A Hybrid Approach for the Capacitated Vehicle Routing Problem with Time Windows

Ilya Bychkov and Mikhail Batsyn

Laboratory of Algorithms and Technologies for Network Analysis,
National Research University Higher School of Economics,
136 Rodionova, 603093, Nizhniy Novgorod, Russia
ibychkov@hse.ru, mbatsyn@hse.ru

Abstract. The Vehicle Routing Problem (VRP) is one of the most popular combinatorial optimization problems which is closely related to the real-life optimization challenges. Being developed for more than 60 years the problem has been considered in many different formulations. In real-life goods distribution such constraints as fleet size and mix, site-dependency constraints, hard and soft time windows, vehicle capacity constraints are very important. In this paper we consider Capacitated Vehicle Routing Problem with hard Time Windows. We propose a hybrid heuristic algorithm which contains elements of ant colony optimization strategy and tabu search technique. Our algorithm shows good performance and results for the well-known Solomon dataset.

Keywords: Vehicle routing problem · Time windows · Hybrid algorithm · Tabu search · Ant colony optimization

1 Introduction

The Vehicle Routing Problem (VRP) is one of the most popular combinatorial optimization problems which is closely related to the real-life optimization challenges. The problem's community is quite large - various VRP challenges and knowledge databases can be found all over the Internet. In everyday life many distribution companies use specific algorithms and software to solve different variations of VRP. The importance of the problem grows today with the achievements in drone delivery and unmanned vehicles popularization. Modern VRP formulations usually contain many different types of constraints. To our experience in case of distribution companies the most critical constraints are hard and soft time windows (VRPTW). Other research papers also confirm the vitality of time windows constraints [5, 11, 12].

There are many heuristic algorithms with promising results. Jawarneh & Abdullah [3] present the adaptive bee colony optimization algorithm with sequential insertion heuristic for initial solution. The authors use several neighborhoods based on swap and shift moves. Lau et al. [6] modeled VRPTW as a linear constraint satisfaction problem and introduced an efficient local search method for solving it. Braysy & Gendreau [1] proposed an adaptation of a popular tabu search metaheuristic for solving VRPTW.

Hybrid algorithms where several heuristics or metaheuristics ideas are combined are widely used for solving various combinatorial optimization problems [4, 7, 8, 10]. In this paper we introduce a new hybrid ant colony optimization algorithm combined with tabu search technique in which we allow visiting infeasible solutions during the search process. Allowing infeasible solutions is inspired by a unified tabu search algorithm of Cordeau et al. [2]. Our algorithm shows good performance and results for the well-known Solomon's dataset [9].

2 Mathematical Formulation

In this paper we consider the Capacitated Vehicle Routing with Time Windows (CVRPTW). In this formulation we are given a limited number of vehicles with the same capacity to serve customers in the specified time intervals. In practice these intervals called time windows may refer to customer working hours or concrete hour when it is convenient for a customer to unload a vehicle. Time window constraints are known to be the hardest part of VRPTW [5].

According to CVRPTW formulation each customer i has an associated pair of values $[e_i, l_i]$ which represent the earliest and the latest time when unloading can be started. However, if a vehicle arrives before a customer i is ready to start service we assume that the vehicle waits for the beginning of the time window e_i. On the other hand, any vehicle is allowed to finish servicing customer i even after the right bound of time window l_i.

We consider the CVRPTW formulation in which we have a set K of identical vehicles with capacity Q. The number of vehicles is $|K|$, but it is not required to use all of them.

A straightforward mathematical formulation for CVRPTW is given in [11]. Here we have graph $G = (V, A)$ with a set of nodes V representing customers and a set of arcs A representing roads between customers. There are n $= |V|$ customers numbered from 1 to n. Also there are two auxiliary vertices with numbers 0 and $n + 1$ representing the depot node for route start and finish respectively. We are also given the cost matrix C where c_{ij} indicates the cost of traveling from customer i to customer j and the matrix of traveling times T where it takes t_{ij} time units to get from customer i to customer j. For each customer i there are time window $[e_i, l_i]$, demand q_i and service time s_i. Let us define $\Delta^+(i)$ the set of nodes directly reachable from i and $\Delta^-(i)$ - the set of nodes from which i is directly reachable. Parameters E, L determine the earliest time when a vehicle can leave the depot and the latest time when it can return.

The decision variables are specified as follows:

$$x_{ijk} = \begin{cases} 1, & \text{if arc } (i,j) \text{ is used by vehicle } k \\ 0, & \text{otherwise} \end{cases}$$

$$w_{ik} = \begin{cases} \text{service start time, if customer } i \text{ appears in the route of vehicle } k \\ 0, \quad \text{otherwise} \end{cases}$$

With all the variables and parameters above we can now formulate the problem considered in this paper as follows

(CVRPTW):

$$min \quad \sum_{k \in K} \sum_{(i,j) \in A} c_{ij} x_{ijk} \tag{1}$$

Subject to:

$$\sum_{k \in K} \sum_{j \in \Delta^{+}(i)} x_{ijk} = 1 \quad \forall i \in V, \tag{2}$$

$$\sum_{j \in \Delta^{+}(0)} x_{0jk} = 1 \quad \forall k \in K, \tag{3}$$

$$\sum_{i \in \Delta^{-}(n+1)} x_{i,n+1,k} = 1 \quad \forall k \in K, \tag{4}$$

$$\sum_{i \in \Delta^{-}(j)} x_{ijk} - \sum_{i \in \Delta^{+}(k)} x_{jik} = 0 \quad \forall k \in K, j \in V \tag{5}$$

$$x_{ijk}(w_{ik} + s_i + t_{ij} - w_{jk}) \leq 0 \quad \forall k \in K, (i,j) \in A, \tag{6}$$

$$e_i \sum_{j \in \Delta^{+}(i)} x_{ijk} \leq w_{ik} \quad \forall k \in K, i \in V, \tag{7}$$

$$l_i \sum_{j \in \Delta^{+}(i)} x_{ijk} \geq w_{ik} \quad \forall k \in K, i \in V, \tag{8}$$

$$w_{0k} \geq E \quad \forall k \in K, \tag{9}$$

$$w_{n+1,k} \leq L \quad \forall k \in K, \tag{10}$$

$$\sum_{i \in V} q_i \sum_{j \in \Delta^{+}(i)} x_{ijk} \leq Q \quad \forall k \in K, \tag{11}$$

$$x_{ijk} \in \{0,1\} \quad \forall k \in K, (i,j) \in A. \tag{12}$$

Objective function (1) minimizes the total cost of all the routes in the solution. Constraint (2) ensures that each customer is assigned to exactly one route. Constraints (3), (4) and (5) guarantee that we have only one outcoming edge from the start depot vertex, only one incoming edge to the end depot vertex and each customer has the same number of incoming and outcoming edges. Inequality (6) prohibits service to start earlier than the earliest possible arrive time from the previous customer taking into account service and travel times. Inequalities (7), (8), (9), (10) provide feasibility with respect to the given time windows. Finally, constraint (11) prohibits overloaded routes with the total demand greater than vehicle capacity Q.

3 Algorithm Description

During the recent years metaheuristic algorithms have become a popular way to solve any combinatorial optimization problem. The so called hybrid methods which incorporate strategies from different metaheuristic approaches are widely used. In this paper we present a hybrid heuristic algorithm which contains Ant Colony Optimization (ACO) and Tabu Search (TS) with the possibility of visiting infeasible solutions during the search. In section 3.3 we provide empirical results which confirm that our hybrid algorithm outperforms separate runs of TS and ACO procedures. The effect of visiting infeasible areas during the search is discussed in Section 3.4.

Algorithm 1 Ant Colony - Tabu Search approach

```
 1: procedure ACO-TS()
 2:     $maxIterations \leftarrow 30$
 3:     $candidateAco \leftarrow \emptyset$
 4:     $candidateTS \leftarrow \emptyset$
 5:     $bestSolution \leftarrow \emptyset$
 6:     for $i \leftarrow 1, maxIterations$ do
 7:         $candidateACO \leftarrow$ runACO($candidateTS$)
 8:         updateIfBetter($bestSolution, candidateACO$)
 9:         if hasUnserved($candidateACO$) $= True$ then
10:             fixUnserved($candidateACO$)
11:         end if
12:         $candidateTS \leftarrow$ runTS($candidateACO$)
13:         updateIfBetter($bestSolution, candidateTS$)
14:     end for
15:     return $bestSolution$
16: end procedure
```

Algorithm 1 provides the top level procedure of our approach. Here we have the main loop which sequentially runs the ant colony algorithm and then tabu search. Variables *candidateACO*, *candidateTS* and *bestSolution* represent best solutions found by ACO, TS and overall best respectively. Variable *maxIterations* limits the number of iterations of the main loop. Inside the loop RUNACO(*candidateTS*) is called and the best solution found by this function is saved to *candidateACO*. RUNACO(*candidateTS*) receives the best solution found by the previous TS function run and uses it to update initial pheromone values. For the first time when *candidateTS* is empty all pheromones are initialized with the same value that will be discussed later. Then the algorithm calls UPDATEIFBETTER(*bestSolution*, *candidateACO*) which replaces the best solution found so far with *candidateACO* if candidate solution is better. Our ACO solution is constructed by adding routes one by one. This sometimes leads to infeasible solutions where some customers are unserved. Since the TS function cannot work with such kind of infeasibilities we run FIXUNSERVED(*candidateACO*) procedure to fix it. This procedure transforms all unserved customers into one-customer routes using additional vehicles. Such transformation in its turn can lead to using more vehicles than we have. However, this type of violations can be easily fixed in tabu search function. Then *candidateACO* solution is passed to RUNTS(*candidateACO*) function and used as a starting solution for the search. After TS function is finished the results are saved to *candidateTS* variable and the best solution is updated if necessary inside UPDATEIFBETTER() .

3.1 Ant Colony Optimization Algorithm

Ant colony optimization strategy is used to obtain an initial solution and then to escape from the local optimum obtained by the tabu search part. In this algorithm each ant constructs a solution in a probabilistic manner using pheromone values associated with every single arc. Initially all pheromone values are set to the same value equal to the total cost of the solution where every customer is served with its own vehicle. This means that initially all the arcs can become a part of the solution under construction with the same probability. Here we use the solution with one-customer routes to fill initial pheromone values. The algorithm for our ACO stage is presented in Algorithm 2. The detailed description is provided below.

Algorithm 2 starts with defining some variables. The first variable which is called *maxIterationsWithoutImprovement* defines how many iterations without improving the current best solution we can run. The next variables which are *iterations*, *colonySize* and *best* define current iteration counter, number of ants generated in each iteration and the best solution found so far respectively. At first procedure UPDATEPHEROMONES(*startSolution*) is called. This procedure gets a solution *startSolution* (or generally a list of solutions), iterate over it and increase current pheromone values for every edge in a solution by $1/c(startSolution)$ where $c(startSolution)$ is the cost of *startSolution*. As a result, it allows us to connect the parts of our approach by passing the TS result to ACO part and letting ACO search to prioritize elements of TS solution.

After that the main loop of ACO algorithm starts. The stopping criterion in this case is reaching *maxIterationsWithoutImprovement* iterations without

Algorithm 2 Ant Colony Optimization algorithm

```
 1: procedure runACO($startSolution$)
 2:     $maxIterationsWithoutImprovement \leftarrow 100$
 3:     $iterations \leftarrow 0$
 4:     $colonySize \leftarrow 100$
 5:     $best \leftarrow \emptyset$
 6:     updatePheromones($startSolution$)
 7:     while $iterations < maxIterationsWithoutImprovement$ do
 8:         $oldBest \leftarrow best$
 9:         $localBestList \leftarrow \emptyset$
10:         $current \leftarrow \emptyset$
11:         for $i \leftarrow 1, colonySize$ do
12:             $current \leftarrow$ runSingleAnt()
13:             if isBetter($current, best, 0$) $= True$ then
14:                 $best \leftarrow current$
15:                 $iterations \leftarrow 0$
16:             end if
17:             if isBetter($current, bestLocal, 5$) $= True$ then
18:                 $bestLocalList \leftarrow bestLocalList \cup current$
19:             end if
20:         end for
21:         updatePheromones($bestLocalList$)
22:         evaporate()
23:         $iterations \leftarrow iterations + 1$
24:     end while
25:     return $best$
26: end procedure
```

improving the best solution. Inside the main loop we have variables *oldBest*, *localBestList* and *current* which represent the best solution in the previous iterations, the list of best solutions in the current iteration and the current solution under consideration. Variable *oldBest* is needed to compare current iteration solutions only to the solutions from the previous iterations, *localBestList* is used to update pheromone values at the end of iteration.

Then an inner loop starts where a colony of ants is formed. Every single solution for the problem which is represented by a single ant is constructed by calling $current \leftarrow$ runSingleAnt(). Then we have to update the current best solution if necessary and add *current* solution for future pheromone update if this solution is good enough. Both of these actions are done with the help of isBetter($solution1, solution2, gap$) function which gets two solutions and gap value as parameters. If $solution1$ has a lower cost than $solution2$ or $solution1$ cost is within the *gap* percent from $solution2$ the function returns true. Since runSingleAnt() can produce solutions where some customers are unserved, isBetter($solution1, solution2, gap$) returns true value immediately if the number of unserved customers in $solution1$ is strictly less than in $solution2$.

We call isBetter($current, best, 0$) to test if a better solution than the current best is found. Also isBetter($current, bestLocal, 5$) is called when the algorithm determines which solutions in the current iteration can be used for future pheromones update. If a solution's cost is within 5% from the best one obtained in the previous iterations or it has less unserved customers - it is added to *localBestList* list. After the whole colony is generated we update the pheromone values in updatePheromones(*bestLocalList*). Then the pheromone evaporation procedure evaporate() is called. It reduces the pheromone value on every edge: $pheromone_{ij} = pheromone_{ij} * \rho$. Here ρ is the evaporation rate parameter empirically set to 0.8.

Algorithm 3 Creating a single ant solution

```
1: procedure runSingleAnt()
2:     $solution \leftarrow \emptyset$
3:     repeat
4:         $isInserted \leftarrow$ addNewRoute($solution$)
5:     until $isInserted == True$ & getRoutesNumber($solution$) $< |K|$
6:     return $solution$
7: end procedure
```

Every candidate solution in our ACO approach is constructed using runSingleAnt(), addNewRoute() and insertNextCustomer() procedures. These procedures are described in Algorithm 3, Algorithm 4 and Algorithm 5 respectively. Function runSingleAnt() from Algorithm 3 forms a new solution from

Algorithm 4 Procedure for adding new routes

```
1: procedure addNewRoute($solution$)
2:     $unservedCustomers \leftarrow$ getUnservedCustomers($solution$)
3:     $route \leftarrow \emptyset$
4:     if $unservedCustomers = 0$ then
5:         return $False$
6:     end if
7:     repeat
8:         $isInserted \leftarrow$ insertNextCustomer($route, unservedCustomers$)
9:     until $isInserted = True$
10:    if isEmpty($route$) then
11:        return $False$
12:    end if
13:    addRoute($solution, route$)
14:    return $True$
15: end procedure
```

scratch. Step by step it adds a new route by calling addNewRoute(*solution*).

The solution construction terminates when ADDNEWROUTE(*solution*) returns false (there are no possible routes to add) or when the total number of routes equals to the number of vehicles available. Algorithm 4 describes a new route creation process. At first we check if there are unserved customers in the *solution* and set the current route under creation as an empty route. Then we call procedure INSERTNEXTCUSTOMER(*route*, *unservedCustomers*) to extend *route* until there is a customer who can be added without any constraint violations. Finally, ADDROUTE(*solution*, *route*) is called to append a new route to the *solution*.

Procedure INSERTNEXTCUSTOMER(*route*, *unservedCustomers*) which is presented in Algorithm 5 appends a new customer to *route* with some probability. This probability depends on the current pheromone values for the arc from the last customer to a new one and the cost of this arc. The list of probabilities for every customer is initially empty. Then the algorithm iterates over all the unserved customers. If the customer c can be inserted to the end of *route* without time window and load violations we define its probability to be chosen as $\frac{pheromones[prev][c]}{costs[prev][c]}$. After iterating over all the unserved customers we check if there is a non-null value in *probs* for at least one customer by calling ISEMPTY(*probs*). Strictly speaking values in *probs* cannot be called probabilities so we normalize all these values with NORMALIZE(*probs*). Then we choose a customer to be added to the *route* using *probs* as probabilities inside GETRANDOM(*probs*, *unservedCustomers*) and then add it in ADDCUSTOMER(*nextCustomer*, *route*).

Algorithm 5 Customer insertion procedure

```
 1: procedure INSERTNEXTCUSTOMER($route, unservedCustomers$)
 2:     $probs \leftarrow \emptyset$
 3:     $prev \leftarrow$ GETLASTCUSTOMER($route$)
 4:     for $c \in unservedCustomers$ do
 5:         if ISFEASIBLEINSERTION($c, route$) $= True$ then
 6:             $probs_c \leftarrow \frac{pheromones[prev][c]}{costs[prev][c]}$
 7:         end if
 8:     end for
 9:     if ISEMPTY($probs$) then
10:         return $False$
11:     end if
12:     NORMALIZE($probs$)
13:     $nextCustomer \leftarrow$ GETRANDOM($probs, unservedCustomers$)
14:     ADDCUSTOMER($nextCustomer, route$)
15:     return $True$
16: end procedure
```

3.2 Tabu Search Algorithm

After ant colony part is finished we use its best solution to improve it with tabu search technique. Our algorithm is inspired by the unified tabu search from [2]. We consider a single neighborhood structure $N(S)$ obtained by moving any customer i in solution S from its current position to another position in the same route or another route within the solution S.

One of the features of our algorithm is the possibility of forming new routes, which is defined as moving a customer from its current route to a new route. Also we allow all kind of constraints to be violated at some cost. Let $c(S)$ be the total cost of solution S, $load(r)$ be the total load of some route r inside S, $tw(r)$ be the total lateness time for route r. For each type of infeasibility (time windows, load, routes number) we introduce a control coefficient. These coefficients are α, β, γ for time windows, load and routes number violations respectively. The key goal of these control coefficients is to force the tabu search algorithm to decrease infeasibilities when we stay in the infeasible area for too long. This goal is achieved by adding penalties for each type of violations to the cost function. Therefore total solution cost is defined as:

$$cost(S) = c(S) + \alpha \sum_{r \in S} tw(r) + \beta \sum_{r \in S} \max(0, load(r) - Q) + \gamma \max(0, |S| - |K|) \tag{13}$$

where $|S|$ is the total number of routes in the solution S. To prevent the search from visiting only infeasible solutions we control the time spent inside infeasible area by using dynamically adjusted control parameters α, β, γ. Initially all these parameters are set to the starting value equal to 1. Each time we move from solution S_m to solution S_{m+1} while, for example, $tw(S_m) < tw(S_{m+1})$, the control parameter α is multiplied by a fixed scaling factor which is set to 1.4. If we move to a solution where violation is increased comparing to the previous one, the corresponding control parameter is always multiplied by the scaling factor. On the other hand, if we move to a solution where violation is decreased the control parameter stays the same. Only when a violation value becomes equal to 0 the control parameter for this particular violation type is reset to 1. This way the search is allowed to visit infeasible areas, but the exponential growth of α, β, γ parameters forces the algorithm to move back to feasible regions.

As a part of tabu search technique we use short term memory to avoid the search moving within some loops. We append moves to a tabu list structure and they become prohibited for a fixed number$tabuTenure = 10$ of iterations. For long term memory strategy we store the number of times each edge has been transited to a solution. Each time we evaluate a move, the algorithm estimates how many times the edges we want to include has been transited to the solution before. The resulting value is also added to the cost function as a penalty. Such an approach allows us to prevent some edges to be transited to a solution too often and let the algorithm to consider other options. It provides global diversification for the search.

The pseudocode for the TS procedure is presented in Algorithm 6. Algorithm 6 starts with defining *iterations*, $maxIterationsWithoutImprovement$ and *best* variables which represent iterations counter, stopping criterion for the main loop and the current best solution respectively. Inside the main loop we declare *bestDelta* which is the best cost difference found between the current solution and a possible candidate solution. Variable $bestMove$ stores the best move found so far. Function fillPossibleMoves($currentSolution$) looks for all possible moves from the current solution. Then the algorithm iterates over all $possibleMoves$ from the previous step. getCost($currentSolution, move$) is run for each move to compute the cost delta. If this *delta* is better than *bestDelta* found so far and *move* is not in the tabu list *bestDelta* is updated. Here we use the first descent strategy and perform the first improving move found. This way we leave the inner loop immediately when an improving move

Algorithm 6 Tabu Search algorithm

```
1: procedure TS($currentSolution$)
2:     $iterations \leftarrow 0$
3:     $maxIterationsWithoutImprovement \leftarrow 500$
4:     $best \leftarrow currentSolution$
5:     while $iterations < maxIterationsWithoutImprovement$ do
6:         $bestDelta \leftarrow \infty$
7:         $bestMove \leftarrow \emptyset$
8:         $possibleMoves \leftarrow$ fillPossibleMoves($currentSolution$)
9:         for $move \in possibleMoves$ do
10:            $delta \leftarrow$ getCost($currentSolution, move$)
11:            if $delta < bestDelta$ & isTabu($move$) $= False$ then
12:                $bestDelta \leftarrow delta$
13:                $bestMove \leftarrow move$
14:                if $delta < 0$ then
15:                    break
16:                end if
17:            end if
18:        end for
19:        updateTabuList($bestMove$)
20:        updateTransitions($bestMove$)
21:        applyMove($currentSolution, bestMove$)
22:        updatePenalties($currentSolution$)
23:        $iterations \leftarrow iterations + 1$
24:        if isBetter($currentSolution, best, 0$) $= True$ then
25:            $best \leftarrow current$
26:            $iterations \leftarrow 0$
27:        end if
28:    end while
29:    return $best$
30: end procedure
```

is found. UPDATETABULIST($bestMove$) marks the customer and the source route from $bestMove$ as tabu active elements. We prohibit any moves in which a customer and a destination route are tabu active for $tabuTenure$ iterations, where $tabuTenure$ equals to 10. Long term memory is updated by UPDATETRANSITIONS(). Then we apply the best move found with APPLYMOVE($currentSolution, bestMove$) and adjust the control coefficients α, β, γ.

3.3 Hybrid vs. Separate Algorithms

We have performed several computational experiments to test our hybrid approach in comparison to separate runs of ACO and TS algorithms. Our hybrid approach termination criterion has been set to 100 iterations without improvement for ACO part and 500 iterations without improvement for TS. Then we run the hybrid approach for 30 times, find the longest run and set this value as the time limit for a single ACO run. For a fair comparison with a separate ACO algorithm we have run ACO also for 30 times, each time with the determined time limit. For TS algorithm we have started with one-customer routes solution and limited its running time to the total time of 30 hybrid algorithm runs. The results of our experiments are very similar for almost all instances from Solomon's dataset, so we provide only some examples in Table 1. The best results among all the algorithms are in bold. Here we can conclude that on all problems from Solomon's dataset our hybrid approach shows a good performance in comparison with separate ACO and TS runs.

Table 1. Performance tests of the hybrid approach in comparison to separate ACO and TS algorithms

Instance	ACO			Hybrid			TS
	Min	Avg	Max	Min	Avg	Max	
C103	1983,92	2092,47	2158,93	**828,06**	844,84	893,86	904,00
C104	1618,40	1713,18	1780,16	**825,54**	874,79	951,52	878,97
C203	1749,11	1831,31	1911,70	**591,17**	648,53	697,13	623,41
C204	1527,66	1595,26	1653,51	**594,60**	659,12	740,39	702,39
R103	2230,23	2318,88	2390,62	**1238,24**	1263,68	1308,59	1277,47
R104	1900,23	1975,75	2035,35	**1003,98**	1028,00	1069,36	1024,17
R203	1726,29	1780,87	1838,50	**901,73**	949,52	998,20	926,93
R204	1481,46	1516,05	1550,86	**755,36**	802,40	839,78	802,68
RC103	2347,84	2470,44	2555,68	**1301,06**	1360,81	1400,26	1375,22
RC104	1967,23	2148,03	2226,91	**1151,24**	1199,94	1256,72	1188,27
RC203	1873,54	1873,54	2034,34	**959,23**	1014,95	1090,36	1026,28
RC204	1517,43	1626,50	1676,57	**818,09**	865,78	927,12	886,24

Table 2. Performance tests of the infeasible moves feature in TS

Instance	Feasible	Infeasible
C101	1609,62	**828,94**
C102	1939,26	**861,77**
C201	2026,22	**632,05**
C202	2079,49	**677,12**
R101	1972,98	**1694,51**
R102	2076,42	**1499,84**
R201	1839,54	**1223,76**
R202	1636,69	**1090,26**
RC101	2074,31	**1680,83**
RC102	2010,11	**1511,16**
RC201	1926,34	**1323,87**
RC202	1846,68	**1146,75**

Table 3. Results comparison for clustered Solomon instances

Instance	ABCO	ACO-TS				Gap
		Min	Avg	Max	Time	
C101	828,94	**828,94**	831,02	859,82	6,70	0
C102	828,94	**828,94**	831,61	869,73	6,65	0
C103	835,71	**828,06**	844,84	893,86	7,35	0,92
C104	885,06	**825,54**	874,79	951,52	9,12	6,72
C105	828,94	**828,94**	834,68	877,63	7,05	0
C106	828,94	**828,94**	832,05	882,77	7,34	0
C107	828,94	**828,94**	840,29	893,27	7,12	0
C108	831,73	**828,94**	835,76	869,22	8,42	0,34
C109	840,66	**828,94**	842,94	923,80	10,46	1,39
C201	591,56	**591,56**	624,77	682,39	10,09	0
C202	591,56	**591,56**	654,16	748,76	10,45	0
C203	593,21	**591,17**	648,53	697,13	11,46	0,34
C204	606,90	**592,13**	659,12	740,39	9,63	2,43
C205	588,88	**588,88**	647,58	706,69	11,29	0
C206	588,88	**588,49**	648,99	752,19	15,08	0,06
C207	590,59	**588,28**	649,92	708,58	12,26	0,39
C208	593,15	**588,32**	663,74	709,93	6,70	0,81

3.4 Infeasible Moves

Visiting infeasible solutions has been considered for many combinatorial optimization problems. Cordeau et al. [2] used the moves violating time window and load constraints for CVRPTW and obtained good quality results. In our approach we allow only feasible solutions (with respect to time windows and load constraints) to appear during the ACO stage. However, iterative nature of solution construction procedure where routes are added one by one can lead to

Table 4. Results comparison for random Solomon instances

Instance	ABCO	ACO-TS Min	Avg	Max	Time	Gap
R101	**1643,18**	1653,23	1679,98	1702,31	8,97	-0,61
R102	**1476,11**	1488,18	1519,51	1569,05	9,94	-0,82
R103	1245,86	**1236,88**	1263,68	1308,59	8,88	0,72
R104	1026,91	**1003,98**	1028,00	1069,36	11,09	2,23
R105	**1361,39**	1378,94	1404,88	1431,05	8,26	-1,29
R106	1264,50	**1255,16**	1294,08	1335,89	10,02	0,74
R107	1108,11	**1088,68**	1116,52	1144,19	11,23	1,75
R108	994,68	**960,80**	987,04	1037,09	10,86	3,41
R109	1168,91	**1158,20**	1191,54	1231,45	10,63	0,92
R110	1108,22	**1095,43**	1120,85	1153,86	10,31	1,15
R111	1080,84	**1061,61**	1100,18	1124,90	9,85	1,78
R112	992,22	**965,87**	1005,11	1052,81	8,90	2,66
R201	1197,09	**1162,55**	1205,22	1251,18	9,80	2,89
R202	1092,22	**1065,88**	1106,81	1153,03	9,22	2,41
R203	983,06	**892,97**	949,52	998,20	9,56	9,16
R204	845,30	**755,36**	802,40	839,78	9,65	10,64
R205	999,54	**987,93**	1027,36	1062,86	8,91	1,16
R206	955,94	**911,40**	954,97	989,51	9,22	4,66
R207	903,59	**833,26**	876,30	977,59	9,64	7,78
R208	769,96	**726,71**	770,10	804,71	10,06	5,62
R209	935,57	**877,44**	920,14	953,98	9,81	6,21
R210	988,34	**933,52**	978,29	1007,01	9,86	5,55
R211	867,95	**766,48**	813,46	856,551	10,78	11,69

using more vehicles than available. The proposed tabu search procedure allows all kind of infeasible moves to be done (time windows, load, routes number). We have performed some empirical tests to prove the effectiveness and importance of this feature as a part of our algorithm. The performance of the TS procedure has been tested using two scenarios - when infeasible moves prohibited and permitted. In both cases TS starts with the solution where every single customer is served with its own vehicle. This way we have time window and load constraints satisfied at the cost of big number of vehicles. Tabu search procedure is allowed to do all kind of moves to reach the first feasible solution and then infeasible moves become prohibited in one of the scenarios. The experiment has been performed on Solomon's dataset with running time limited to 60 seconds for a single TS run. The results of this experiment show that in each test case allowing infeasible moves leads to much better objective values than allowing only feasible ones. Some examples are presented in Table 2. As a result of the experiment we can report that allowing infeasible moves undoubtedly wins in comparison to only feasible moves. The cost difference is huge for all the problems from Solomon's dataset.

Table 5. Results comparison for random clustered Solomon instances

Instance	ABCO	ACO-TS				Gap
		Min	Avg	Max	Time	
RC101	**1637,40**	1654,62	1688,88	1755,67	8,87	-1,05
RC102	**1486,85**	1494,56	1527,58	1595,39	8,79	-0,52
RC103	**1299,38**	1301,06	1360,81	1400,26	9,50	-0,13
RC104	1200,60	**1151,24**	1199,94	1256,72	10,01	4,11
RC105	**1535,80**	1539,39	1581,33	1630,93	9,15	-0,23
RC106	1403,07	**1392,91**	1433,93	1491,85	9,10	0,72
RC107	1230,32	**1230,30**	1283,76	1372,39	9,75	0,00
RC108	1165,17	**1140,28**	1177,21	1237,36	8,65	2,14
RC201	1315,57	**1291,88**	1344,68	1394,85	10,32	1,80
RC202	1169,72	**1124,99**	1172,53	1229,02	9,93	3,82
RC203	1010,74	**946,69**	1014,95	1090,36	11,37	6,34
RC204	890,28	**818,09**	865,78	927,121	9,47	8,11
RC205	1221,28	**1185,30**	1232,73	1297,56	11,02	2,95
RC206	1097,65	**1082,57**	1135,25	1189,61	10,75	1,37
RC207	1024,17	**992,79**	1049,12	1126,51	9,29	3,06
RC208	864,56	**792,60**	845,72	884,10	9,29	8,32

4 Computational Experiments

We have performed the computational experiments using well known Solomon dataset with 100 customers. Our algorithm is run 30 times for every problem instance and the best results are reported. All experiments are performed on Inter Core i3 3.7 Ghz processor with 8 GB RAM. The algorithm has been programmed in C++ programming language under Windows 10 operating system. The computational results for Solomon dataset are provided in Tables 3, 4 and 5. Each table contains instance name, objective value from [3], our algorithm min, average and max objective value, average running time and the gap value in percent (the difference between [3] and our result). For clustered problem instances (see Table 3) our ACO-TS algorithm results are equal to the results from [3] in 8 of 17 cases. For the rest of test instances we have found better solutions with an improvement from 0,06 to 6,72%. Computations for random Solomon instances are shown in Table 4. Here our algorithm has obtained worse results in 3 of 23 cases with the difference from 0,61 to 1,29% . For the remaining instances our improvement varies from 0,72 to 11,62%. The last test scope is random clustered instances where 4 of 16 problems are solved with worse results (from 0,13 to 1,05%) and for 12 instances the objective value is improved up to 8,32% (see Table 5).

5 Conclusion

In this paper we have presented a new hybrid heuristic approach for solving the Capacitated Vehicle Routing Problem with Time Windows. Our algorithm combines Ant Colony Optimization (ACO) approach and Tabu Search (TS) technique. A move neighborhood with possibility of creating new routes and visiting infeasible areas has been considered to improve the search process. The idea of sequential usage of ACO and TS parts gives major improvements comparing to separate runs of these algorithms. Computational experiments show better results on 41 of 56 considered test instances comparing to the recent results of Jawarneh & Abdullah [3].

Acknowledgement. The research was funded by Russian Science Foundation (RSF project No. 17-71-10107).

References

1. Braysy, O., Gendreau, M.: Tabu search heuristics for the vehicle routing problem with time windows, SINTEF Applied Mathematics, Department of Optimisation, Oslo, Norway, Internal Report STF42 A01022 (2001)
2. Cordeau, J.F., Laporte, G., Mercier, A.: A unified tabu search heuristic for vehicle routing problems with time windows. Journal of the Operational Research Society 52, 928–936 (2001)
3. Jawarneh, S., Abdullah, S.: Sequential insertion heuristic with adaptive bee colony optimisation algorithm for vehicle routing problem with time windows. Plos one 10, 1–23 (2015)
4. Koc, C., Bektas, T., Jabali, O., Laporte, G.: A hybrid evolutionary algorithm for heterogeneous fleet vehicle routing problems with time windows. Computers & Operations Research 64, 11–27 (2015)
5. Laporte, G.: The vehicle routing problem: an overview of exact and approximate algorithms. European Journal of Operational Research 59(3), 345-358 (1992)
6. Lau, H., Lim, Y., Liu, Q.: Diversification of search neighbourhood via constraint-based local search and its applications to VRPTW. In: Proceedings 3rd International Workshop on Integration of AI and OR Techniques (CP-AI-OR). pp. 1–15. Kent, United Kingdom (2001)
7. Minocha, B., Tripathi, S.: Solving school bus routing problem using hybrid genetic algorithm: a case study. Advances in Intelligent Systems and Computing 236, 93–103 (2014)
8. Nai-Wen, L., Chang-Shi, L.: A hyrid tabu search for the vehicle routing problem with soft time windows. In: Yang G. (eds.) Proceedings of the 2012 International Conference on Communication, Electronics and Automation Engineering, Advances in Intelligent Systems and Computing. vol. 181, pp. 507–512. Springer, Heidelberg (2012)
9. Solomon, M.M.: Algorithms for the vehicle routing and scheduling problems with time window constraints. Operations Research 35(2), 254–265 (1987)
10. Subramanian, A., Penna, P., Uchoa, E., Ochi, L.: A hybrid algorithm for the fleet size and mix vehicle routing problem. European Journal of Operational Research 221(2), 285–295 (2012)

11. Toth, P., Vigo, D. : The Vehicle Routing Problem. Society for Industrial and Applied Mathematics, Philadelphia, USA (2002)
12. Toth, P., Vigo, D.: Vehicle Routing: Problems, Methods, and Applications. Society for Industrial and Applied Mathematics. Philadelpia, USA (2014)