Fast, Exact and Scalable Dynamic Ridesharing

Summary

Paper authors: Valentin Buchhold, Peter Sanders, Dorothea Wagner
Summary: Cristain Ion

Introduction

The paper proposes an improvement to the ridesharing problem. The problem is important to for the practical application for large fleets of autonomous vehicle, which might be crucial for the future. Finding a solution for the entire set of vehicles is NP-complete, so most practical implementations process the requests one by one and aim to minimize the increase in the driving time of any vehicle.

The propsed work covers searching in graphs using Dijkstra, Contraction Hierarchies (CH), CustomizedCH, BucketCH and the algorithm which makes use graph searching optimized for the ridesharing problem and constraints.

Their algorithm LOUD is based on contraction hierarchies (CH) with local buckets. The algorithm maintains the forward and reverse CH search spaces, aggressively pruning of the buckets so that only those entries that can possibly contribute to feasible insertions.

Related work

- Dynamic ridesharing is related to the Dial-a-ride problem (DARP) problem in operations research. The DARP primarly considers small instances and solutions where all ride requests are known in advance.
- Travelling salesman problem with time windows
- MATSim
- T-Share algorithm and grid paritions using shortest-path distances using hub labeling and caching
- Dynamic carpooling - drivers can specify fixed source and target and can pick up and drop off passengers thus sharing the cost of the ride. All constrains also apply to drivers

Problem statement

Road network

$G = (V, E)$, directed graph where vertices represent intersections and edges represent road segments with an associated travel time

- $l(v, w)$ - travel time between two adjacent intersections, ≥ 0
- $d(v, w)$ - shortest path between two intersections

Set of vehicles

- Each vehicle $v = (l_i, c, t_{serv}^{min}, t_{serv}^{max})$ has an initial location, a seating capacity, and a service interval
- Vehicle location $l_c(v)$

- Vehicle route: $R(v) =< s_0, \ldots, s_k >$ is a sequence of stops at locations $l(s) \in V$.
    - The route is updated each time the vehicles makes a stop, making s_0 the current stop
    - When driving, the vehicle location is between s_0 and s_1

Ride requests

Ride request $r = (p, d, t_{dep}^{min})$ are received by a dispatching server and are immediately matched to vehicles

- pick-up spot $p \in V$
- drop-off spot $d \in V$
- earliest departure time t_{dep}^{min}

Inserting requests

Insertion is represented by (v, r, i, j). The vehicle v, picks up request r, immediately after stop $s_i(v)$ and drops off request r immediately after stop $s_j(v)$. The model constraints and parameters are:

- Wait time is a hard constraint for each request already matched to a vehicle, where t_{wait}^{max} is a model parameter
    - $t_{dep}^{max}(r') = t_{dep}^{max}(r') + t_{wait}^{max}$
- Trip time is a hard constraint for each request already matched to a vehicle, where α and β are model parameters.
    - $t_{arr}^{max}(r') = t_{dep}^{min}(r') + t_{trip}^{max}(r')$
    - $t_{arr}^{max}(r') = t_{dep}^{min}(r') + \alpha\, dist(p(r'), d(r')) + \beta$
- Each stop takes t_{stop} time, given as a parameter to the model
- The wait time and trip time constraints are added to the objective value of $f(\iota)$ of an insertion ι. γ_{wait} and γ_{trip} are model parameters.
    - $f(\iota) = \delta + \gamma_{wait}\, max\{t_{dep}(p(r)) - t_{dep}^{max}(r), 0\} +$
    - $+ \gamma_{trip}\, max\{t_{arr}(d(r)) - t_{arr}^{max}(r), 0\}$

Solution

Maintaining Feasiblity

- $t_{dep}^{min}(s)$ and $t_{arr}^{min}(s)$ are maintained for each stop on each vehicle route

Inserting a new request $r' = (p', d', t_{dep}^{min'})$ in a route
$< s_0', \ldots, s_{i'}' = p', \ldots, s_{j'}' = d', \ldots, s_{k'}' >$

- $t_{dep}^{min}(s_l') = t_{dep}^{min}(s_{l-1}') + dist(s_{l-1}', s_l') + t_{stop}$, for all s_l', $i' \leq l \leq k'$
- $t_{arr}^{max}(s_l') = min\{t_{arr}^{max}(s_l'), t_{arr}^{max}(s_{l+1}') - dist(s'_l, s'_{l+1}) - t_{stop}\}$

- By maintaining these values, it is possible to check all service, wait and trip time constraints on a route in **constant** time
- Time constraints are satisfied if and only if:
  - $t_{dep}^{min}(s_{i+1}) - t_{stop} + \delta_p \leq t_{arr}^{max}(s_{i+1})$
  - $t_{dep}^{min}(s_{j+1}) - t_{stop} + \delta_p + \delta_d \leq t_{arr}^{max}(s_{j+1})$
  - $t_{dep}^{min}(s_k) + \delta_p + \delta_d \leq t_{serv}^{max}(v)$
- Capacity constraint given by the number of occupied seats for each stop $s \in R$, each vehicle route
  - $o(s'_{i'}) = o(s'_{i'-1})$
  - $o(s'_{j'}) = o(s'_{j'-1})$
  - increment $o(s'_l)$ for all $i' \leq l \leq j'$

Elliptic Pruning

BCH is used to obtained shortest-path distances in order to compute insertion costs. Having two consecutive stops *(s, s')* on a vehicle's route, the new pickup or dropoff spot *v* will be inserted only if $dist(s, v) + dist(v, s') \leq \lambda(s, s')$. The authors use $\lambda(s, s')$ notation for the leeway between two stops. Further, in the paper is presented a theorem which describes which bucket entries are sufficient for the reverse BCH search from *v.* It maintains a highest rank ordering of the vertices on all shortest paths.

Given a ride request $r = (p, d, t_{dep}^{min})$, the algorithm inserts it into any vehicle's routes such that the vehicle's detour plus the violations of the soft constraints is minimized. A request is solved in four phases.

Stage 1. Computing shortest-path distances

- Compute shortest-path distance from the pickup p to the dropoff d with standard CH query
- Compute latest time $t_{dep}^{max}(r)$ for picking request r
- Compute latest time $t_{arr}^{max}(r)$ for dropping request r
- Ordinary insertions costs by computing all shortest-path distances needed to calculate the costs (v, r, i, j) with 0 < i <= j < |R(v)| - 1. This is done by running

two forward BCH searches (from p and d) that scan target buckets and two reverse BCH searches (from p and d) that scan the source buckets.

Stage 2. Trying ordinary insertions

Each bucket stores the set *C* of vehicles, vehicles not contained in *C* allow no feasible ordinary insertion.

- Checking capacity constraints
- Checking remaining hard constraints in constant time

Stage 3. Trying special case insertions

Aside from ordinary insertions, we must find the cost of insertions which depend on shortest-paths not computed by searches with BCH.
Trying all insertions $(v, r, 0, j)$ with $0 \leq j < |R(v)| - 1$ for all vehicles from *C*. Such insertions insert the next pick-up before the next scheduled stop on a vehicle's route.

Stage 4. Updating preprocessed data

Finding a feasible insertions leads to updating the preprocessed data, so the algorithm is ready resolve the next ride request.

The best found insertion is placed into the current route of the vehicle. The arrival and departure times and capacity values are updated in time linear in the length of the route.

Experiments

LOUD is evaluated on the state-of-the-art Open Berlin Scenario, including comparison to related work. The Open berling scenario has been published in two versions. The Berlin-1pct simulates 1% of all adults living in Berlin and Brandenburg. The Berlin-10pct simulates 10%.

Table of Benchmark instances

input	|V|	|E|	veh	req
Berlin-1pct	73689	159039	1000	16569
Berlin-10pct	73689	159039	10000	149185

Inputs

Benchmark instances from Open Berlin Scenario implemented in MATSim, a transport simulation. The simulation MATSim works in iterations. The population is updated at each iteration by movement, departure time, route, mode, and destination choice and outputting each person 24-hour travel pattern.

For experiments it is taken the 500th iteration for the Berlin-1pct as it is recommended for realistic travel patterns.

For Berlin-10pct experiments, it is taken the 250th iteration.

The simulation maintains each vehicle's current state (out of service, idling, driving, or stopping) and a priority queue for pending events. The events are scheduled in advanced and may generate new events in the future. The simulation stops after emptying the event queue.

Parameters

- $t_{stop} = 1min, t_{wait}^{max} = 5min$
- $\alpha = 1.7min, \beta = 2min$
- $\gamma_{wait} = 1, \gamma_{trip} = 10$

Results

LOUD is compared with the MATSim heuristic and exact variants. It is approximately 30 times faster than the MATSim simulation on Berlin-10pct.

			request statistics [m:s]				vehicle statistics [h:m]			
		time	wait		ride	trip	empty	occ	stop	op
instance	algorithm	[ms]	avg	95 %ile						
Berlin 1pct	MATSim-h	13.83	4:11	8:21	14:11	18:22	0:35	3:19	0:33	4:27
	MATSim-e	19.29	4:12	8:20	14:11	18:23	0:36	3:19	0:33	4:28
	LOUD-CH	0.71	4:12	8:20	14:11	18:23	0:36	3:19	0:33	4:28
	LOUD-CCH	0.68	4:12	8:20	14:11	18:23	0:36	3:19	0:33	4:28
Berlin 10pct	MATSim-h	18.33	3:44	8:21	14:52	18:37	0:14	2:31	0:29	3:14
	MATSim-e	23.42	3:47	8:13	14:51	18:37	0:13	2:31	0:29	3:13
	LOUD-CH	0.63	3:47	8:13	14:51	18:37	0:13	2:31	0:29	3:13
	LOUD-CCH	0.69	3:47	8:13	14:51	18:37	0:13	2:31	0:29	3:13

Conclusion

The authors proposed the LOUD algorithm for large scale dynamic ridesharing, which uses a modern approach with many-to-many problem (BCH) considerations. It has a great performance < 1 ms, 30 times faster than current algorithms on Berlin-10pts. For future work, the authors want to test LOUD on larger instances.