Operations Research B

Lars Burghardt, Alexander Wördekemper, Ahmad Hashemi

22.02.2017

Contents

1 Initial solution

For our initial solution we decided to use a sequential construction algorithm with a hill climbing algorithm introduced in [$HoMu12$]. We have chosen these algorithms because they seemed to produce good and fast initial solutions in the literature. We had to do some little adjustments because we could not open as many routes as we want because we have a fix number of vehicles. In line 1 of Algorithm 1 we initialize an empty solution. Then we repeat the following until all customers have been inserted into a route or the number of routes is the number of vehicles we have: Initialize an empty route r (line 3). Then we have a for loop for all unassigned customers. In line 5 we get a random customer i. Then we insert the customer i at the end of the current route. In line 7 we call the hill climbing algorithm to improve the route r. If the new toot is feasible we mark customer i as inserted. If it is not feasible we remove i from the route again. After the for loop for all unassigned customers has finished we add the route r to the solution in line 12.

In the hill climbing see Algorithm 2 to improve the routes we have given a route r. Then we repeat the following until no improvement is achieved in the previous pass: The for loop for each possible pair of locations starts (line 3). If the latter location is more urgent in its upper time window (line 4) then we swap the current two locations in r to get a new route r'. In line 6 we calculate the cost function for r and r'. If the new route has a better cost function value we set $r \leftarrow r'$.
The cost function is dependent of the route duration, the number of time window violations and the number of capacity violations. Their weights w1, w2 and w3 have to be equal to 1.

Algorithm 1 Sequential construction

Initialize an empty solution s
repeat
    Initialize an empty route r
    for (All unassigned customers) **do**
        Get a random next customer i
        Insert the customer i at the end of the current route r
        Call HC (Algorithm 2) to improve route r
        if (route r is feasible) **then**
            Mark i as inserted
        else
            Remove i from route r
    Add route r to solution s
until (All customers have been inserted OR $|s| = |Vehicles|$)

Algorithm 2 Hill climbing

1: Given a route r
2: **repeat**
3:     **for** (Each possible pair of locations) **do**
4:         **if** (The latter location is more urgent in its upper time window bound) **then**
5:             Swap the current two locations in r to get a new route r'
6:             Calculate $cost(r)$ and $cost(r')$
7:             **if** $(cost(r) - cost(r') > 0)$ **then**
8:                 $r \leftarrow r'$
9: **until** (Done) {Stop when no improvement achieved in the previous pass}

1.1 Different approaches

As we found out that we did not find initial solutions for larger instances we tried some different approaches to solve the problem.

1.1.1 Removing the customer with the largest distance

The sequential construction algorithm adds random customers to the route and after the root gets improved by the hill climbing algorithm we remove the customer we had just added if the new route is infeasible. Instead of removing the just added customer we tried to remove the customer which needed the most time on the route. For every customer and it's destination we added the distance from the last node to the customer/destination and from the customer/destination to the next node. We removed the customer which needed the most time. In experiments we found out that this approach was no improvement compared to the one we had before.

1.1.2 Removing a random customer from the route

Another thing we tried was to remove a random customer from the route if the customer we just added resulted in an infeasible solution given by the hill climbing algorithm. The idea was to be able to remove customers from the route again after they have been successfully added before. One problem of our approach is that when we add a customer which is bad for the route but still results in a feasible route we might not be able to add other customers because of the *bad* customer we already added. The random removing strategy should be able to remove such *bad* customers. In tests this approached performed much worse than the one we decided to take.

1.1.3 Parallel construction of the routes

Inspired by $[HoMu12]$ we tried different approaches by constructing the routes parallel. For this we initialized all routes for all vehicles at the beginning. After that we sorted the customers by several different criteria. Dependant on the criteria we chose a customer and add it to a route. We start with the first route. Then we use the hill climbing algorithm to improve the route. If the route is feasible we mark the customer as added. If it is not feasible we remove the customer again and try to add it to the next route.

2 Large neighborhood search

The large neighborhood search tries to improve a given solution. The approach was introduced in [$JaHe11$]. The algorithm gets the parameters maxSize (the maximum number of customers to be removed), range (to increase the neighborhood size progressively), iterations (the number of iterations) and probability (the probability to accept a worse solution).
We have given a solution s and set *current* to the solution s in line 2. Then we have 3 nested for loops in which we produce a new solution *new* by removing randomly i+j customers from the current solution. In line 7 we start another for loop for all removed customers. We add the customer to a randomly selected route r in the new route *new*. then we always improve the route by calling the hill climbing algorithm. After the for loop we generate a random number *pr* between 0 and 1 in line 10. If the new solution is feasible we check if the cost function of the new solution has a better value than the cost function of the current solution. If this is the case or if *pr* is smaller than *probability* we set $current = new$. Then we check if the cust function value of the current solution is smaller than the cost function value of the given solution s (line 14). If it is better we update our solution s by setting it to our *current* solution.

Algorithm 3 LNS ($maxSize, range, iterations, probability$)

1: Given a solution s
2: $current \leftarrow s$
3: **for** $(i \leftarrow 2; i \leq masSize - range; i \leftarrow i + 1)$ **do**
4:     **for** $(j \leftarrow 0; j \leq range; j \leftarrow j + 1)$ **do**
5:         **for** $(k \leftarrow 0; k < iterations; k \leftarrow k + 1)$ **do**
6:             $new \leftarrow$ Remove randomly $i + j$ customer in *current*
7:             **for** (All removed customer) **do**
8:                 Add customer to a randomly selected route r in *new*
9:                 Call HC (Algorithm 2) to improve route r
10:             $pr \leftarrow$ random number between 0 and 1
11:             **if** (*new* is feasible solution) **then**
12:                 **if** ($cost(new) < cost(current)$ OR $pr < probability$) **then**
13:                     $current = new$
14:                     **if** $(cost(current) < cost(s))$ **then**
15:                         $s = current$

3 Relevant work in the literature

In the literature nearly all possible meta heuristics have been tested for the dial-a-ride problem. [$JaHe11$] introduced a large neighborhood search for the problem. It seemed to work well which is the reason why we decided to implement this meta heuristic A different approach was introduced in [$CoLa03$] where they used a tabu search heuristic for the dial-a-ride problem. In [$Parragh\ et\ al.\ 10$] they used a variable neighborhood search to solve the problem. In [$Jørgensen\ et\ al.\ 07$] genetic algorithms were introduced the solve the dial-a-ride problem. Every approach has its pros and cons but in most cases the large neighborhood search gets the best solutions in the literature.

4 How to compile and run the code

Our approach is implemented in C# using Microsoft Visual Studio Enterprise 2015 under Windows. To compile and run the code, simply open the solution file *ORB.DARP.sln* and run the project using *F5* (debug) or *Shift + F5* (no debug). A copy of Microsoft Visual Studio Enterprise 2015 can be downloaded at Dreamspark UPB.
Another method to run the code, is to open a command line window and enter the path to the compiled *orbdar.exe* with the additional parameters.

Example: *orbdar.exe [optional: 60] gen_10_2_75_8_10_1.darp*

5 Experimental investigation of our approaches components and performance

Instance	Index	AVG initial solution	Best solution	AVG improvement
gen_10_2_75_8_10	1	1387	945	31,87%
	2	1228	1226	0,16%
	3	1097	1097	0%
	4	1339	1268	5,30%
	5	1137	1137	0%
gen_8_2_75_8_10	1	2524	2288	9,35%
	2	2417	1963	18,78%
	3	2256	2074	8,07%
	4	2982	2172	27,20%
	5	2188	1919	12,29%

Table 1: Performance comparison of our LNS approach

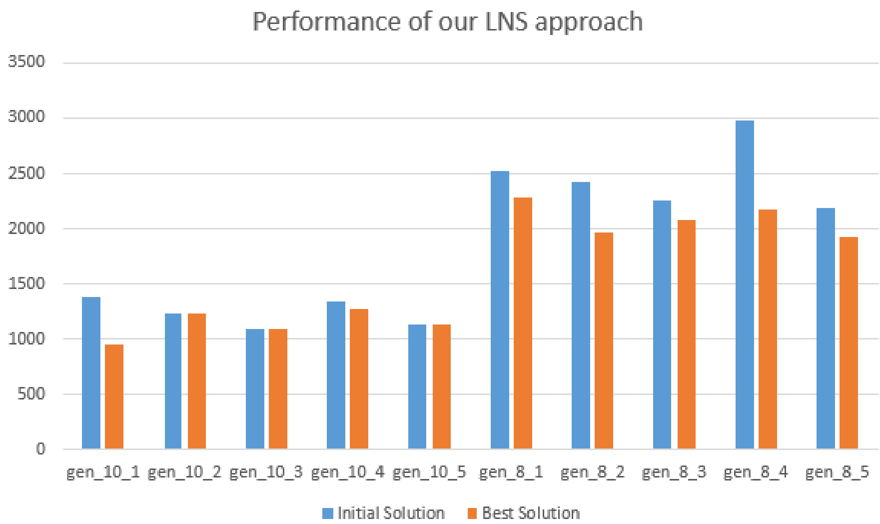

Figure 1: Performance of our LNS approach

6 Literature

[$HoMu$12] M. I. Hosny and C. L.Mumford, "Constructing initial solutions for the multiple vehicle pickup and delivery problem with time windows", Journal of King Saud University, Computer and Information Sciences, vol. 24, no. 1, pp. 59–69, 2012.

[$JaHe$11] S. Jain , P. Van Hentenryck, "Large neighborhood search for dial-a-ride problems", In: Principles and practice of constraint programming, Notes in computer science, vol. 6876. Springer, 2011.

[$CoLa$03] J.-F. Cordeau, G. Laporte, "A tabu search heuristic for the static multi-vehicle dial-a-ride problem", Transportation Research Part B 37: 579–594, 2003

[$Parragh\ et\ al.$ 10] S.N. Parragh, K.F. Doerner, R.F. Hartl, "Variable neighborhood search for the dial-a-ride problem", Computers & Operations Research 37 (6),1129–1138, 2010.

[*Jørgensen et al.* 07] R.M. Jørgensen , J. Larsen, K.B. Bergvinsdottir, "Solving the dial-a-ride problem using genetic algorithms", J Oper Res Soc 58:1321–1331, 2007.