A Sum-of-Squares Heuristic for a Dynamic Vehicle Routing Problem

Matthew I.C. Randall∗ Ahmed Kheiri† Adam N. Letchford†

November 2023

Abstract

We consider a simple family of dynamic vehicle routing problems, in which we have a fixed fleet of identical vehicles, and customer requests arrive during the route-planning process. For this kind of problem, it is natural to use an insertion heuristic, possibly in combination with local search. We test several such heuristics computationally. It turns out that a parallel insertion heuristic, based on a certain “sum-of-squares” insertion criterion, performs remarkably well even without local search.

Keywords: dynamic vehicle routing; insertion heuristics; parallel insertion

1 Introduction

Vehicle routing problems (VRPs) are a very well-known class of combinatorial optimisation problems, and there is a huge literature on them, including several books (e.g., [2, 8, 19]). An important distinction in the VRP literature is between static VRPs, in which all of the relevant data is known before the routes need to be planned, and dynamic VRPs, in which new information can come in during the route-planning process, or possibly even after the vehicles have set off (e.g., [7, 14, 16]). Dynamic VRPs tend to be much harder to solve than static ones, yet they have received less attention.

In this paper, we consider one specific dynamic VRP, which is a simplified version of a problem that we encountered in project with a large retail company. There is a fixed fleet of vehicles, and a limit on the duration of any single route. Customer requests arrive one at a time, via a website, from locations that are not known in advance. Whenever a request arrives, an automated decision must be made on whether or not to accept it. At

∗STOR-i Centre for Doctoral Training, Lancaster University, Lancaster LA1 4YR, UK. E-mail: M.Randall1@lancaster.ac.uk
†Department of Management Science, Lancaster University, Lancaster LA1 4YX, UK. E-mail: {A.Kheiri,A.N.Letchford}@lancaster.ac.uk
some point, no more requests can be accepted and the vehicle routes are fixed. After that, the vehicles are loaded and they depart. The goal is to maximise the number of satisfied requests. (A detailed formal description of the problem will be given in Section 2.)

We will see that our problem is an \(NP\)-hard, dynamic, stochastic, combinatorial optimisation problem. Moreover, in the real-life application, the decision on whether or not to accept any given request must be made within a few seconds. As a result, the use of exact optimisation methods is out of the question, and one must use a heuristic.

As long ago as 1971, it was realised that it is natural to use an insertion heuristic for dynamic VRPs of this kind [20]. In an insertion heuristic, we start with a collection of “empty” routes, and then iteratively attempt to insert each new customer request into one of the routes. When the routes are deemed to be “full”, or no more requests are expected, the procedure ends.

Insertion heuristics have been used for many different VRPs, including both static VRPs (e.g., [4,15,17,18]) and dynamic ones (e.g., [1,5,9–11,20]). We are not aware, however, of any systematic empirical comparison between different kinds of insertion heuristic. We are also not aware of any work on insertion heuristics for the particular dynamic VRP addressed here.

In this paper, we describe five different insertion heuristics for our dynamic VRP, which we call “sequential”, “quasi-sequential”, “naive parallel”, “seeded parallel” and “sum-of-squares parallel”. The last one, which is non-standard, inserts new requests in the position that minimises the sum of the squared route durations. We also consider the option of improving each of the five heuristics by performing local search whenever a request is accepted. Extensive computational results will show that the “sum-of-squares” heuristic performs remarkably well, even when local search is not used.

The paper has the following structure. In Section 2, we define our dynamic VRP formally and give an overview of the proposed solution approach. In Section 3, we describe our five insertion heuristics, and describe our local search procedure. In Section 4, we give some computational results. Finally, Section 5 contains some concluding remarks.

2 Problem Definition and Solution Framework

In this section, we define our problem formally and give an overview of our solution framework.

2.1 Problem definition

We now define our dynamic VRP in detail. We are given a compact metric space (e.g., a region of the Euclidean plane), in which there is located a single depot and a set of \(n\) customers. The location of the depot is known in
advance, but \( n \) is not known, and neither are the customer locations. There is a fixed fleet of \( m \) identical vehicles, all located at the depot.

There are no capacity constraints or time windows, but there is a limit on the duration of any given vehicle route. That is, we are given a positive integer \( D \), and the duration of each route must not exceed \( D \). (One can measure \( D \) in terms of distance, rather than time, if desired.)

As in Campbell & Savelsbergh [5], the planning horizon is divided into an ordering period and a delivery period. During the ordering period, customers make requests via a web site, and a heuristic is used to decide which requests to accept. It is only when a customer makes a request that the location of that customer is revealed. Later on, in the delivery period, the vehicles depart from the depot, fulfil the accepted orders, and return to the depot.

We remark that, in practical applications, there is typically a gap between the end of the ordering period and the start of the delivery period. For example, the ordering period might end at midnight, and the delivery period begin at 8am the following morning. During this interval, the orders are picked and loaded onto the vehicles.

We view the ordering period as consisting of \( n \) “decision epochs”, numbered from 1 to \( n \). At the start of epoch \( e \), the \( e \)th customer reveals their location and makes a request. A decision must then be made on whether or not to grant the request.

We assume that each customer is located at a random point in the given metric space. (We will consider two models of the population density, called uniform and exponential. These will be described in Subsection 4.1.) The primary objective is to maximise the expected number of customers served, but a secondary objective is to minimise the total expected travel time.

For brevity, we will call our problem the dynamic VRP with a deadline, or DVRPD for short. The DVRPD is clearly \( \mathcal{NP} \)-hard in the strong sense, via reduction from the standard TSP. Given the stochastic, dynamic and combinatorial nature of the problem, and the fact that decisions must be made quickly in practice, one must resort to a heuristic approach.

### 2.2 Insertion heuristic

As in [5, 9, 20], we use an insertion heuristic to construct and maintain a collection of “tentative” vehicle routes during the ordering period. When a new request comes in, the heuristic will check whether the request can be inserted into one of the tentative routes. If so, the request will be accepted.

To describe this approach in detail, we introduce some notation. For \( e = 1, \ldots, n \), we let \( A_e \) denote the set of customers whose requests have been accepted by the end of epoch \( e \). We also use the convention that \( A_0 = \emptyset \). In epoch \( e \), the insertion heuristic attempts to find a feasible way to insert customer \( e \) into one of current routes. If it is successful, the request of customer \( e \) is accepted, and \( A_e \) is set to \( A_{e-1} \cup \{e\} \). Otherwise, the request
is declined and $A_e$ is set to $A_{e-1}$.

Now, for $i = 1, \ldots, m$ and $e = 1, \ldots, n$, we let $C_{i,e}$ denote the set of customers that have been allocated to the $i$th vehicle at the end of epoch $e$. We also use the convention that $C_{i,0} = \emptyset$ for $i = 1, \ldots, m$. Note that

$$A_e = \bigcup_{i=1}^{m} C_{i,e} \quad (e = 0, \ldots, n).$$

In any given epoch, the current route for vehicle $i$ is stored as an ordered sequence of nodes:

$$R^{(i)} = \left(0, r^{(i)}_1, \ldots, r^{(i)}_{|C_{i,e}|}, 0\right)^T,$$

where $r^{(i)}_j \in \{1, \ldots, e\}$ represents the $j$th customer visited by vehicle $i$. Note that $r^{(i)}_0 = r^{(i)}_{|C_{i,e}|+1} = 0$ denotes the depot, since each route starts and ends there. At the start of the heuristic, the routes are initialised as $R^{(i)} = (0, 0)^T$. If the heuristic accepts the customer’s request during epoch $e$, it then selects a vehicle $i$, along with a position in that vehicle’s route, and $e$ is inserted into $R^{(i)}$ in the given position.

With this notation, the primary objective is to maximise the expected value of $|A_n|$, and the secondary objective is to minimise the expected value of

$$\sum_{i=1}^{m} \sum_{j=0}^{|C_{i,n}|} t\left(r^{(i)}_j, r^{(i)}_{j+1}\right),$$

where $t(j, j') = t(j', j)$ denotes the time taken to travel between customers $j$ and $j'$, and $t(0, j)$ denotes the time taken to travel between the depot and customer $j$. Moreover, the constraint on route duration amounts to imposing

$$\sum_{j=0}^{|C_{i,n}|} t\left(r^{(i)}_j, r^{(i)}_{j+1}\right) \leq D \quad (i = 1, \ldots, m).$$

### 2.3 Local search

In some applied settings, the average time interval between successive decision epochs may be large enough to permit the application of a local search procedure, immediately after an order has been accepted. The hope is that this will reduce the length of the “tentative” vehicle routes, which may allow more customer requests to be accepted later on.

To explore the potential of this idea, we consider two types of moves, called swaps and relocations. (These moves are similar to those used by Osman [13] for static VRPs.) The procedure for a swap is as follows. We select a pair of customers $j, j' \in A_e$ at random, and check if swapping their
current positions in the routes yields a feasible solution. If this is the case, we then check if the swap yields an improvement in the total travel time (or the sum of the squared durations). If so, we accept the move.

The procedure for a relocation is as follows. We select a customer \( j \in A_e \) at random. We then select a route at random, and a random position on any route. Following this, we check if relocating customer \( j \) to the new position yields a feasible solution. If this is the case, we then check if the relocation yields an improvement in the total time (or the sum of the squared durations). If so, we accept the move.

Since the time available to perform local search is typically limited in practice, we perform only \( 10e \) local search iterations at the end of decision epoch \( e \). In each iteration, we perform a swap or relocation at random, with equal probability.

3 Five Insertion Heuristics

In his seminal paper, Solomon [18] made a distinction between sequential insertion heuristics, in which routes are constructed one at a time, and parallel ones, in which several routes are constructed simultaneously. In this section, we describe a simple sequential insertion heuristic and four different parallel insertion heuristics.

3.1 Sequential insertion

In sequential insertion, we allocate as many customers as possible to vehicle \( i = 1 \), until we encounter a customer that cannot be inserted into the route of that vehicle (due to the limit on the route duration). From that point, we allocate as many customers as possible to vehicle \( i = 2 \), and so on. The process continues until (a) there are no more vehicles available or (b) the ordering period has ended.

Now, suppose that we have found that customer \( e \) can be inserted into the \( i \)th route. If there are several possible insertion points, we simply choose the point which leads to the smallest increase in the length of the route. More precisely, we search sequentially through \( R^{(i)} \) to find the position \( j \) which minimises

\[
t(r_j^{(i)}, e) + t(e, r_{j+1}^{(i)}) - t(r_j^{(i)}, r_{j+1}^{(i)}).
\]

We remark that, although sequential insertion is a very simplistic heuristic, it may be necessary in real-world applications where orders can arrive even after some of the vehicles have been sent out.

3.2 Quasi-sequential insertion

Observe that it could happen that the current customer cannot be inserted into route \( i \), but a later customer can be. (This could happen if the current
customer is far away from route \(i\), but a later customer is closer.) This leads us to consider a modified version of sequential insertion, which we call quasi-sequential.

The idea is as follows. If it is possible to insert a customer into the first route, we do so. Otherwise, we check whether the customer can be inserted into the second route. If it is possible, we do so. And so on. The process continues until the end of the ordering period. If there are several possible insertion points in any given epoch, we again choose the point which leads to the smallest increase in the duration of the given route.

3.3 Naive parallel insertion

Our third insertion rule is called naive parallel insertion. Here, we attempt to construct routes for all of the vehicles in parallel. More precisely, whenever a new customer requests a delivery, we check all possible insertion positions in all of the \(m\) routes. If a feasible insertion point exists, the request is accepted. If there are several feasible insertion points, we choose the point which leads to the smallest increase in the total duration of the routes.

We call this rule naive for the following reason. In practice, we found that the naive insertion heuristic behaves in a more-or-less sequential manner. To see why, suppose that we have just inserted the first customer into the first route, and we are now considering where to insert the second customer. Due to the triangle inequality, it will almost always be cheaper to insert the second customer into the first route rather than into one of the other routes. This in turn will make it more likely that the third customer will be inserted into the first route as well. As a result, the routes tend to “fill up” one after another, and the resulting solution tends to be very similar to the one obtained with quasi-sequential insertion.

We will comment on this phenomenon again in Section 4.2.

3.4 Parallel insertion with seeds

Our fourth insertion rule is called parallel insertion with seeds. In this method, we “seed” the routes, in an attempt to prevent the behaviour mentioned in the previous subsection. More precisely, the first \(m\) accepted orders are allocated to different routes. From that point on, we proceed as in naive parallel insertion.

We remark that some care is needed when seeded insertion is used in combination with local search. At the end of any given decision epoch, the application of local search can result in one of the routes being emptied, which effectively “un-seeds” the route. To address this, we simply insert the next customer into the empty route, thereby “re-seeding” it.
3.5 Sum-of-squares insertion

Our fifth and final rule is inspired by the work of Bektaş and Letchford [3]. They observed that, in optimal solutions to (static) VRPs, some routes are often much longer than others. Such solutions will be perceived as unfair by the drivers. To address this, they proposed to minimise the sum of the squared route lengths rather than the sum of the route durations.

Although [3] was concerned with static VRPs, one can apply the same concept to dynamic VRPs. To do this, we modify the naive insertion rule in the following way. If there are several feasible insertion points for a given customer \( e \), we choose the point which leads to the smallest increase in the sum of the squared route durations. That is, we check all vehicles \( i \) and positions \( j \) sequentially for that which minimises

\[
(T_{i,e-1} + t(e, r_j^{(i)}) + t(e, r_{j+1}^{(i)}) - t(r_j^{(i)}, r_{j+1}^{(i)}))^2 - T_{i,e-1},
\]

where

\[
T_{i,e-1} = \sum_{k=0}^{\binom{n-1}{m-1}} t(r_k^{(i)}, r_{k+1}^{(i)}),
\]

is the total travel time of route \( i \) at the end of epoch \( e - 1 \).

We call this last insertion rule sum-of-squares (SoS) insertion. We will see in the next section that the SoS insertion performs remarkably well.

4 Computational Results

In this section, we present some extensive computational results. Subsection 4.1 explains how we created our test instances. Subsection 4.2 presents the results that we obtained without local search, and Subsection 4.3 presents the results that we obtained when local search was added.

All of our heuristics were coded in Python, and all experiments were run on an Intel Xeon® Gold 6248R processor with 128GB of RAM, running at 3.00 GHz under Ubuntu 22.04.1 LTS. The Python routines have been made freely available on GitHub.

4.1 Test instances

As far as we know, there are no benchmark instances available for the dynamic VRP under consideration. Accordingly, we created some instances of our own, as follows. The number of vehicles \( m \) is set to 5 and the number of customers \( n \) is set to 100. We assume that the depot and customers are located in a circle of radius 30km in the Euclidean plane. All vehicles are assumed to travel at 30km/hr. We then consider three different parameters, each with two values:

\[https://github.com/ahmedkheiri/dynamic-insertion\]
• Duration limit ($D$): this is set to either 2 hours or 4 hours.

• Depot location (DL): the depot is placed either at the centre of the circle (Cn), or on its boundary (Bd).

• Population density model (Pop): this can be “uniform” (Uni) or “exponential” (Exp). “Uniform” means that the customers are located uniformly at random within the circle, and “exponential” means that the population density decreases exponentially as one moves away from the centre of the circle, as explained below.

This gives $2^3 = 8$ different instance types. We generated 1000 random sets of customer locations each for the uniform and exponential population density models. We then used these sets of locations, with the appropriate depot locations and duration limits, to create 1000 instances for each of the 8 instance types.

Our “Exp” instances are based on a slight modification of a population density model which is commonly used for urban populations (e.g., [6,12]). The model is

$$f(d) = \lambda \exp(-\gamma d),$$

where $f(d)$ is the population density function, $d$ is the distance from the city centre, $\lambda$ is the population density at the city centre, and $\gamma$ is a constant called the density gradient. To adapt this model to our context, we (a) assume that the city centre is at the centre of the circle, (b) replace the distance $d$ with the travel time, measured in hours, and (c) set the population density to zero for points that lie outside the circle. We set $\gamma$ to $\ln(10) \approx 2.303$, which means that the density at the center of the circle is ten times the density at the boundary of the circle. Rather than setting $\lambda$ in advance, we generate the customers using rejection sampling, and stop as soon as the number of customers reaches 100.

4.2 Results with insertion alone

Table 1 reports the average number of customers served, for each instance type and each of the five insertion heuristics. Each figure is the mean over the 1000 instances. The largest value for each setting is in bold font.

We see that the sequential insertion heuristic performs very poorly. The quasi-sequential and naive parallel heuristics are a lot better, and they find solutions of very similar quality. This is due to the phenomenon mentioned in Subsection 3.3. The seeded heuristic improves on this further. Finally, the SoS heuristic performs very well in all cases. It is slightly worse than the seeded heuristic when $D = 4$ and the depot is on the boundary, but it beats all other heuristics in all other cases.

As one would expect, setting the time deadline to four hours rather than two enables one to serve more customers. It is also apparent from the table
that one can serve more customers when the depot is at the centre of the circle rather than on the boundary. This is simply because, in that case, more customers tend to be located close to the depot. When the population density is exponential, the effect is even more marked. Apart from that, we see no obvious patterns.

In order to gain more insight into the behaviour of the five insertion rules, we computed some more statistics for the instance type “4-Cn-Exp”. First, for each of the five rules, and for each epoch of the ordering period, we computed the mean number of vehicles in use (i.e., the mean number of vehicles that had at least one customer assigned to them). Second, for each of the five rules, and for each epoch, we computed the mean total travel time. Figures 1 and 2 show the results, where each curve corresponds to a different insertion rule.

The curve in the middle of Figure 1, which is green in the online version

Table 1: Average number of customers served (to 3 dp) for the eight instance types. The best value for each type is given in bold.

<table>
<thead>
<tr>
<th>D</th>
<th>DL</th>
<th>Pop</th>
<th>Seq</th>
<th>QSeq</th>
<th>Naive</th>
<th>Seed</th>
<th>SoS</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>Cn</td>
<td>Uni</td>
<td>7.132</td>
<td>35.302</td>
<td>37.407</td>
<td>37.790</td>
<td>38.712</td>
</tr>
<tr>
<td>2</td>
<td>Cn</td>
<td>Exp</td>
<td>9.777</td>
<td>44.858</td>
<td>48.866</td>
<td>50.758</td>
<td>52.225</td>
</tr>
<tr>
<td>2</td>
<td>Bd</td>
<td>Uni</td>
<td>9.689</td>
<td>29.162</td>
<td>30.302</td>
<td>30.374</td>
<td>30.920</td>
</tr>
<tr>
<td>4</td>
<td>Cn</td>
<td>Uni</td>
<td>24.961</td>
<td>66.824</td>
<td>72.734</td>
<td>88.812</td>
<td>98.241</td>
</tr>
<tr>
<td>4</td>
<td>Cn</td>
<td>Exp</td>
<td>35.453</td>
<td>75.502</td>
<td>82.013</td>
<td>95.208</td>
<td>99.877</td>
</tr>
<tr>
<td>4</td>
<td>Bd</td>
<td>Uni</td>
<td>14.912</td>
<td>57.378</td>
<td>64.602</td>
<td>67.315</td>
<td>66.747</td>
</tr>
<tr>
<td>4</td>
<td>Bd</td>
<td>Exp</td>
<td>21.533</td>
<td>63.106</td>
<td>71.387</td>
<td>74.936</td>
<td>73.597</td>
</tr>
</tbody>
</table>

Figure 1: Mean number of vehicles in use at each epoch with various insertion methods for instance type “4-Cn-Exp”.
of this paper, corresponds to the sequential insertion rule. It fills up the routes one at a time, but has no ability to add more orders to a route once it has moved on. This can result in calling new vehicles into use somewhat prematurely. The two curves below (black and red) represent the quasi-sequential and naive parallel insertion rules. We see that the two methods give almost identical results, as we expected. The next curve up (orange) represents the SoS insertion rule. It is apparent that more vehicles are in use throughout the ordering period than with the previously mentioned rules. Finally, the curve at the top (blue) corresponds to the seeded parallel insertion rule. It inserts the first $m$ customers into different vehicles, as it is designed to.

As for Figure 2, we observe that throughout the ordering period, heuristics which are designed to be “more parallel” tend to have a lower total travel time. (A minor exception to this is that the sequential heuristic gives slightly lower travel times than some of the other heuristics in the later epochs, but this is due to serving far fewer customers.) More importantly, we see that considerable time savings can be made using the SoS heuristic, which is impressive, given that the SoS heuristic serves more customers for this instance type (see again Table 1).

It remains to be explained why the SoS heuristic performs so well. A possible explanation follows. By penalising long routes, the SoS rule causes the routes to have similar lengths throughout the ordering period. As a result, none of the routes are likely to be completely full until near the end of the ordering period. Thus, for all customers except the last few, there tend to be many feasible insertion positions. This increased number of options allows for more efficient insertion of the later customers, which tends to outweigh the slightly less efficient insertion of the earliest customers.
We remark that we also tested the five insertion heuristics on some other kinds of instances, with very similar results. We omit details for brevity.

4.3 Results with local search

The results obtained with local search are summarised in Tables 2 and 3. These tables have the same format as Table 1. Table 2 corresponds to the case in which the local search uses the regular objective function of minimising the total travel time, whereas Table 3 corresponds to the case in which the local search uses the SoS objective.

First we compare Table 2 with Table 1. We see that the addition of local search improves the average number of served customers in all cases. We also see that with local search, the gap in performance between all heuristics other than the sequential one is significantly narrowed. The combination of SoS insertion with local search seems to be particularly useful when the depot lies in the centre.

Next we compare Tables 2 and 3. We see that the use of the SoS criterion within the local search leads to a significant improvement in almost all cases. The exception to this is when the sequential heuristic is being used, but this is to be expected, since the SoS approach was specifically designed with parallel insertion in mind.

Finally, we compare the last columns of Tables 1 and 3. We see that SoS insertion already performs pretty well in most cases, with the result that the addition of (SoS-based) local search yields less benefit than one might expect. The exception is when \( D = 4 \) and the depot is on the boundary, where local search helps quite a bit. Given that local search is harder to implement than insertion, using SoS insertion alone may be an attractive alternative in some applications (especially if the average time between consecutive customer requests is small).

Table 2: Average number of customers served when performing local search aimed at minimising total travel time. The best value for each instance type is given in bold.

<table>
<thead>
<tr>
<th>( D )</th>
<th>DL</th>
<th>Pop</th>
<th>Seq</th>
<th>QSeq</th>
<th>Naive</th>
<th>Seed</th>
<th>SoS</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>Cn</td>
<td>Exp</td>
<td>11.256</td>
<td><strong>53.998</strong></td>
<td>53.910</td>
<td>53.941</td>
<td>53.968</td>
</tr>
<tr>
<td>2</td>
<td>Bd</td>
<td>Uni</td>
<td>11.902</td>
<td>32.200</td>
<td>32.230</td>
<td>32.198</td>
<td>32.194</td>
</tr>
<tr>
<td>4</td>
<td>Cn</td>
<td>Uni</td>
<td>50.886</td>
<td>98.484</td>
<td>98.803</td>
<td>98.843</td>
<td><strong>99.157</strong></td>
</tr>
<tr>
<td>4</td>
<td>Cn</td>
<td>Exp</td>
<td>71.135</td>
<td>99.807</td>
<td>99.875</td>
<td>99.882</td>
<td><strong>99.922</strong></td>
</tr>
<tr>
<td>4</td>
<td>Bd</td>
<td>Uni</td>
<td>20.127</td>
<td>79.580</td>
<td>79.576</td>
<td><strong>79.894</strong></td>
<td>79.799</td>
</tr>
<tr>
<td>4</td>
<td>Bd</td>
<td>Exp</td>
<td>30.303</td>
<td>88.990</td>
<td><strong>89.484</strong></td>
<td>89.259</td>
<td>89.285</td>
</tr>
</tbody>
</table>
Table 3: Average number of customers served when performing local search aimed at minimising the sum of squared travel times. The best value for each instance type is given in bold.

<table>
<thead>
<tr>
<th>D</th>
<th>DL</th>
<th>Pop</th>
<th>Seq</th>
<th>QSeq</th>
<th>Naive</th>
<th>Seed</th>
<th>SoS</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>Cn</td>
<td>Uni</td>
<td>7.501</td>
<td></td>
<td>99.765</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Cn</td>
<td>Exp</td>
<td>10.738</td>
<td>55.636</td>
<td>55.170</td>
<td>55.300</td>
<td>55.268</td>
</tr>
<tr>
<td>2</td>
<td>Bd</td>
<td>Uni</td>
<td>11.858</td>
<td>32.309</td>
<td>32.305</td>
<td>32.324</td>
<td>32.315</td>
</tr>
<tr>
<td>4</td>
<td>Cn</td>
<td>Uni</td>
<td>44.109</td>
<td>99.099</td>
<td>99.696</td>
<td>99.700</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Cn</td>
<td>Exp</td>
<td>63.129</td>
<td>99.887</td>
<td>99.984</td>
<td>99.982</td>
<td>99.975</td>
</tr>
<tr>
<td>4</td>
<td>Bd</td>
<td>Uni</td>
<td>19.457</td>
<td>80.064</td>
<td>80.295</td>
<td>80.291</td>
<td>80.351</td>
</tr>
<tr>
<td>4</td>
<td>Bd</td>
<td>Exp</td>
<td>29.635</td>
<td>89.595</td>
<td>90.029</td>
<td>90.011</td>
<td>90.045</td>
</tr>
</tbody>
</table>

5 Conclusions

We tested five different insertion heuristics for dynamic VRPs, both with and without local search. The main conclusions are (a) if a parallel insertion heuristic is implemented in a naive way, it is likely to perform as poorly as the sequential version; (b) good results are obtained by inserting customers in the position that minimises the sum of the squared route durations; (c) performing local search between customer requests usually enables one to serve more customers, particularly when the local search itself aims to minimise the sum of squared route durations; (d) SoS insertion alone can be an attractive alternative.

An interesting topic for future research is whether the “sum-of-squares” insertion heuristic can be adapted to more complex dynamic VRPs, such as ones in which each prospective customer must be offered a selection of possible time windows (see [5]). We hope to address this in a future paper.

Acknowledgements: The first author gratefully acknowledges financial support from the EPSRC (through the STOR-i Centre for Doctoral Training under grant EP/S022252/1), and from Tesco PLC.

References


