

A Fast Savings Algorithm for Solving Large-Scale Euclidean CVRPs

Kenji Kanazawa(✉)

University of Tsukuba, 1-1-1 Tennodai, Tsukuba, Japan
kanazawa@cs.tsukuba.ac.jp

Abstract. Clarke and Wright's savings algorithm (CW algorithm) is one of the most widely used heuristic methods for solving the capacitated vehicle routing problem (CVRP). Given a scenario with n customers and a depot, the CW algorithm initially generates n distinct delivery routes, each connecting a single customer to the depot. It then iteratively merges the two routes based on their saving values under the constraint that the total load of items delivered on the new route does not exceed the vehicle capacity. Herein, the saving value represents the distance reduction achieved merging two routes in to one by linking two customers from different routes. This involves generating $n \times (n-1)/2$ saving values and sorting them in descending order in order to merge the routes with larger saving values preferentially. However, it leads to the overall time complexity of $\mathcal{O}(n^2 \log n)$ when comparison-based sorting algorithms such as the merge sort are employed. In this paper, we propose a variant of the CW algorithm for solving large-scale Euclidean CVRPs achieving a best-case time complexity of $\mathcal{O}(n^2)$. Our proposed approach calculates the saving values using approximate Euclidean distances derived from only integer arithmetic, then sorts the obtained saving values by distribution-based sorting algorithms that can arrange integer values in linear time. Experimental results show that our proposed approach handles large-scale problems involving $3,000$ to $30,000$ customer nodes in 2.88 s on average while preserving the solution quality, which corresponds to 6.0× speedup over the original CW algorithm.

Keywords: capacitated vehicle routing problem · savings algorithm · approximate norm · distribution-based sorting algorithms

1 Introduction

The Capacitated Vehicle Routing Problem (CVRP) is a well-known NP-hard combinatorial optimization problem, which searches for a set of delivery routes starting and ending at a depot that cover all customers with multiple capacity-constrained vehicles. Its goal is to minimize the total distance of the vehicles traveled while ensuring that the total load of delivery items along each route does not exceed the capacity of vehicles.

CVRP instances are typically represented as undirected complete graphs, where nodes represent customers and a depot, and edge weights correspond to the distances between nodes.

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2026
O. Gervasi et al. (Eds.): ICCSA 2025 Workshops, LNCS 15888, pp. 59–74, 2026.
https://doi.org/10.1007/978-3-031-97596-7_4

Clarke and Wright's savings algorithm (CW algorithm) [1] is one of the most representative heuristic algorithms for solving CVRP owing to its simplicity and practicality. Given n customer nodes, the CW algorithm initially generates n distinct delivery routes, each linking the depot to a single customer. It then iteratively selects two distinct delivery routes to merge to a single route, guided by *saving* value, i.e., the distance reduction obtained by merging the two routes into one by linking two customer nodes that belong to different routes to each other, without violating the vehicle's capacity restriction.

The CW algorithm involves deriving the saving values for all possible pairs of the initial n routes and sorting the saving values in descending order so that the algorithm preferentially selects route pairs with greater distance reductions for merging. The sorting process introduces a computational bottleneck because it is forced to sort $\mathcal{O}(n^2)$ saving values. Consequently, the overall time complexity is $\mathcal{O}(n^2 \log n)$ when using standard comparison-based sorting algorithms such as merge sort and quick sort. Although the CW algorithm is relatively efficient among approaches for solving CVRP, its execution time rapidly grows as n increases.

In this paper, we propose a variant of the CW algorithm for efficient solving Euclidean CVRP where the distances adhere to the Euclidean metric. Our proposed approach calculates the saving values from an approximate Euclidean distance introduced by Chaudhuri et al. [2] using only integer arithmetic. It then sorts the saving values using distribution-based sorting algorithms such as counting sort [3] or pigeonhole sort [4], which efficiently sort elements with integer key values in linear time. By doing so, our proposed approach achieves a best-case time complexity of $\mathcal{O}(n^2)$. Experimental results show that our proposed approach obtains feasible solutions for large-scale problem instances involving $3,000$ to $30,000$ customers in 2.88 s on average while preserving the solution quality, which corresponds to $6.0\times$ speedup over the original CW algorithm.

2 Clarke and Wright's Savings Algorithm

Algorithm 1 illustrates the CW algorithm. Given a CVRP instance consisting of a triple (V, D, and q), where V represents a set of nodes comprising n customers and a depot, D denotes the demand set, indicating the load of items to deliver for each customer, and q describes the capacity of each vehicle, the algorithm proceeds as follows:

1. It generates an empty set S and an initial route set R_{init} composed of n separate routes, each linking a single customer to the depot, and sets R to R_{init} (Line 6 in Algorithm 1).
2. It then calculates the saving values s_{ij} (Lines 9−17). Herein, s_{ij} denotes the reduction in Euclidean distance achieved by merging two distinct routes linking customers i and j. In R_{init}, Euclidean distance to each customer i and j is calculated as $2 \cdot c_{0i}$ and $2 \cdot c_{0j}$, respectively, where c_{0i} represents the Euclidean distance between the depot (denoted as 0) and customer i. This is because each vehicle must travel back and forth the between the depot

and a customer. For a newly formed route generated by linking i and j, the total Euclidean distance is calculated as $c_{0i} + c_{ij} + c_{0j}$, yielding $s_{ij} = 2 \cdot c_{0i} + 2 \cdot c_{0j} - (c_{0i} + c_{0j} + c_{ij}) = c_{0i} + c_{0j} - c_{ij}$. Note that $s_{ij} > 0$ indicates a reduction in the distance. s_{ij} along with (i, j) is stored to a set S.

3. It sorts (s_{ij}, i, j) in S in descending order of s_{ij} (Line 18).
4. It iteratively executes the following steps over the elements in S from the head to the tail (Line 20):
   (a) For s_{ij} under consideration, two separate routes in R are merged by linking i and j subject to the following conditions:
      – Total load of delivery items along the two routes does not exceed q.
      – Neither i nor j is *interior* to its current route (i.e., at least one of them must be adjacent to the depot in the traversal order).
   (b) If S has not been exhausted, then it goes back to Step 4a to process the next element in S. Otherwise, it returns R at that time and terminates.

Algorithm 1. Procedure of CW algorithm

```
 1: function CW(V, D, q)
 2: // V: node set of a depot (denoted as 0) and n customers (denoted as 1, ..., n)
 3: // D: demand set
 4: // q: capacity of each vehicle
 5:
 6:     R = R_init = GenInitialRouteSet(V);
 7:
 8:     S = ∅; /* S: set of the saving values along with the corresponding node indices */
 9:     for all (i, j) do /* i, j ∈ V, i ≠ j. */
10:         c_ij = EuclideanDistance(i, j);
11:         c_0i = EuclideanDistance(0, i);
12:         c_0j = EuclideanDistance(0, j);
13:         s_ij = c_0i + c_0j − c_ij;
14:         S = S ∪ {(s_ij, i, j)};
15:         //c_ij: distance between i and j.
16:         //s_ij: saving value when two distinct routes are merged by linking customers i and j.
17:     end for
18:     S = Sort_inDescendingOrder(S);
19:
20:     R = ConstructRoute(R, S, D, q);
21:     return R;
22: end function
```

In terms of time complexity, Step 1 takes $\mathcal{O}(n)$, while Steps 2 and 4 handle $O(n^2)$ saving values, leading to the time complexity of $\mathcal{O}(n^2)$ for both. Sorting $\mathcal{O}(n^2)$ saving values in Step 3 consumes $\mathcal{O}(n^2 \log n)$. Hence, the overall time complexity is $\mathcal{O}(n^2 \log n)$.

3 Literature Review

Many approaches for improving CW algorithm have been proposed till date [5–7,10,12,13]. Among them, literatures [5,6], and [7] mainly intended to

improve the accuracy of the original CW algorithm but they did not evaluate their approach using large-scale CVRP instances (with at least 3,000 customer nodes). On the other hand, literatures [10,12,13] mainly aimed at reducing the time complexity to treat large-scale CVRP instances. Hereinafter, we focus on the latter ones.

In [10], a modified version of CW algorithm named CW^{100} was described. This approach reduced memory consumption and execution time of the original CW algorithm by constructing a solution only considering saving values between each customer and its 100 nearest nodes. Implemented in JAVA on an AMD Ryzen-3 1300X running at 3.5 GHz with 4 GB of main memory, CW^{100} reportedly required 3 to 34 s to solve instances ranging from 3,000 to 12,000 customer nodes, including post-optimization via the Lin-Kernighan (LK) heuristic.

Authors in [12,13] first presented a GPU-accelerated CW algorithm exploiting CUDA cores in [12], and then proposed more sophisticated approach in [13] to optimize the distance and saving value calculations by parallel processing on RTX-4090 GPU. In [13], they also introduced a "tabu list" of the saving values to exclude the saving values that became superfluous for solution construction. As a result, they achieved an average and peak speedups of 9.7× and 48×, respectively, compared with a Python-based sequential implementation running on a Core-i9 14900HX CPU operating at 2.2 GHz with 64 GB main memory [13]. However, their approach did not address the inefficiencies in sorting the saving values. This limitation left the method inefficient for large-scale instances, as solving a problem with 30,000 customer nodes still required 183 s [13].

4 Our Approach

As discussed previously, the most computationally intensive step is sorting the saving values, which requires $\mathcal{O}(n^2 \log n)$ time when using standard comparison-based sorting algorithms. To further enhance efficiency, our approach employs the approximate Euclidean distance proposed by Chaudhuri et al. [2] so that it can calculate the saving values via only integer arithmetic. This enables the sorting of the saving values through distribution-based sorting algorithms, specifically counting sort [3] and pigeonhole sort [4], which can operate in linear time for integer sorting. Given m elements with integer key values ranging from 0 to $k-1$, both counting sort and pigeonhole sort exhibit a worst case time complexity of $\mathcal{O}(m+k)$ [3] [4]. Consequently, by employing these sorting methods for sorting the saving values, the overall time complexity of the CW algorithm can be reduced to $\mathcal{O}(n^2)$ when the range of the saving values is sufficiently smaller than n^2, as discussed in Sect. 5.

The following subsections will first outline the principles of counting sort and pigeonhole sort for integer key sorting. Following that, we detail how to apply the approximate Euclidean distance by Chaudhuri et al. [2] to the CW algorithm to adopt these sorting techniques, and finally describe our modified CW algorithm.

4.1 Distribution-Based Sorting Algorithm

Counting Sort. Counting sort [3] is a sorting algorithm that sorts integers by counting their occurrences.

Algorithm 2 demonstrates the process of counting sort for sorting integers ranging from 0 to $k-1$ in *descending* order. Here, 'A' represents an array of given integers ranging from 0 to $k-1$. Algorithm 2 initially sets $cnt[i]$ to 0 ($i = 0$ to k), then increments $cnt[i]$ whenever $A[j] = i$ ($j = 0$ to $|A|-1$) to count the occurrences of $i \in A$ (Lines 6–8). Next, it takes the running sum of $cnt[\ ]$ from the tail to head (Lines 9–11). At this point, $cnt[A[j]]$ indicates the last position of $A[j]$ in the sorted array. Then, it iteratively places $A[j]$ to $B[cnt[A[j]]$ decrementing $c[A[j]]$ (Lines 12–15). This process ensures that elements with the equivalent value are placed in the array B in series without missing or exceeding. Finally it returns B.

Algorithm 2. Counting sort for sorting in descending order

```
1: function CountingSort(A, k)
2: // A: array of integer key values.
3:     for i = 0 to k do
4:         cnt[i] = 0; /* cnt[i]: occurrences of values equal to or greater than i. */
5:     end for
6:     for j = 0 to |A| − 1 do
7:         cnt[A[j]]++;
8:     end for
9:     for i = k − 1 to 0 do
10:        cnt[i] += cnt[i + 1];
11:    end for
12:    for j = |A| − 1 to 0 do
13:        B[cnt[A[j]]] = A[j];
14:        cnt[A[j]]−−;
15:    end for
16:    return B;
17: end function
```

Pigeonhole Sort. Pigeonhole sort [4] is a sorting algorithm that prepares for an array of "pigeonholes" and arranges the order of given integers by placing each integer into the pigeonhole whose order corresponds to the placed integer value.

Algorithm 3 demonstrates the process of pigeonhole sort for sorting integers ranging from 0 to $k-1$ in descending order. The definitions of A and k are the same as in Algorithm 2, while $occ[i]$ represents the number of occurrences of $i \in A$. The algorithm starts by initializing $occ[i]$ to 0 (Lines 3–5). It then places each element $A[j]$ into the pigeonhole at position $[A[j]][occ[A[j]]]$ (denoted as $pigeonhole[A[j]][occ[A[j]]]$) and increments $occ[A[j]]$ (Lines 6–9). Hence, $pigeonhole[i][\ ]$ gathers all integers in A that have the value i. Finally, the algorithm retrieves the integers from the pigeonholes by scanning from the tail to the top. It stores them to output array B from the first position to the last (Lines 11–17), and returns B.

Algorithm 3. Pigeonhole sort for sorting in descending order

```
1: function PigeonholeSort(A,k)
2: // A: array of integer key values.
3:     for i = 0 to k − 1 do
4:         occ[i] = 0;
5:     end for
6:     for j = 0 to |A| − 1 do
7:         pigeonhole[A[j]][occ[A[j]]] = A[j];
8:         occ[A[j]]++;
9:     end for
10:
11:    i_B = 0;
12:    for i = k − 1 to 0 do
13:        for j = 0 to occ[i] do
14:            B[i_B] = pigeonhole[i][j];
15:            i_B++;
16:        end for
17:    end for
18:    return B;
19: end function
```

4.2 Calculating the Saving Values in Integer Arithmetic Based on Approximate Euclidean Distance by Chaudhuri et al [2]

Both Algorithms 2 and 3, are designed to work with non-negative integer key values. However, in Euclidean CVRP, the saving values could be real numbers and might include negative values. Additionally, these methods require prior knowledge of the key value range. To address these constraints, the saving values are converted into integers by calculating and rounding them while simultaneously determining their maximum and minimum values. Once the minimum is identified, all the saving values can be offset by this minimum.

There are several possible options for rounding:

(1) Round coordinates of the customer and the depot nodes immediately on reading the CVRP instance. All subsequent process are then performed using integer operations.
(2) Input the CVRP node coordinates as floating-point values, calculate the Euclidean distances and round them to integers, and subsequently derive the saving values from these integer distances.
(3) Perform all calculations to derive the saving values in floating-point starting from reading the instance and ending with obtaining the saving values, and then round the saving values to integers.

Although the method (1) is potentially the least computationally intensive, it requires an alternative way to calculate Euclidean distances using only integer arithmetic. For this purpose, our approach adopts the approximate norm proposed by Chaudhuri et al. [2].

Chaudhuri et al. [2] introduced a method to approximate the L_n norm by combining L_∞ and L_1 norms. Formula (1) presents this approximation for two

n-dimensional vectors, X and Y.

$$L_n = |x_{i_{max}} - y_{i_{max}}| + \frac{1}{n - \lceil \frac{n-2}{2} \rceil} \cdot \sum_{i=0, i \neq i_{max}}^{n-1} |x_i - y_i|$$

$$= |x_{i_{max}} - y_{i_{max}}| + \frac{1}{n - \lceil \frac{n-2}{2} \rceil} \cdot (\sum_{i=0}^{n-1} |x_i - y_i| - |x_{i_{max}} - y_{i_{max}}|) \quad (1)$$

Herein, x_i and y_i denote the i-th elements of X and Y, respectively, while i_{max} represents the index of the largest component of $|x_i - y_i|$, i.e., $i_{max} = \underset{0 \leq i \leq n-1}{\text{argmax}}(|x_i - y_i|)$. Recognizing that $|x_{i_{max}} - y_{i_{max}}|$ and $\sum_{k=i}^{n-1} |x_i - y_i|$ correspond to L_∞ and L_1 norms of X and Y, respectively, Formula (1) can be rewritten as follows:

$$L_n = L_\infty + \frac{1}{n - \lceil \frac{n-2}{2} \rceil} \cdot (L_1 - L_\infty) \quad (2)$$

Substituting $n = 2$ in Formula (2), we obtain L_2, i.e., the approximate Euclidean distance as follows:

$$L_2 = \frac{1}{2}(L_\infty + L_1) \quad (3)$$

Given X and Y as integers, Formula (3) can be calculated with only integer operations. Specifically, L_∞ is derived through two subtractions and absolute value calculations in integers, followed by a single multiplexing that selects the element with larger value from $|x_i - y_i|$ $(i = 0, 1)$, while L_1 is computed through two integer subtractions and absolute calculations, and a single integer addition. $(L_\infty + L_1)/2$ is then acquired by addition of two integers L_∞ and L_1 and applying a right shift operation, which is equivalent to the division by 2. In this way, we can directly derive approximate Euclidean distance between two distinct nodes in a given CVRP instance using only integer arithmetic.

4.3 Procedure of Our Modified CW Algorithm

Algorithm 4 presents our modified version of the CW algorithm. Its procedure is basically same as Algorithm 1, except that the saving values are treated as integers as already mentioned. In Algorithm 4, s_{min} and s_{max} denote the minimum and maximum values of s_{ij}, respectively. **GenInitialRouteSet()** and **ConstructRoute()** are the same functions as those in Algorithm 1. The remaining functions work as follows:

- **Round()**: Rounds the coordinate data to the nearest integers.
- **ApproximateEuclideanDistance()**: Calculates approximate Euclidean distances based on the above-mentioned description.
- **Sort_inDescendingOrder()**: Sorts the saving values along with their corresponding customer indices in descending order of the saving values (using counting sort or pigeonhole sort). Unlike Algorithm 1, this function accepts s_{min} and s_{max} as inputs to determine the valid range of the saving values and also offsets them based on s_{min}.

Algorithm 4. Procedure of modified CW algorithm

```
1: function CW(V, D, q)
2:     s_min = inf;
3:     s_max = − inf;
4:     V = Round(V);
5:     R = R_init = GenInitialRouteSet(V);
6:
7:     S = ∅; /* S: set of the saving values along with the corresponding node indices */
8:     for all (i, j) do /* i, j ∈ V, i ≠ j. */
9:         c_ij = ApproximateEuclideanDistance(i, j);
10:        c_0i = ApproximateEuclideanDistance(0, i);
11:        c_0j = ApproximateEuclideanDistance(0, j);
12:        s_ij = c_ij − c_0i − c_0j;
13:        S = S ∪ {(s_ij, i, j)};
14:        if (s_max < s_ij) then
15:            s_max = s_ij;
16:        end if
17:        if (s_min > s_ij) then
18:            s_min = s_ij;
19:        end if
20:    end for
21:    S = Sort_inDescendingOrder(S, s_max, s_min); /* Counting sort or Pigeonhole sort */
22:
23:    R = ConstructRoute(R, S, D, q);
24:    return R;
25: end function
```

5 Evaluation

To assess our proposed approach, we evaluate the following four different variants of CW algorithm based on without/with discretizing the saving values and the sorting algorithms:

1. The original CW algorithm, in which the saving values are not discretized (denoted as CW_0).
2. A variant in which the saving values are discretized, and sorted by merge sort (CW_1).
3. A variant in which the saving values are discretized and sorted by counting sort ($OURS_c$).
4. A variant in which the saving values are discretized and sorted by pigeonhole sort ($OURS_p$).

Discretizing the savings values are performed based on the approximate Euclidean distance by Chaudhuri et al. [2] as described in Sect. 4. All tested codes are written in C language, compiled by gcc with -O3 optimization option, and executed on a single thread of AMD Threadripper 3970X operating at 3.7 GHz with 64 GB of main memory.

We evaluate above-mentioned four implementations using Euclidean CVRP instances ranging from 500 to 1,000 and 3,000 to 30,000 customer nodes drawn from two distinct benchmark sets in CVRPLIB [8], i.e., Uchoa et al. [9] and Arnold et al. [10] benchmark sets. Hereinafter, we denote the former as *Set_X* and latter as *Set_XXL*.

Table 1. Range and total count of the saving values (Set_X)

Instance	number of customers	m (number of saving values)	$s_{min} \sim s_{max}$	$(s_{max} - s_{min} + 1)/m$
X-n502-k39	501	125,250	$0 \sim 2365$	1.89×10^{-2}
X-n513-k21	512	130,816	$-2 \sim 1375$	1.05×10^{-2}
X-n524-k153	523	136,503	$-2 \sim 2061$	1.51×10^{-2}
X-n536-k96	535	142,845	$-2 \sim 1428$	1.00×10^{-2}
X-n548-k50	547	149,331	$0 \sim 2928$	1.96×10^{-2}
X-n561-k42	560	156,520	$-2 \sim 1449$	9.26×10^{-3}
X-n573-k30	572	163,306	$0 \sim 2356$	1.44×10^{-2}
X-n586-k159	585	170,820	$-1 \sim 2528$	1.48×10^{-2}
X-n599-k92	598	178,503	$-2 \sim 2206$	1.24×10^{-2}
X-n613-k62	612	186,966	$-2 \sim 1462$	7.82×10^{-3}
X-n627-k43	626	195,625	$0 \sim 2973$	1.52×10^{-2}
X-n641-k35	640	204,480	$0 \sim 2800$	1.37×10^{-2}
X-n655-k131	654	213,531	$-1 \sim 1333$	6.24×10^{-3}
X-n670-k130	669	223,446	$-2 \sim 2047$	9.16×10^{-3}
X-n685-k75	684	233,586	$-2 \sim 1425$	6.10×10^{-3}
X-n701-k44	700	244,650	$0 \sim 2903$	1.19×10^{-2}
X-n716-k35	715	255,255	$-2 \sim 2096$	8.21×10^{-3}
X-n733-k159	732	267,546	$-2 \sim 1450$	5.42×10^{-3}
X-n749-k98	748	279,378	$-2 \sim 1346$	4.82×10^{-3}
X-n766-k71	765	292,230	$0 \sim 2942$	1.01×10^{-2}
X-n783-k48	782	305,371	$-2 \sim 2645$	8.66×10^{-3}
X-n801-k40	800	319,600	$0 \sim 2931$	9.17×10^{-3}
X-n819-k171	818	334,153	$-2 \sim 1452$	4.35×10^{-3}
X-n837-k142	836	349,030	$-2 \sim 2506$	7.18×10^{-3}
X-n856-k95	855	365,085	$-2 \sim 1465$	4.01×10^{-3}
X-n876-k59	875	382,375	$0 \sim 2523$	6.60×10^{-3}
X-n895-k37	894	399,171	$-2 \sim 2342$	5.87×10^{-3}
X-n916-k207	915	418,155	$0 \sim 2906$	6.95×10^{-3}
X-n936-k151	935	436,645	$-2 \sim 1442$	3.30×10^{-3}
X-n957-k87	956	456,490	$-2 \sim 1871$	4.10×10^{-3}
X-n979-k58	978	477,753	$0 \sim 2965$	6.21×10^{-3}
X-n1001-k43	1,000	499,500	$-1 \sim 2678$	5.36×10^{-3}

Table 2. Range and total count of the saving values (Set_XXL)

Instance	number of customers	m (number of saving values)	$s_{min} \sim s_{max}$	$(s_{max} - s_{min} + 1)/m$
Leuven1	3,000	4,498,500	$-2 \sim 2295$	5.10×10^{-4}
Leuven2	4,000	7,998,000	$-2 \sim 3555$	4.44×10^{-4}
Antwerp1	6,000	17,997,000	$-2 \sim 2844$	1.58×10^{-4}
Antwerp2	7,000	24,496,500	$-2 \sim 4637$	1.89×10^{-4}
Ghent1	10,000	49,995,000	$-2 \sim 2817$	5.63×10^{-5}
Ghent2	11,000	60,494,500	$-2 \sim 3354$	5.54×10^{-5}
Brussels1	15,000	112,492,500	$-2 \sim 2488$	2.21×10^{-5}
Brussels2	16,000	127,992,000	$-2 \sim 3858$	3.01×10^{-5}
Flanders1	20,000	199,990,000	$-2 \sim 31141$	1.56×10^{-4}
Flanders2	30,000	499,985,000	$-2 \sim 42082$	8.42×10^{-5}

5.1 Suitability of Counting and Pigeonhole Sorts

First, we examine the range and the total count of the saving values to determine whether counting sort or pigeonhole sort is suitable for using in the CW algorithm. As mentioned in Sect. 4, the range of the key values must be less than the total count of the key values to enable efficient sorting by counting sort and pigeonhole sort. In this assessment, the saving values are discretized by the approximate distances by Chaudhuri et al. [2] as described in Sect. 4.2.

Tables 1 and 2 present the corresponding evaluation results. In each table, the first column lists the name of the instances, while the second to fifth columns indicate the number of customer nodes, the total count of the saving values, the minimum and maximum saving values, and the range of the saving values over the total count of the saving values. As shown in Tables 1 and 2, the range of the saving values is substantially smaller than the total count of the saving values, thereby making it possible for the CW algorithm to achieve a best-case time complexity of $\mathcal{O}(n^2)$ using counting sort or pigeonhole sort.

5.2 Solution Quality

Next, we compare the solution quality achieved by each implementation. To assess the influence of the approximation of saving values, we compare the solution quality obtained by CW_0 with that by the other three implementations (CW_1, $OURS_c$, and $OURS_p$). The metric of this evaluation is the total Euclidean distance of the resulting routes, which is computed in accordance with the method described in [11] for calculating 2D Euclidean distance.

Tables 3 and 4 present the comparison of the solution quality. In each table, the second column displays quality for the best known solutions reported in [8], while the third and fourth columns show the solution quality obtained by CW_0 and the other three implementations. The values in parentheses indicate the relative gap between the best known solution quality and the solution quality obtained by each approach, calculated as $(c - c_{bks})/c_{bks} \times 100$. Herein, c_{bks} and c denote the best known solution quality and the solution quality obtained by each approach. Values on the last row in each table shows the average relative gap for each benchmark set. Since CW_1, $OURS_c$, and $OURS_p$ attained the same solution quality in every test case, we present the results collectively in Tables 3 and 4. This arose because all these three implementations yielded the identical orderings of the discretized saving values regardless of sorting algorithms. As shown in Tables 3 and 4, deterioration of solution quality with approximating the saving values is very little across all the test cases.

5.3 Execution Time

Finally, we compare the execution time of each implementation. In this evaluation, we executed each approach 10 times, respectively, and then compared the average time of the 10 trials because we noticed that the variations of execution time were non-negligible especially for small instances. We are now finding

Table 3. Comparison of solution quality (*Set_X*)

Instance	Best [8]	CW_0		CW_1, $OURS_c$, $OURS_p$	
X-n502-k39	69,226	71,784	(3.70 %)	71,253	(2.93 %)
X-n513-k21	24,201	27,088	(11.93 %)	26,768	(10.61 %)
X-n524-k153	154,593	164,488	(6.40 %)	166,726	(7.85 %)
X-n536-k96	94,846	99,959	(5.39 %)	99,778	(5.20 %)
X-n548-k50	86,700	89,792	(3.57 %)	89,611	(3.36 %)
X-n561-k42	42,717	45,628	(6.81 %)	45,976	(7.63 %)
X-n573-k30	50,673	53,287	(5.16 %)	53,303	(5.19 %)
X-n586-k159	190,316	199,690	(4.93 %)	200,431	(5.31 %)
X-n599-k92	108,451	113,065	(4.25 %)	114,271	(5.37 %)
X-n613-k62	59,535	62,758	(5.41 %)	62,727	(5.36 %)
X-n627-k43	62,164	65,548	(5.44 %)	65,841	(5.91 %)
X-n641-k35	63,682	68,220	(7.13 %)	67,747	(6.38 %)
X-n655-k131	106,780	108,226	(1.35 %)	108,302	(1.43 %)
X-n670-k130	146,332	158,692	(8.45 %)	159,616	(9.08 %)
X-n685-k75	68,205	71,503	(4.84 %)	71,817	(5.30 %)
X-n701-k44	81,923	85,427	(4.28 %)	85,919	(4.88 %)
X-n716-k35	43,373	45,831	(5.67 %)	46,073	(6.23 %)
X-n733-k159	136,187	139,446	(2.39 %)	140,459	(3.14 %)
X-n749-k98	77,269	79,573	(2.98 %)	80,025	(3.57 %)
X-n766-k71	114,417	120,402	(5.23 %)	120,253	(5.10 %)
X-n783-k48	72,386	76,496	(5.68 %)	77,360	(6.87 %)
X-n801-k40	73,305	77,218	(5.34 %)	77,908	(6.28 %)
X-n819-k171	158,121	166,547	(5.33 %)	166,195	(5.11 %)
X-n837-k142	193,737	202,048	(4.29 %)	202,590	(4.57 %)
X-n856-k95	88,965	92,740	(4.24 %)	92,805	(4.32 %)
X-n876-k59	99,299	102,182	(2.90 %)	103,068	(3.80 %)
X-n895-k37	53,860	59,071	(9.68 %)	58,522	(8.66 %)
X-n916-k207	329,179	342,876	(4.16 %)	343,448	(4.33 %)
X-n936-k151	132,715	145,897	(9.93 %)	148,830	(12.14 %)
X-n957-k87	85,465	89,442	(4.65 %)	89,646	(4.89 %)
X-n979-k58	118,976	123,857	(4.10 %)	123,823	(4.07 %)
X-n1001-k43	72,355	77,700	(7.39 %)	77,947	(7.73 %)
			(5.41 %)		(5.71 %)

Table 4. Comparison of solution quality (*Set_XXL*)

Instance	Best [8]	CW_0		CW_1, $OURS_c$, $OURS_p$	
Leuven1	192,848	200,829	(4.14 %)	200,978	(4.22 %)
Leuven2	111,391	125,063	(12.27 %)	126,089	(13.19 %)
Antwerp1	477,277	498,916	(4.53 %)	498,870	(4.52 %)
Antwerp2	291,350	322,575	(10.72 %)	322,199	(10.59 %)
Ghent1	469,531	490,230	(4.41 %)	490,544	(4.48 %)
Ghent2	257,748	286,542	(11.17 %)	288,815	(12.05 %)
Brussels1	501,719	531,589	(5.95 %)	533,372	(6.31 %)
Brussels2	345,468	385,108	(11.47 %)	385,980	(11.73 %)
Flanders1	7,240,118	7,525,070	(3.94 %)	7,535,722	(4.08 %)
Flanders2	4,373,244	4,804,764	(9.87 %)	4,813,413	(10.07 %)
			(7.85 %)		(8.12 %)

the reasons, but the execution time for small instances might be affected by the task scheduling of the operating system on our evaluation platform.

Tables 5 and 6 present the execution time of CW_0 and Tables 7 and 8 display the execution time of CW_1, $OURS_c$, and $OURS_p$. The instance names are abbreviated in the first column, while T_{total}, $T_{calc.sv}$, and T_{sort} indicate the total execution time, time to compute the saving values, and time to sort the saving values. Columns labeled as X_0 and X_1 in Tables 7 and 8 display the speedup of T_{total} in each implementation over CW_0 and CW_1, respectively. Note that T_{total} includes solution construction and additional overhead such as reading the instance files and dynamic memory allocation. Overall, CW_1, $OURS_c$, and $OURS_p$ outperform CW_0 as can be seen in Tables 5, 6, 7 and 8.

Table 5. Execution time of CW_0 for Set_X (in seconds)

Instance	T_{total}	$T_{calc.sv}$	T_{sort}
n502	0.019	0.00191	0.0134
n513	0.022	0.00240	0.0159
n524	0.028	0.00258	0.0166
n536	0.025	0.00265	0.0162
n548	0.024	0.00262	0.0167
n561	0.026	0.00288	0.0186
n573	0.023	0.00242	0.0169
n586	0.031	0.00309	0.0185
n599	0.030	0.00330	0.0199
n613	0.030	0.00344	0.0214
n627	0.029	0.00364	0.0203
n641	0.031	0.00391	0.0222
n655	0.036	0.00392	0.0238
n670	0.036	0.00353	0.0231
n685	0.036	0.00430	0.0249
n701	0.034	0.00404	0.0245
n716	0.034	0.00391	0.0249
n733	0.042	0.00389	0.0276
n749	0.043	0.00514	0.0293
n766	0.042	0.00520	0.0298
n783	0.043	0.00545	0.0311
n801	0.045	0.00599	0.0327
n819	0.053	0.00630	0.0349
n837	0.051	0.00595	0.0346
n856	0.053	0.00643	0.0376
n876	0.053	0.00671	0.0383
n895	0.056	0.00740	0.0404
n916	0.065	0.00765	0.0413
n936	0.069	0.00787	0.0452
n957	0.067	0.00796	0.0456
n979	0.070	0.00930	0.0468
n1001	0.072	0.00808	0.0501
average	0.041	0.00481	0.0282

Time to calculate the saving values in CW_1, $OURS_c$, and $OURS_p$ is overall faster than that in CW_0, except for some cases like n573 in $OURS_c$ and n502

Table 6. Comparison of execution time for *Set_XXL* (in seconds)

Instance	T_{total}	$T_{calc.sv}$	T_{sort}
L1	0.63	0.0530	0.531
L2	1.11	0.0904	0.957
A1	2.68	0.193	2.376
A2	3.59	0.264	3.170
G1	7.74	0.535	6.896
G2	9.30	0.647	8.286
B1	18.29	1.180	16.453
B2	20.38	1.339	18.314
F1	32.88	2.049	29.717
F2	74.96	4.501	68.118
average	17.16	1.085	15.482

Table 7. Execution time of CW_1, $OURS_c$, and $OURS_p$ for *Set_X* (in seconds)

Instance	CW_1				$OURS_c$					$OURS_p$				
	T_{total}	$T_{calc.sv}$	T_{sort}	X_0	T_{total}	$T_{calc.sv}$	T_{sort}	X_0	X_1	T_{total}	$T_{calc.sv}$	T_{sort}	X_0	X_1
n502	0.018	0.00184	0.0124	1.1	0.0083	0.00163	0.00177	2.3	2.2	0.029	0.00192	0.0231	0.7	0.6
n513	0.017	0.00181	0.0116	1.3	0.0073	0.00160	0.00175	2.9	2.3	0.018	0.00177	0.0137	1.2	0.9
n524	0.023	0.00191	0.0124	1.2	0.014	0.00180	0.00195	2.0	1.7	0.031	0.00191	0.0197	0.9	0.8
n536	0.020	0.00211	0.0118	1.3	0.013	0.00222	0.00236	2.0	1.6	0.024	0.00231	0.0154	1.1	0.9
n548	0.020	0.00199	0.0132	1.2	0.011	0.00228	0.00241	2.2	1.8	0.034	0.00227	0.0273	0.7	0.6
n561	0.020	0.00229	0.0133	1.3	0.010	0.00223	0.00243	2.6	2.0	0.021	0.00230	0.0147	1.2	0.9
n573	0.019	0.00213	0.0134	1.2	0.010	0.00251	0.00269	2.2	1.9	0.028	0.00235	0.0222	0.8	0.7
n586	0.027	0.00257	0.0144	1.2	0.015	0.00223	0.00253	2.1	1.8	0.037	0.00282	0.0252	0.8	0.7
n599	0.025	0.00285	0.0157	1.2	0.013	0.00242	0.00254	2.4	2.0	0.031	0.00271	0.0218	1.0	0.8
n613	0.022	0.00236	0.0140	1.4	0.011	0.00258	0.00274	2.7	1.9	0.024	0.00290	0.0159	1.3	0.9
n627	0.024	0.00279	0.0158	1.2	0.012	0.00279	0.00302	2.4	2.0	0.036	0.00279	0.0281	0.8	0.7
n641	0.024	0.00289	0.0169	1.3	0.011	0.00265	0.00290	2.8	2.2	0.035	0.00325	0.0273	0.9	0.7
n655	0.029	0.00299	0.0171	1.3	0.016	0.00317	0.00318	2.3	1.8	0.025	0.00295	0.0140	1.5	1.1
n670	0.030	0.00303	0.0174	1.2	0.016	0.00299	0.00320	2.3	1.9	0.031	0.00274	0.0196	1.2	1.0
n685	0.028	0.00350	0.0180	1.3	0.014	0.00333	0.00359	2.5	2.0	0.025	0.00338	0.0156	1.4	1.1
n701	0.029	0.00371	0.0196	1.2	0.013	0.00321	0.00355	2.6	2.2	0.036	0.00324	0.0271	1.0	0.8
n716	0.027	0.00349	0.0188	1.2	0.013	0.00354	0.00377	2.6	2.1	0.029	0.00368	0.0205	1.2	0.9
n733	0.032	0.00322	0.0189	1.3	0.018	0.00332	0.00358	2.4	1.8	0.030	0.00386	0.0163	1.4	1.1
n749	0.033	0.00418	0.0208	1.3	0.017	0.00423	0.00452	2.5	1.9	0.028	0.00435	0.0161	1.5	1.2
n766	0.034	0.00461	0.0222	1.2	0.017	0.00448	0.00470	2.5	2.0	0.040	0.00427	0.0286	1.1	0.9
n783	0.032	0.00375	0.0225	1.3	0.016	0.00422	0.00447	2.7	2.1	0.038	0.00480	0.0269	1.1	0.9
n801	0.036	0.00480	0.0247	1.3	0.015	0.00453	0.00462	3.0	2.4	0.037	0.00386	0.0271	1.2	1.0
n819	0.041	0.00514	0.0242	1.3	0.020	0.00437	0.00429	2.7	2.1	0.034	0.00540	0.0174	1.6	1.2
n837	0.040	0.00425	0.0251	1.3	0.022	0.00540	0.00549	2.4	1.9	0.040	0.00480	0.0252	1.3	1.0
n856	0.038	0.00444	0.0249	1.4	0.019	0.00534	0.00551	2.7	2.0	0.030	0.00532	0.0164	1.8	1.3
n876	0.041	0.00521	0.0279	1.3	0.018	0.00544	0.00547	3.0	2.3	0.037	0.00463	0.0246	1.4	1.1
n895	0.043	0.00604	0.0296	1.3	0.017	0.00530	0.00551	3.3	2.5	0.038	0.00602	0.0249	1.5	1.1
n916	0.052	0.00644	0.0310	1.2	0.024	0.00539	0.00561	2.7	2.2	0.050	0.00635	0.0303	1.3	1.0
n936	0.050	0.00646	0.0302	1.4	0.025	0.00634	0.00622	2.8	2.0	0.037	0.00693	0.0176	1.9	1.4
n957	0.048	0.00616	0.0320	1.4	0.022	0.00704	0.00646	3.1	2.2	0.035	0.00585	0.0206	1.9	1.4
n979	0.050	0.00705	0.0338	1.4	0.022	0.00719	0.00675	3.2	2.3	0.045	0.00702	0.0295	1.6	1.1
n1001	0.051	0.00719	0.0357	1.4	0.022	0.00803	0.00670	3.2	2.3	0.043	0.00713	0.0279	1.7	1.2
average	0.032	0.00385	0.0206	1.3	0.016	0.00387	0.00395	2.6	2.0	0.033	0.00393	0.0219	1.3	1.0

Table 8. Comparison of execution time for *Set_XXL* (in seconds)

Instance	CW_1				$OURS_c$					$OURS_p$				
	T_{total}	$T_{calc.sv}$	T_{sort}	X_0	T_{total}	$T_{calc.sv}$	T_{sort}	X_0	X_1	T_{total}	$T_{calc.sv}$	T_{sort}	X_0	X_1
L1	0.41	0.0432	0.321	1.5	0.14	0.0434	0.0522	4.5	2.9	0.16	0.0438	0.0689	4.0	2.6
L2	0.76	0.0753	0.617	1.5	0.24	0.0749	0.102	4.7	3.2	0.27	0.0746	0.134	4.1	2.8
A1	1.65	0.157	1.390	1.6	0.48	0.158	0.215	5.6	3.5	0.50	0.158	0.239	5.4	3.3
A2	2.35	0.214	1.986	1.5	0.69	0.216	0.327	5.2	3.4	0.76	0.214	0.395	4.7	3.1
G1	4.68	0.433	3.971	1.7	1.28	0.435	0.566	6.1	3.7	1.28	0.433	0.572	6.1	3.7
G2	5.90	0.524	5.046	1.6	1.61	0.525	0.759	5.8	3.7	1.66	0.520	0.819	5.6	3.5
B1	10.50	0.959	9.001	1.7	2.74	0.955	1.252	6.7	3.8	2.68	0.951	1.197	6.8	3.9
B2	12.65	1.093	10.933	1.6	3.30	1.084	1.592	6.2	3.8	3.37	1.082	1.668	6.0	3.8
F1	20.01	1.687	17.419	1.6	5.50	1.678	2.917	6.0	3.6	6.56	1.676	3.990	5.0	3.0
F2	47.12	3.700	41.452	1.6	12.84	3.690	7.168	5.8	3.7	15.54	3.676	9.894	4.8	3.0
average	10.60	0.888	9.214	1.6	2.88	0.886	1.495	6.0	3.7	3.28	0.883	1.898	5.2	3.2

in $OURS_p$. This suggests that discretizing the saving values with Euclidean distance approximation by Chaudhuri et al. [2] makes more sense than just rounding Euclidean distance calculated by *sqrt*() function.

The fact that CW_1 outperforms CW_0 indicates that discretizing the saving values itself contributes speeding up CW algorithm. This is because comparing the saving values in merge sort as well as calculating the saving values can be carried out by only integer operation. However, CW_1 still requires the time complexity of $\mathcal{O}(n^2 \log n)$ owing to merge sort for sorting the saving values, thereby leading to the limited speedup.

$OURS_c$ outperforms CW_0 and CW_1 in all the tested cases. It elapses 0.016 s for *Set_X*, and 2.88 s for *Set_XXL* on average, corresponding to 2.6× and 6.0× performance gain over CW_0, respectively. In particular, it solves Flanders2 comprising 30,000 customer nodes in 12.8 s, yielding 5.8× and 3.7× speedup over CW_0 and CW_1, respectively. $T_{calc.sv}$ and T_{sort} under $OURS_c$ remain lower than those under CW_0 in all the tested cases. Notably, T_{sort} is approximately 1/10 on average in *Set_XXL* compared with CW_0 and CW_1, accounting for a substantial portion of the performance advantage. Considering that counting sort can be applicable for only integer sort key values, discretizing the saving values based on approximate Euclidean distance has an essential role for such that performance gain.

As for $OURS_p$, it shows smaller performance gains overall and even underperforms CW_0 in some instances of *Set_X* while it achieves the comparable performance to $OURS_c$ in *Set_XXL*. To simplify the implementation, $OURS_p$ allocates sufficient memory with the same size for each pigeonhole to deal with any distribution of the saving values. This may cause considerable overhead and lead to the performance degradation for *Set_X*.

5.4 Comparison to Related Work

Herein, we compare our approach with two previous work aiming at reducing the time complexity mentioned in Sect. 3, i.e., CW^{100} [10] and GPU acceleration [13].

As mentioned in Sect. 3, CW^{100} in [10] required 3 to 34 s to solve instances ranging from 3,000 to 12,000 customer nodes on AMD Ryzen-3 1300X running at 3.5 GHz with 4 GB RAM, including post-optimization via the Lin-Kernighan (LK) heuristic. As detailed in Sect. 5, $OURS_c$ solved comparably sized instances (Leuven1, Leuven2, Antwerp1, Antwerp2, Ghent1, and Ghent2) in 1.61 s. In addition, we have confirmed that our approach did not exceed 2.17 s for these benchmarks even with the post-optimization via LK. Although a direct comparison is difficult due to the different evaluation environments, it is noteworthy that our approach presented an order-of-magnitude speedup over CW^{100}.

As mentioned in Sect. 3, the GPU acceleration in [13] did not address the inefficiencies in sorting the saving values. thereby leading to left the method inefficient for large-scale instances. As a result, it was about one-fifth and one-tenth the speed of our approach on average in *Set_X* and *Set_XXL*, respectively.

6 Conclusion

In this paper, we have proposed a fast savings algorithm that efficiently solves large-scale Euclidean CVRPs. Our proposed approach derives the saving values through the approximate Euclidean distance proposed by Chaudhuri et al. [2] using only integer arithmetic, thereby enabling the use of linear-time sorting algorithms for integer value sorting and attaining a best-case time complexity of $\mathcal{O}(n^2)$ for CVRP instances with n customer nodes. Experimental results demonstrate that our approach processes large-scale CVRP instances with $3,000$ to $30,000$ customer nodes in 2.88 s on average while preserving the solution quality, corresponding to 6.0× speedup over the original CW algorithm.

References

1. Clarke, G., Wright, J.W.: Scheduling of vehicles from a central depot to a number of delivery points. Oper. Res. **12**(4), 568–581 (1964)
2. Chaudhuri, D., Murthy, C.A., Chaudhuri, B.B.: A modified metric to compute distance. Pattern Recogn. **25**(7), 667–677 (1992)
3. Cormen, T.H., Leiserson, C. E., Rivest, R.L., Stein, C.: Introduction to Algorithms (Third Edition). MIT-Press (2022)
4. Stephens, R.: Essential Algorithms: A Practical Approach to Computer Algorithms Using Python and C#. John Wiley & Sons, Inc (2019)
5. Altinel, İ., K., Öncan, T.: A new enhancement of the Clarke and Wright savings heuristics for the capacitated vehicle routing problem. J. Operat. Res. Soc. **56**(8), 954–961 (2005)
6. Doruyan, T., Çatay, B.: A robust enhancement to the Clarke-Wright savings algorithm. J. Operat. Res. Soc. **62**(1), 223–231 (2011)

7. Stanjoević, M., Stanjoević, B., Vujošević, M.: Enhanced savings calculation and its applications for solving capacitated vehicle routing problem. Appl. Math. Comput. **219**, 10302–10312 (2013)
8. CVRPLIB. http://vrp.galgos.inf.puc-rio.br/index.php/en/, Accessed 25th Mar 2025
9. Uchoa, E., Pecin, D., Pessoa, A., Poggi, M., Vidal, T.: New benchmark instances for the capacitated vehicle routing problem. Eur. J. Oper. Res. **257**, 845–858 (2017)
10. Arnold, F., Gendreau, M., Sörensen, K.: Efficiently solving very large-scale routing problems. Comput. Oper. Res. **107**, 32–42 (2019)
11. Reinelt, G.: TSPLIB-a traveling salesman problem library. ORSA J. Comput. **3**(4), 376–384 (1991)
12. Guerriero, F., Saccomanno, F.P.: Accelerating the Clarke-Wright algorithm using GPUs. Control Cybernet. **53**(2), 371–383 (2024)
13. Guerriero, F., Saccomanno, F.P.: A parallel implementation of the clarke-wright algorithm on GPUs. ICORES-2025, pp. 100–111 (2025)