Incremental Management of Oversubscribed Vehicle Schedules in Dynamic Dial-A-Ride Problems

Zachary B. Rubinstein, Stephen F. Smith, and **Laura Barbulescu**
The Robotics Institute
Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, PA 15213
{zbr,sfs,laurabar}@cs.cmu.edu

Abstract

In this paper, we consider the problem of feasibly integrating new pick-up and delivery requests into existing vehicle itineraries in a dynamic, dial-a-ride problem (DARP) setting. Generalizing from previous work in oversubscribed task scheduling, we define a controlled iterative repair search procedure for finding an alternative set of vehicle itineraries in which the overall solution has been feasibly extended to include newly received requests. We first evaluate the performance of this technique on a set of DARP feasibility benchmark problems from the literature. We then consider its use on a real-world DARP problem, where it is necessary to accommodate all requests and constraints must be relaxed when a request cannot be feasibly integrated. For this latter analysis, we introduce a constraint relaxation post processing step and consider the performance impact of using our controlled iterative search over the current greedy search approach.

Introduction

Many practical planning and scheduling problems are dynamic, where new requests arrive over time and must be continually integrated into currently executing plans and schedules. In such environments, it is generally undesirable (and usually computationally infeasible) to regenerate a new solution to accommodate each new request, due to a combination of problem scale, the pace at which new requests arrive and execution events unfold, and the need to maintain stability in executing processes. This is the case in the dynamic pick-up and delivery scheduling problem that motivates our research. In this domain, advance reservations permit the construction of day-ahead schedules, but both same-day requests and other execution events (traffic congestion, trip cancelations, vehicle breakdowns, etc.) require dispatchers to constantly integrate and re-integrate requests into the current schedule.

Iterative, repair-based search techniques offer one approach to incrementally revising and improving plans and schedules as changes to requirements are introduced (e.g., (Zweben et al. 1994; Chien et al. 2000; Cesta, Oddi, and Smith 2000). They have also been effectively applied to a closely related problem, that of maximizing the number of tasks that can be performed in oversubscribed problem settings (i.e., where resource demand exceeds capacity). One particular iterative repair search procedure, *TaskSwap* (Kramer and Smith 2003; 2004), is particularly relevant to the problem of incrementally accommodating new requests over time. TaskSwap, which operates similarly to ejection-chain algorithms (Lim and Zhang 2007; Nagata and Brysy 2009), conducts a controlled search around the "footprint" of a currently unscheduled task (request), temporally retracting conflicting tasks to allow placement of the new task, and then attempting to feasibly reinsert all retracted tasks elsewhere in the schedule. If all requested tasks make it back in, then the schedule is feasibly extended. TaskSwap has been successfully applied to large-scale mission scheduling problems (Kramer, Barbulescu, and Smith 2007).

One assumption made by TaskSwap is that tasks are atomic and self-contained. This implies that resources (or resource capacity) can be allocated fully to a given task for its duration and then freed up for use independently by other tasks. Unfortunately, this assumption does not hold for pick-up and delivery problems, where a given request decomposes into a pick-up, drop-off task sequence with intervening travel, and depending on vehicle capacity constraints, it is possible (and often advantageous) to interleave execution of the pick-ups and drop-offs of several requests.

In this paper, we extend the original TaskSwap procedure for use in problem domains with non-atomic requests whose constituent tasks can be interleaved if resource capacity constraints allow. We focus specifically on dynamic Dial-a-Ride (DARP) problems, and we evaluate our new procedure, which we call Generalized Task Swap (GTS), in this context. First we consider performance on a set of synthetic benchmark problems that have been previously studied in the literature, and show the ability of GTS to solve difficult feasibility problems. We then consider its use on a real-world paratransit scheduling problem, where it is necessary to accommodate all requests and constraints must be relaxed when a request cannot be feasibly integrated. For this latter analysis, we introduce a constraint relaxation post-processing step to GTS, and show that GTS is able to significantly reduce the total delay necessary to accommodate all requests over the current greedy search approach.

Dial-a-Ride Problem (DARP)

Dial-a-Ride Problems have been extensively studied in the literature; for an in-depth review of DARPs see (Cordeau and Laporte 2007). The objective in a DARP is to design vehicle itineraries to satisfy requests for travel between pick-up and drop-off locations at specified times. A request consists of pick-up and drop-off locations, time windows within which the pick-up and the drop-off must be completed, and the number and types of passengers, e.g., one ambulatory and one wheelchair passenger. The time window for a drop-off may be specified in absolute times or relative to when the pick-up occurs. For example, a common way to specify the latest time a drop-off can occur is to provide a maximum time that a passenger can ride in a vehicle. DARPs require models both for vehicle types that describe their capabilities, including the number and types of passengers they can carry and, possibly, their speeds, and for computing the travel duration between two locations. As input, DARPs are given a set of requests and a specific set of vehicles. A solution for a DARP is a set of vehicle itineraries, i.e., timelines for each vehicle consisting of a series of pick-up and drop-off tasks corresponding to the given requests and the requisite travel between those tasks. The resulting itineraries form the overall schedule.

DARPs may be oversubscribed, i.e., not all requests can be serviced within their constraints, in which case the constraints may be relaxed in order to help accommodate the extra requests. There often are optimizing objectives for DARPs, such as minimizing cost by minimizing the number of vehicles used while allowing some level of constraint relaxation, or maximizing service quality by determining the minimal number of vehicles to satisfy the requests without relaxing constraints. Formally, there are two classes of DARPs, static and dynamic. In a static DARP, all the requests and the available vehicle are known up front. In a dynamic DARP, the requests are serviced as they arrive and the available vehicles can be increased. Most real-world DARPs are hybrids, with a majority of the requests but not all being known up front and there mostly being a static number of available vehicles with extra ones available at a cost. In this paper, we focus on hybrid DARPs and the problem of incrementally adding new requests to an existing schedule.

Generalized Task Swap (GTS)

Generalized Task Swap (GTS) is an iterative improvement algorithm based on TaskSwap that attempts to insert new requests into an existing schedule. The basic intuition behind both TaskSwap and GTS is that, by reallocating scheduled requests whose tasks overlap in time with those in the new request, sufficient resource capacity can be freed up to accommodate the new request. GTS differs from TaskSwap in that it operates on more complex resource capacity requests.

Fundamental to both TaskSwap and GTS is the determination of *conflict sets* - alternative sets of requests that could be retracted to make room on the timeline for the new request. The original TaskSwap procedure assumes that each request consists of a single task whose duration depends solely on the resource assigned. Under this assumption, where there are no dependencies with any other tasks assigned to the same resource, identification of conflict sets is straightforward. A new request may be assigned to a given resource if there is a contiguous block of time on the resource within the temporal bounds of the request with sufficient capacity to service it. The conflict sets then are those sets of currently scheduled requests that overlap with the new request and collectively consume enough capacity to prevent the new request from being supported. To find these conflict sets for a given resource, it is sufficient to simply scan the capacity profile of the resource for periods where available capacity is less than required and then extract possible subsets of tasks scheduled within these periods that could provide this capacity.

GTS relaxes the assumption that a request must be a single, self-contained task. Instead, it assumes that requests can have more complex structure, consisting of a sequence of located tasks (e.g., pick-ups and drop-offs) whose execution may require intervening travel. In addition to typical window and duration constraints associated with constituent tasks, there can also be temporal dependencies (e.g., maximum temporal separation constraints) between successive tasks in a given request. It may also be possible to interleave the execution of constituent tasks with those of other requests (e.g., capitalizing on shared ride time) if resource capacity constraints allow.

The constraints and interdependencies associated with satisfying such requests add significant complexity to the determination of conflict sets. Indeed, two major issues make it difficult to decide which requests to retract such that a new request's tasks can be inserted. First, to figure out the overlapping scheduled tasks, one needs to consider the time dependencies between each of the new request's tasks when scheduled and the previous/next tasks on the timeline (since this determines, for example, how early the task can start depending on what is scheduled before it). Second, as overlapping tasks get retracted, timebounds of and travel durations between some of the remaining scheduled tasks change; short of actually retracting the tasks, one would have to compute an approximation of the effect of the retraction on the remaining scheduled tasks (and therefore on the available capacity). Instead, GTS systematically retracts subsets of the requests that overlap with the new request and collects those that allow for the new request to be inserted. To enforce all the complex temporal constraints, GTS takes advantage of an underlying simple temporal network (Dechter, Meiri, and Pearl 1991), which maintains consistency among constraints by detecting violations in the network as new constraints are added.

In order to determine if a new request can be inserted on a timeline, GTS must have, as a prerequisite, a search procedure for inserting a request, i.e., the tasks corresponding to the request, onto a resource timeline such that all constraints are satisfied. GTS makes use of an underlying insertion procedure that generates all feasible slots, i.e., times between tasks on the timelines in which a new task can be inserted while satisfying all constraints, and picks the best ones according to an insertion heuristic (an example is provided below).

Algorithm 1 GTS(request-to-insert, resource-timelines, max-moves)

```
 1: protected ← (request-to-insert)
 2: retracted ← (request-to-insert)
 3: num-moves ← 0
 4: while (new-request ← pop(retracted)) ∧ (num-moves < max-moves) do
 5:   if ¬insert(new-request,resource-timelines) then
 6:     conflict-sets ← ()
 7:     for timeline in resource-timelines do
 8:       conflict-sets ← conflict-sets ∪ compute-conflict-sets(new-request,timeline,protected)
 9:     if conflict-sets then
10:       conflict-set ← select-best(conflict-sets)
11:       timeline ← timeline(first(conflict-set))
12:       for request in pop(conflict-set) do
13:         retract(request)
14:         push(request,protected)
15:         push(request,retracted)
16:       insert(new-request,timeline)
17:       num-moves ← num-moves + 1
18:     else
19:       undo()
20:       return false
21: return true
```

Algorithm 1 shows the flow of GTS. At the top level, it is an iteration that attempts to insert new requests. The iteration terminates if there are no more requests to insert, if a request cannot be inserted, or if the number of requests moved has exceeded a threshold. GTS is successful in inserting the new request if the first case is true, otherwise it fails. The iteration begins with the retracted requests being initialized with the new request. Within the iteration, the insertion procedure is applied to the first retracted request to see if it can be inserted. If it can, then that iteration ends and the next retracted request is processed. If it cannot be inserted, then the conflict sets are computed in lines 7 and 8 and, in line 10, the best conflict set is selected using a retraction heuristic (an example is provided below). In line 11, the timeline that contains the tasks of the requests in the best conflict set are saved for inserting the new request once the conflicting ones are retracted. Then, in lines 12 through 15, the requests in the set are retracted and added to the retracted list and the protected list, which is used to prevent a request from being relocated more than once per invocation of GTS (how the protected list is used is described below). In lines 16 and 17, the new request is inserted on the saved timeline from which the requests were retracted, and the counter for the number of moved requests is incremented by one. The next iteration then starts with the next retracted request to process. If GTS fails to insert the new request, then all the relocated requests are undone in line 19, returning them to their original positions on their original timelines.

Algorithm 2 shows the details of how the conflict sets are computed. In lines 3 and 4, the set of requests overlapping in time with the new request is computed by iterating through

Algorithm 2 compute-conflict-sets(new-request, timeline, protected)

```
 1: conflict-sets ← ()
 2: overlap-requests ← ()
 3: for tasks in new-request do
 4:   overlap-requests ← overlap-requests ∪ compute-overlap-requests(task,timeline)
 5: overlap-requests ← overlap-requests \ protected
 6: for i ← 1 to |overlap-requests| do
 7:   conflict-set ← ()
 8:   for j ← i to |overlap-requests| do
 9:     push(overlap-requests[j],conflict-set)
10:     retract(overlap-requests[j])
11:     if insert(new-request,timeline) then
12:       push(conflict-set,conflict-sets)
13:       break {return from the innermost loop}
14:   undo()
15: return conflict-sets
```

the constituent tasks of the new request and, for each one, uniquely collecting the requests on the timeline that have tasks that overlap with it. Tasks overlap if their time windows, i.e., the intervals from the earliest start time to the latest finish time for each task, intersect. In the case of the new request, the time window is determined by the temporal constraints that came with the request. To prevent the protected requests from being relocated, they are removed from the overlapping requests. In lines 6 through 13, the conflict sets are generated by enumerating subsets of the overlapping requests and testing them by retracting them and seeing if the new request can be inserted. If it can, the set is collected. The subsets of overlapping requests are generated by, first, testing all overlapping requests, then all but the first, followed by all but the first and second, and so on until there are no more overlapping requests. Note that, to keep this enumeration computationally efficient, it is not exhaustive. After each set is tested, all retractions and insertions are undone, i.e., all requests are returned to their original state before the test. After all sets are tried, the collected conflict sets are returned.

GTS improves on TaskSwap by providing a more general method for computing conflict sets. TaskSwap takes advantage of a specific problem structure to efficiently generate conflict sets, but its method cannot be applied to more general problem structures where plan solutions for requests may have arbitrary structures consisting of multiple, possibly interleaving tasks and time dependencies between successive tasks. GTS's method of mapping over all of a request's tasks and actually retracting and inserting to test if a set of overlapping requests are in the conflict sets can be applied to these more complex domains, extending the benefits of task swapping to a broader class of problems.

Insertion and Retraction Heuristics

As indicated above, GTS relies on core procedures for inserting and retracting requests that requires choices over sets of alternatives (feasible slots in the insertion case, conflict

sets in the retraction case). To make these choices efficiently we consider alternative insertion and retraction heuristics.

For DARPs, the insertion heuristic provides estimates for the value of pairs of slots if the new request's pick-up and drop-off tasks were inserted into them. The two insertion heuristics we designed and implemented for this domain are the following:

- *Minimum Additional Travel Duration* - the amount of time that had to be added to the vehicle timeline to insert the pick-up and drop-off tasks is calculated for each pair of slots and the one with the lowest value is selected. The intuition behind this heuristics is that minimizing the travel time on the timeline should result in more slack being available for future requests.
- *Minimum Total Tardiness* - The cumulative tardiness resulting when inserting a request's pick-up and drop-off tasks into a pair of slots is calculated for each pair of slots and the one with the minimum tardiness is selected. Tardiness for both pick-up and drop-off tasks is computed by taking the difference between its earliest scheduled start time and its earliest possible start time. For drop-off tasks that do not have explicit time windows, the earliest possible start time is the earliest possible finish time of the pick-up plus the direct transit time from the pick-up location to the drop-off location. If two pairs have the same tardiness, then minimum additional travel duration is used to determine the best pair. The intuition behind this heuristic is that it will result in better service quality for the requests at the cost of efficiently using the timeline. However, as will be demonstrated in the evaluation section, it appears to result both in good service quality and in efficient use of the timelines. We believe that this performance is due to this heuristic tending to insert a request's tasks early in their possible windows, allowing them more flexibility to slide later as new requests are inserted.

The retraction heuristic we developed for using GTS on DARPs, is *Minimum Conflicts*, which selects the set of requests that have the fewest members. In the case of ties, the set that results in the minimum additional travel duration when the new trip is inserted is selected. The motivation behind this heuristic is that, by minimizing the number of retracted requests, there will be fewer of them and, consequently, it will tend to be easier to reassign them.

We evaluated these heuristics and the performance of GTS on DARPs, both on a set of benchmark problems and on a set of real-world problems. To produce an initial schedule for these experiments, we implemented a greedy basic scheduler (GBS) that iteratively invokes the same insertion procedure as GTS on a set of unscheduled requests. The experiments and their outcomes are described in the following section.

Basic Evaluation

To both validate the design and calibrate the performance of our new GTS algorithm, we analyze the results of running it on a benchmark of challenging feasibility DARP instances published by (Cordeau 2006) (problems available at http://www.hec.ca/chairedistributique/data/darp), that have been fairly extensively studied in the literature and were proved feasible/infeasible by sophisticated, extended search techniques (Berbeglia, Pesant, and Rousseau 2011; Jain and Van Hentenryck 2011). Because these challenging benchmark instances contain infeasible problems, they are also a good match with the oversubscribed character of our paratransit target application. The purpose of our empirical study using the benchmark instances is twofold. First, we use the GTS performance results for the benchmark instances to identify the best parameter setting for GTS. Second, we assess the contribution of running with GTS in terms of both finding feasible solutions for the feasible instances and decreasing the number of unscheduled requests in the solutions for the other instances.

The problems in the benchmark are randomly generated, where the coordinates of the pick-up and drop-off nodes are chosen in the $[-10, 10]$x$[-10, 10]$ square according to a uniform distribution and the depot is located at $(0, 0)$, the time horizon is up to 12 hours, and the time windows are 15 minutes. There are two sets of problems in the benchmark; the vehicle capacity is 3 for the instances in set *a* and 6 for the instances in set *b*. The maximum ride time (max-ride-time) is 30 minutes for set *a* and 45 minutes for set *b*.

Following the approach of (Berbeglia, Pesant, and Rousseau 2011), we run our algorithms on the original sets *a* (14 problems) and *b* (13 problems), as well as 3 modified versions of these instance sets. All the instances have at least 40 requests each (and at most 96 requests). For the first modified version, the max-ride-time is decreased to 30 minutes (note that this only affects the problems in set *b*); we denote this set as the RT=30 set. For the second version of the two problem sets, the max-ride-time is decreased to 22, denoted RT=22. Finally, for the third version, the number of available vehicles is reduced to 75% of its original size, denoted 75% vehicles.

We designed a factorial experiment where the independent variables were the ordering of the requests and the insertion heuristic. We ran GBS by itself and invoked GTS when there were unscheduled requests after running GBS; we recorded the number of unscheduled requests after GBS and then after running GTS. The values used for the ordering of the requests were: unsorted, earliest-start-time first (EST-first) and most-constrained first (most-constr-first). We defined most-constr-first based on the minimum duration of the request (the duration of the transit from its pick-up location to the drop-off location), where longer min-duration requests are more constrained. We considered the two insertion heuristics, *Minimum Additional Travel Duration* (min-added-duration) and *Minimum Total Tardiness* (min-tardiness). The dependent variables in our experiment were the number of instances for which a feasible solution has not been found and the number of unscheduled requests per instance.

The best overall results (minimizing the number of instances with unscheduled requests) are produced by the GTS runs with min-tardiness and EST-first. GTS significantly improves the solutions produced by the GBS; Table 1 shows the average percentage of the unscheduled requests after running GBS that get scheduled by GTS (columns 4 and 8).

Note that even though GTS assigns the greatest percentage of unscheduled requests for the unsorted ordering (with both insertion heuristics), the solutions produced by GBS with the unsorted order have more unscheduled requests than for any other ordering, meaning that GTS starts with worse solutions (columns 2 and 6). All the GTS runs found feasible solutions for all the instances in the Original and RT=30 sets; all of the unscheduled requests after the GTS runs (columns 3 and 7) were found for the RT=22 and 75%vehicles set. For the RT=22 set, all the GTS configurations found nine infeasible instances, with 13 unscheduled requests. Although the unsorted order with min-tardiness found solutions with fewer unscheduled requests for two of the instances in the 75% set, the unsorted order failed to find a feasible solution for a4-48; therefore in terms of instance feasibility, EST-first does a better job.

When comparing the number of feasible/infeasible instances produced by the best GTS parameter setting (with EST-first and min-tardiness) with the results published for Tabu Search and CP in (Berbeglia, Pesant, and Rousseau 2011), GTS matches the published results for the Original, RT=30 and RT=22 sets. We report in parentheses in columns 5 and 9 the number of infeasible instances for which the number of unscheduled requests found by GTS is one, since for instances proved infeasible, having only one unscheduled request is the best solution that can be achieved. For five out of the nine infeasible instances for RT=22, GTS finds one unscheduled request (for the other four instances, the solutions produced by GTS have two unscheduled requests each). For the 75%vehicles problem set, GTS finds all the feasible solutions except for one, a5-60 (Tabu Search also did not find a solution for this instance, however CP did). Similarly to Tabu Search and CP, GTS does not find a feasible solution for b7-84 – CP could not solve this instance in the allocated time. The best schedules found for both of these instances by GTS have one unscheduled request each. Also, GTS with EST-first and min-tardiness finds a feasible solution for b8-96; Tabu Search also found a solution for this instance, however, CP did not.

The results show that the min-tardiness insertion heuristic for the GTS runs is enabled by the EST-first ordering of the requests to produce the best performance on the benchmark instances. We conjecture that min-tardiness is better able to preserve the slack in the time windows for the pick-ups/drop-offs, by scheduling them as early in their time window as possible and thus mitigating the effect of time window fragmentation by retaining flexibility in the time window. In this sense, min-tardiness is similar to the max-availability insertion heuristic used with **TaskSwap** in (Kramer and Smith 2004).

The results also show that even though GTS searches a rather limited portion of the search space, confined around the initial solution as built by GBS, with the exception of one instance in the 75%vehicles set, the results for these feasibility benchmark problems match the results reported by more extensive optimization techniques as CP and Tabu Search. In terms of the CPU time, the runs without GTS took on average 1.4 seconds, with a minimum of 0.12 seconds and a maximum of 5.9 seconds. GTS run times varied between 0.9 and 10.3 seconds; the average and median CPU times for GTS were 1.8 and 2.4 seconds respectively, comparable to the median CPU times for CP and Tabu Search, which were 1.9 and 1.4 seconds, respectively. While for some instances GTS is slightly slower, extended CP and Tabu Search run times were recorded for a number of instances, with a maximum of 109 seconds for CP and 78 seconds for Tabu Search (the maximum CPU time for GTS was just 10.3 seconds).

In comparing GTS to CP and Tabu Search, it is important to note that GTS addresses two additional objectives. First, GTS was designed to operate as an incremental search technique in an environment where schedules are modified as they execute. To minimize the confusion of humans executing the schedules, the insertions of new requests must emphasize maintaining schedule stability. GTS minimizes the number of schedule modifications by moving tasks only as needed to make room for the unscheduled requests. Second, the types of problems to which GTS is applied require that all requests must be satisfied, even if some constraints have to be relaxed. GTS, with its underlying STN representation facilitating constraint relaxation, was designed to be coupled with relaxation strategies for generating solutions to oversubscribed problems.

A Real World Paratransit Application

While we evaluated GTS on the benchmark problems, it was designed to operate within a real-world application for helping paratransit dispatchers manage the allocation of trip requests to vehicles through the myriad of unexpected events that occur during the day's execution. This system is being integrated into the operations at ACCESS Transportation Systems, the largest paratransit organization in the Southwestern Pennsylvania Region.

The ACCESS version of the DARP is more complex than that in the benchmark problems. The allowable pick-up window is from ten minutes before the preferred time to twenty minutes after it. The latest drop-off time is constrained by a maximum ride-time constraint, which is the larger of twenty minutes or twice the direct transit time from the pick-up location to the drop-off location. No trip can last longer than two hours. Also, the travel model is more complex in ACCESS in that, in addition to distance, it considers the time-of-day when travel occurs, with travel within rush-hour resulting in longer durations. A problem consists of the set requests for a single day and a configuration of vehicles. The scale of these problems is much larger than the benchmark problems, with a typical problem having between 30 and 50 vehicles servicing between 850 and 950 requests.

Relaxation

In the real-world application, GTS is invoked in the context of already having an existing schedule executing and a new request has to be added in. In ACCESS, it is not acceptable to refuse a request, i.e., all requests must be serviced. In the case where the vehicles are oversubscribed, the service quality constraints must be relaxed in order to service the new request. This relaxation introduces delay for the requests. The dispatcher's objective is to minimize the amount of relaxation/delay that has to be introduced. Naturally, the best

Ordering heur.	Min-added-dur				min-tardiness			
	GBS total #unsch.	GTS total #unsch.	% unsch. sched.	#infeas. inst.	GBS total #unsch.	GTS total #unsch.	% unsch. sched.	#infeas. inst.
None	87	32	69%	19(13)	85	29	72%	17(11)
EST	66	34	52%	20(14)	79	31	67%	16(8)
Most-constr	72	39	49%	19(10)	63	40	38%	21(13)

Table 1: Total number of unscheduled requests before and after GTS, average percentage of unscheduled requests assigned by GTS and total number of infeasible instances after running GTS for the RT=22 and 75%vehicles sets.

case is when a feasible, i.e., non-relaxed, insertion option (a pair of slots for the pick-up and drop-off tasks) is available to service the new request. The base insertion procedure employed by both GTS and GBS is used to find feasible options that are easy for the dispatcher to implement. If none are found, then GTS is applied to find a more complex to implement but still feasible option for the new request. Finally, if GTS cannot find an option, then the new request's constraints, i.e., the latest pick-up and drop-off times, are incrementally relaxed by a constant amount until an option can be found using the base insertion procedure. In the experiments we describe below, a key metric is the amount constraints have to be relaxed in order to accommodate the delay in the new requests.

Experimental Results

The results in this section were produced for a set of five days of data, from February 21 to February 25, 2011, provided to us by a local paratransit company. The company employs three types of vehicles: sedans (with a capacity of four ambulatory passengers), shoppers (capacity 14 ambulatory passengers), and vans (seating both ambulatory and wheelchair passengers, with a maximum of 10 ambulatory, or four ambulatory and three wheelchair passengers). Table 2 summarizes the main characteristics of the five days of data in terms of number of requests received and number of vehicles available each day to allocate the requests.

Day	Reqs.	Sedans	Shoppers	Vans
Day1	844	7	9	15
Day2	827	12	7	15
Day3	902	12	9	15
Day4	797	11	10	14
Day5	884	14	6	16

Table 2: Number of requests and number of vehicles available for each of the five days.

We ran GBS and GTS for the five days of data. We used the algorithm settings that produced the best results for the benchmark problems: the unscheduled requests were sorted in increasing order of their earliest-start-time for the pick-ups (EST-first), and min-tardiness was used as the insertion heuristic. The results show that GTS is able to build feasible schedules for all five days. Most days the paratransit company's vehicle schedule is oversubscribed; since having unscheduled requests is not an option, the constraints for such requests are relaxed until all the requests can be scheduled. To assess the change in algorithm performance for various level of oversubscription, we reduced the number of available vehicles each day to a certain percentage of the original number (the new number of vehicles is computed by taking the ceiling of the product of the desired percentage and the original count for each of the vehicle types).

The runs for 95% showed that all the problems were still feasibly solved; we proceeded then to reduce the vehicle count to 90%, 85% and finally 75%. For each run we collected the number of unscheduled requests and the amount of delay needed to fit these requests in the schedule (in minutes). The results are summarized in Tables 3 and 4.

The results show that as resource levels are reduced, some but not all the requests get scheduled by GTS. For the 90% resource level, on average 85% of the unscheduled requests at the end of the GBS run are assigned by GTS. For the 85% resource level, on average 78% of the unassigned requests after GBS get scheduled by GTS. Finally, for the 75% resource level, on average 53% of the unassigned requests get scheduled by GTS. In terms of CPU times, when GTS is applied to the five-day data using the original vehicle configurations described in Table 2, it averages 6.7 seconds for adding a new request. This performance is well within the usability constraints for paratransit dispatchers.

As expected, the schedules produced by GBS plus relaxation display larger amounts of total delay than the schedules produced by GTS plus relaxation. Surprisingly, there seems to be a trend in our results showing that the average delay per request decreases with the use of GTS (even though after running GTS there are more scheduled requests in the schedule).

Day	Orig.		90%		85%		75%	
	GBS	GTS	GBS	GTS	GBS	GTS	GBS	GTS
Day1	5	0	15	0	20	2	33	13
Day2	1	0	2	0	5	1	27	14
Day3	1	0	8	1	12	3	31	17
Day4	0	0	13	2	10	1	23	5
Day5	1	0	6	1	15	7	29	19

Table 3: Results for the number of unscheduled requests.

Conclusions

TaskSwap was shown to be an efficient iterative repair search procedure for a large-scale mission scheduling problem. Motivated by a real-world DARP, we generalized

	Orig.		90%		85%		75%	
Day	GBS	GTS	GBS	GTS	GBS	GTS	GBS	GTS
Day1	252.6	0	988.9	0	1257.9	80.2	1866.1	594.2
Day2	36.9	0	206.6	0	310.2	53.4	2771.3	1248.7
Day3	10.5	0	250.6	54.6	556.8	15.9	2207.4	1035.7
Day4	0	0	265.1	17.8	320.4	6.9	1351.7	246.2
Day5	57.3	0	211.4	52.7	557.5	285.7	1843.3	1238.3

Table 4: Results for the total delay (in minutes).

TaskSwap to handle the complexity of the constraints in this application. We evaluated the performance of the new technique (GTS) both on a set of difficult synthetic feasibility benchmark problems and on a set of real-world instances. The results show that: 1) GTS matches the performance results reported by more sophisticated, extended search techniques on the benchmark instances and 2) for the real-world instances, GTS significantly improved on the results of a greedy search approach in terms of both increasing the number of feasibly scheduled requests and reducing the total amount of delay.

Acknowledgements

The research described in this paper was funded in part by a grant from the 2011 Federal Transit Administration (FTA) New Freedoms Initiative, the Department of Defense Advance Research Projects Agency (DARPA) under US Army Award No. W911NF-11-2-0009, and the Robotics Institute at Carnegie Mellon University. Any opinions findings and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of DARPA and FTA.

References

Berbeglia, G.; Pesant, G.; and Rousseau, L.-M. 2011. Checking the feasibility of dial-a-ride instances using constraint programming. *Transportation Science* 45:399–412.

Cesta, A.; Oddi, A.; and Smith, S. F. 2000. Iterative flattening: A scalable method for solving multi-capacity scheduling problems. In *Proceedings of The Seventeenth National Conference on Artificial Intelligence*.

Chien, S.; Knight, R.; Stechert, A.; Sherwood, R.; and Rabideau, G. 2000. Using iterative repair to improve responsiveness of planning and scheduling. In *Proceedings of the Fifth International Conference on Artificial Intelligence Planning and Scheduling*, 300–307.

Cordeau, J.-F., and Laporte, G. 2007. The dial-a-ride problem: models and algorithms. *Annals OR* 153(1):29–46.

Cordeau, J.-F. 2006. A Branch-and-Cut Algorithm for the Dial-a-Ride Problem. *Operations Research* 54(3):573–586.

Dechter, R.; Meiri, I.; and Pearl, J. 1991. Temporal constraint networks. *Artificial Intelligence* 49:61–95.

Jain, S., and Van Hentenryck, P. 2011. Large neighborhood search for dial-a-ride problems. In *Proceedings of the 17th international conference on Principles and practice of constraint programming*, CP'11, 400–413. Berlin, Heidelberg: Springer-Verlag.

Kramer, L. A., and Smith, S. F. 2003. Maximizing flexibility: A retraction heuristic for oversubscribed scheduling problems. In Gottlob, G., and Walsh, T., eds., *IJCAI*, 1218–1223. Morgan Kaufmann.

Kramer, L., and Smith, S. F. 2004. Task swapping for schedule improvement - a broader analysis. In *Proceedings 14th International Conference on Automated Planning and Scheduling (ICAPS 04)*.

Kramer, L.; Barbulescu, L.; and Smith, S. F. 2007. Understanding performance tradeoffs in algorithms for solving oversubscribed scheduling problems. In *Proceedings 22nd Conference of the Association for the Advancement of Artificial Intelligence (AAAI-07)*.

Lim, A., and Zhang, X. 2007. A two-stage heuristic with ejection pools and generalized ejection chains for the vehicle routing problem with time windows. *INFORMS Journal on Computing* 19(3):443–457.

Nagata, Y., and Brysy, O. 2009. A powerful route minimization heuristic for the vehicle routing problem with time windows. *Operations Research Letters* 37(5):333–338.

Zweben, M.; Daun, B.; Davis, E.; and Deale, M. 1994. *Scheduling and Rescheduling with Iterative Repair*. Morgan Kaufmann Publishers. chapter 8.