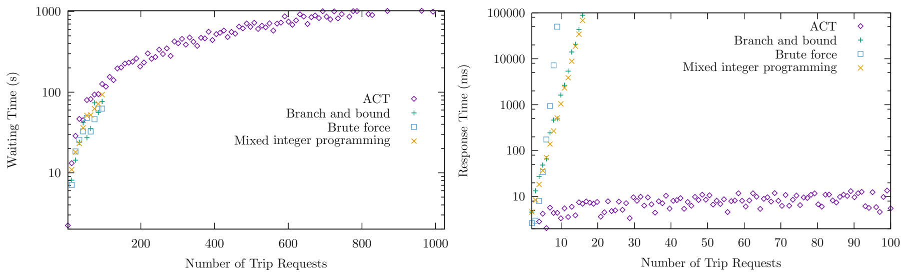
Because after ACT builds the whole trie, a system can select the shortest itinerary with cluster $cost(S_c) \leq cost(S_o) + 2(x+1) \times \mu$. However, when the constraints of points of the best itinerary are relaxed, the corresponding cluster-based itinerary can also depend on the following theorem:

Theorem 2. *$cost(S_c) \leq cost(S_o) + 2(x+1) \times \mu$ where x defines the number of points in the cluster when constraints of all points in $S_o > x\mu$.*

5 Simulation Results

To compare our proposed method with its points we have made extensive simulation experiments. The implementation of the ACT approach [6] was used directly as a Java library, while for the traditional optimisation algorithms and mixed integer programming methods we used MATLAB's (R2014b build) Optimisation Toolbox. The experiments were conducted on an Intel i7-2600 SMP-based GNU/Linux computer using the v4.1 kernel and OpenJDK v1.6.

Figure 5a compares the waiting time for a taxi as exhibited by the ACT approach with branch and bound, brute-force and mixed integer optimisation algorithms as the number of travel requests increase with a fixed number of 2^{10} taxis. Notably although traditional algorithms cannot continue processing as the problem sizes become too large, ACT can scale to higher capacities in terms of response time. This also confirms our hypothesis that situations where a large number of passengers wish to depart from a single point infer the biggest issue for capacity. ACT through clustering combines such points in a trie to provide an urban-scalable approach. Figure 5b shows the time complexity of each algorithm to further highlight these observations. This also demonstrates the workability of non-optimal taxi dispatch against traditional optimisation algorithms.



(a) Waiting time for a taxi (b) Time complexity for each algorithm

Fig. 5. A comparison of ACT against existing approaches (see related work)

6 Conclusion and Future Work

This paper formulates and proposes an ACT-based approach with a cluster-based optimisation to match real-time travel requests to taxis in a road network to realise efficient context-aware taxi dispatch while ensuring all conditions of a given travel request are met. Our proposed solution might not find the best solution but it provides an acceptable solution efficiently. It also clearly outperforms current state of the art optimisation-based solutions, as shown by large scale experiments. Future work will consider the uncertainty issues in scheduling [16] as this will likely form a major road block in achieving scalable context-aware transport systems. It will also include a more complete analysis of the cluster-based optimisation and provide the full proofs to the relevant theorems to further outline the soundness of the cluster-based optimisation and investigate how the inevitable privacy issues could be tackled in such frameworks.

References

1. Smirnov, A., Levashova, T., Shilov, N., Kashevnik, A.: Context-aware decision support in dynamic environments: theoretical and technological foundations. In: Obaidat, M.S., Koziel, S., Kacprzyk, J., Leifsson, L., Ören, T. (eds.) Simulation and Modeling Methodologies, Technologies and Applications. Advances in Intelligent Systems and Computing, vol. 319, pp. 3–20. Springer, Heidelberg (2015)
2. Hwang, R.H., Hsueh, Y.L., Chen, Y.T.: An effective taxi recommender system based on a spatio-temporal factor analysis model. Inf. Sci. **314**, 28–40 (2015)
3. Zhang, D., He, T., Liu, Y., Lin, S., Stankovic, J., et al.: A carpooling recommendation system for taxicab services. IEEE Trans. Emerg. Top. Comput. **2**, 254–266 (2014)
4. He, W., Hwang, K., Li, D.: Intelligent carpool routing for urban ridesharing by mining gps trajectories. IEEE Trans. Intell. Transp. Syst. **15**, 2286–2296 (2014)
5. Dong, H., Zhang, X., Dong, Y., Chen, C., Rao, F.: Recommend a profitable cruising route for taxi drivers. In: 2014 IEEE 17th International Conference on Intelligent Transportation Systems (ITSC), pp. 2003–2008. IEEE (2014)
6. Morris, A., Patsakis, C., Dragone, M., Manzoor, A., Cahill, V., Bouroche, M.: Urban scale context dissemination in the internet of things: challenge accepted. In: 9th International Conference on Next Generation Mobile Applications, Services and Technologies. IEEE (2015)
7. Morris, A., Bouroche, M., Cahill, V.: Urban scale dissemination in mobile pervasive computing environments. PECCS **2014**, 18 (2014)
8. del Carmen Rodríguez-Hernández, M., Ilarri, S.: Towards a context-aware mobile recommendation architecture. In: Awan, I., Younas, M., Franch, X., Quer, C. (eds.) MobiWIS 2014. LNCS, vol. 8640, pp. 56–70. Springer, Heidelberg (2014)
9. Schmidt, M., Schöbel, A.: Timetabling with passenger routing. OR Spectrum **37**, 75–97 (2015)
10. Gacias, B., Meunier, F.: Design and operation for an electric taxi fleet. OR Spectrum **37**, 171–194 (2015)
11. Qu, M., Zhu, H., Liu, J., Liu, G., Xiong, H.: A cost-effective recommender system for taxi drivers. In: Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp. 45–54. ACM (2014)
12. Hu, T.Y., Chang, C.P.: A revised branch-and-price algorithm for dial-a-ride problems with the consideration of time-dependent travel cost. J. Adv. Transp. (2014)
13. Nisse, N., Mazauric, D., Coudert, D.: Experimental evaluation of a branch and bound algorithm for computing pathwidth. In: Gudmundsson, J., Katajainen, J. (eds.) SEA 2014. LNCS, vol. 8504, pp. 46–58. Springer, Heidelberg (2014)
14. Rais, A., Alvelos, F., Carvalho, M.S.: New mixed integer-programming model for the pickup-and-delivery problem with transshipment. Eur. J. Oper. Res. **235**, 530–539 (2014)
15. Bonami, P., Kilinç, M., Linderoth, J.: Algorithms and software for convex mixed integer nonlinear programs. In: Lee, J., Leyffer, S. (eds.) Mixed Integer Nonlinear Programming. The IMA Volumes in Mathematics and its Applications, vol. 154, pp. 1–39. Springer, Heidelberg (2012)
16. Cats, O., Gkioulou, Z.: Modeling the impacts of public transport reliability and travel information on passengers waiting-time uncertainty. EURO J. Transp. Logistics, 1–24 (2015)