About the prediction of good edges for the CVRP and its applications

Clément Legrand-Lixon
Univ Rennes, F-35000 Rennes, France

No Institute Given

Abstract. Few decades ago, it was almost impossible to solve with a high precision large combinatorial optimization problems. Due to their exponential solution space, even now with all computers of the world, small instances cannot be optimally solved with a naive approach. However, considering the Capacitated Vehicle Routing Problem, it is now possible to find near-optimal solutions in just few hours, even for very large instances. Achieving such results was possible thanks to a very active community on the subject, which developed plenty of heuristics and meta-heuristics to solve the problem. For instance recent works used machine learning to predict if a given solution was near-optimal or not. According to the good results they obtained, we decided to use machine learning to predict the probability that have two customers are connected in good solutions. This article focuses on how we did the learning phase, and then how we exploited it to improve the Clarke & Wright algorithm and a local search heuristic.

Keywords: Combinatorial optimization · Capacitated Vehicle Routing Problem · Local search · Meta-heuristics · Machine learning

reference: Internship realized from the 03rd June 2019 to the 04th August 2019, and supervized by professor Daniele Vigo at University of Bologna, Scuola di Ingegneria e Architettura.

Introduction

Everyday companies have to tackle plenty of real-life problems. In the case of delivery companies, they have to manage a fleet of vehicles to deliver every customers within a certain time. Such real-life problems are often combinatorial optimization problems. These problems are generally very hard to solve because their solution space grows exponentially with the size of the problem. The Vehicle Routing Problem (VRP) is one of them. Its purpose is to design routes of vehicle to serve n customers while minimizing the total length of the fleet. Each route must start and end at the same place, called depot. This problem dates back to 1959 [7] and has been extensively studied ([5],[22],[8]). The VRP is a NP-complete problem, and its definition can be easily extended by taking into account several real-world constraints such as time windows, multi-depot, heterogeneous vehicles, pickup and delivery, ... The collection of extended problems has been thoroughly surveyed by Laporte [16], Lahyani [15] and Braekers [2]. One of the most studied variant of the VRP, includes a parameter for customers: their demand, and a new constraint for vehicles: a capacity (which can be the same for all vehicles). By adding this constraint, the sum of the demands of all customers on a route cannot exceed the capacity of the vehicle. This problem is also known as the Capacitated Vehicle Routing Problem (CVRP). We can find in the literature many algorithms and heuristics to solve the CVRP [13]. These algorithms can be classified as exact algorithms and approximation algorithms. Exact algorithms find the optimal solution of a problem with an exhaustive search. To be efficient such algorithms must use pruning techniques which restrict the solution space. On the other hand, approximation algorithms can find good solutions, but can not guarantee their optimality. Within approximation algorithms we can find heuristics and meta-heuristics. Heuristics can only guide choices of an algorithm and are specific to a problem. Contrary to a heuristic, a meta-heuristic can be used for different problems. For example stochastic local search [12] and genetic algorithm [11] are meta-heuristics. Moreover approximation algorithms need a trade-off between two opposites trends: intensification and diversification. Intensification means that the algorithm will concentrate the search around the best solution found. While diversification tries to explore new areas of the solution space.

An efficient way to improve heuristics and meta-heuristics on a problem, is to use specific-knowledge of the problem. In his thesis, Florian Arnold [1], describes an efficient knowledge-guided local search to find good solutions for the CVRP and variants. Following this work, we try to predict edges that appear in good solutions with a neural network, then we use these predictions to improve the Clarke & Wright [6] algorithm and a simple local search algorithm [1].

Section 1 presents the problem more formally and introduces the notation. Section 2 explains how a local search can be performed to solve optimization problems and details a way to perform a guided local search. Section 3 describes how the neural network has been trained, and how the Clarke & Wright algorithm has been improved. Finally section 4 compares results obtained with our last year algorithm, and those obtained with our new algorithm.

1 Problem specification

This section recalls first the notion of combinatorial optimization, and then formalizes the routing problem.

1.1 Combinatorial Optimization Problems

Formally, a combinatorial optimization problem A is a quadruple (I, f, m, g) where:

- I is a set of instances;

- given an instance $x \in I$, $f(x)$ is the set of feasible solutions, also called *solution space* (i.e. which respect all the constraints of the problem);
- given an instance x and a feasible solution s of x, $m(x, s)$ denotes the measure (or the cost) of s, which is usually a positive real number;
- g is the goal (or objective) function, and is either min or max.

The goal is then to find for some instance x an *optimal solution*, that is, a feasible solution s^* with:

$$m(x, s^*) = g\{m(x, s')|s' \in f(x)\}$$

Most discrete optimization problems are NP-complete, therefore approximate methods (like meta-heuristics) are developed to solve them with high accuracy. Recent trends[1] are focusing on the integration of learning techniques into meta-heuristics to improve their performance. Moreover many real-life problems can be designed as discrete optimization problems, like parcels delivery. Thus there is a lot of interest in finding algorithms to solve such problems efficiently.

1.2 Routing Problems

Vehicle Routing Problem Routing problems are omnipresent in the field of combinatorial optimization and can be formally described as follows [1]. Let N denote the number of destinations that need to be visited from one point of origin. In the following, we will use the standard terminology and call destinations *customers* and the point of origin *depot*. Every customer is usually identified with a non negative integer in $\{1, \dots, N\}$ and the depot with 0. We can model a routing problem as a graph $G = (V, E)$, where V denotes the set of nodes, and E the set of edges. The $|V| = N+1$ nodes reflect the customers and the depot (v_0). The edges (i, j) reflect the connections between nodes i and j that can be taken. In the standard formulation of the problem, G is complete and undirected , so that each node can be visited from each other node. Each edge (i, j) is annotated with a cost $c(i, j)$ that is realized if the edge is taken. This cost usually corresponds to the distance between two nodes, however, it can also be replaced by other metrics. A *path* in G is a sequence of adjacent edges $(i, j), (j, k), \dots$ so that all visited nodes are distinct. A *route* in G is then defined as a path that starts at v_0, and has an additional edge back to v_0, i.e., $(v_0, i), (i, j), \dots, (k, l), (l, v_0)$. We can now give the definition of the Vehicle Routing Problem.

Definition 1 (Vehicle Routing Problem (VRP)). *Given a complete graph G, find a set of routes that visits all customers and minimizes the sum of the costs of the involved edges.*

In order to define the objective function of the *VRP* and constraints on routes we introduce other notations. We define $\delta_{i,j}$ which is equal to 1, if there exists a route in which the customer j is served after the customer i, and 0 otherwise. Note that according to this definition we can not have $\delta_{i,j} = 1$ and $\delta_{j,i} = 1$. It is also possible to restrict the latter definition to a single route r with the notation

$\delta^r_{i,j}$: it will be equal to 1 if the customer j is served after the customer i in route r, and 0 otherwise. In this way, we can link a solution s, that is, a set of routes, with a matrix $\mathcal{M}$ of size $N+1$ such that $\mathcal{M}(s)_{i,j} = c(i,j) \cdot \delta_{i,j}$. Hence the cost of a solution s for the graph G is defined as follows:

$$m(G,s) = \sum_{i=0}^{N} \sum_{j=0}^{N} \mathcal{M}(s)_{i,j} = \sum_{i=0}^{N} \sum_{j=0}^{N} c(i,j) \cdot \delta_{i,j}$$

We can also formalize constraints on routes as follows:

- All customers are exactly served once: $\forall i > 0 \sum_{j=0}^{N} \delta_{i,j} = 1$;
- All routes start and finish at the depot: $\forall r \sum_{j=0}^{N} \delta^r_{0,j} = 1$ and $\sum_{i=0}^{N} \delta^r_{i,0} = 1$.

Thus solving a *VRP* instance, means finding a solution s^* which respects all previous constraints (also called feasible solution), such that:

$$m(G,s^*) = \min_{s\ feasible} m(G,s)$$

Routing problems are NP-hard, which means that every NP-problem can be reduced to them, where NP stands for nondeterministic polynomial time. In other words, routing problems are at least as hard as any NP-problem, and there does not (yet) exist a deterministic algorithm which solves them in polynomial time. Moreover the solution space of the VRP grows exponentially (we have already $N!$ possible combinations to build a single route). For instance with $N = 60$ customers, the number of VRP solutions is larger than the number of observable atoms in the universe (around 10^{81}). That's why efficient heuristics have been developed to find optimal solutions.

Capacitated Vehicle Routing Problem In many applications the routes have to respect limitations. For instance, a truck can only transport a given amount of parcels. Formally, these constraints are modelled by allocating a demand d_i to each customer i. The sum of the demand of all customers on a route may not exceed a certain value $\mathcal{Q}$. This limitation is commonly called *capacity constraint.* For instance, d_i could represent the number of parcels that have to be delivered to customer i and $\mathcal{Q}$ could define the limit of parcels that can be transported in one vehicle. Because of these limitations multiple routes might have to be planned. This lead to the definition of the Capacitated Vehicle Routing Problem as introduced by Dantzig and Ramser [7].

Definition 2 (Capacitated Vehicle Routing Problem (CVRP)). *Given a complete graph G, find a set of routes that visits all customers, **respects capacity constraints** and minimizes the sum of the costs of the involved edges.*

The cost and objective functions for the CVRP are still the same as for the VRP. We have only added a constraint on routes, which can be formalized as follows:

– On all routes the capacity constraint is satisfied: $\forall r \sum_{i=0}^{N} \sum_{j=0}^{N} d_{i,j} \cdot \delta_{i,j}^{r} \leq \mathcal{Q}$.

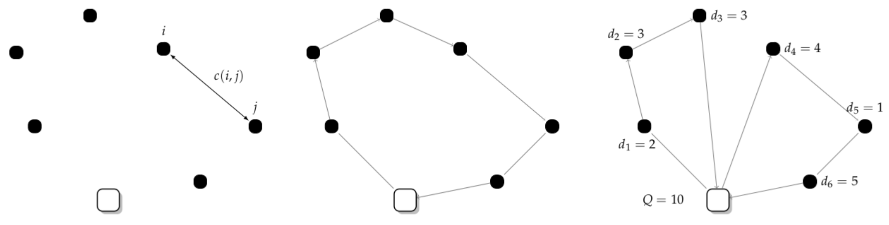

Fig. 1. (Left) A routing problem is defined by a depot (square) node and several customer nodes (black dots) that need to be visited. Each edge between two nodes is annotated with a label $c(i, j)$. (Middle) A solution for a VRP. (Right) A solution for a CVRP, in which the capacity restriction has to be respected.

The figure 1 gives an example of a routing problem instance.

Evidently as the CVRP is more complex than the VRP, its solution space is much larger. Thus there exists in the literature, plenty of algorithms to solve the CVRP. One of the most recent has been designed by F. Arnold in his thesis [1], and performs a knowledge-guided local search to find good results. As we have drown from this algorithm, the next section focuses on bases of local search algorithms.

2 Local Search

Local search has proven to be the cornerstone of many solutions techniques for various combinatorial optimization problems [12]. Efficient meta-heuristics generally perform an effective local search, e.g., variable neighborhood search [20], or guided local search [23]. In this section we aim to introduce the notion of local search, and present some knowledge that can be used to guide a CVRP heuristic.

2.1 Efficient Local Search for the CVRP

Notion of local search. The basic idea underlying local search is that high-quality solutions of an optimization problem can be found by iteratively improving a solution using small (local) modifications, called *moves*. A *local search operator* specifies a move *type* and generates a *neighborhood* of the current solution. Given solution s, the neighborhood of a local search operator is the set of solutions $\mathcal{N}(s)$, that can be reached from s by applying a single move of that type.

After generating the neighborhood of the current solution, the neighborhood is evaluated, and local search uses an *exploration strategy* to accept at most one solution from the neighborhood $\mathcal{N}(s)$ to become the next current solution. The hill climbing (i.e. the selection of the best solution of the neighborhood according to the cost function) is the most commonly used move strategy, however the entire exploration of the neighborhoods can be time consuming. To avoid this, the *first improvement* strategy is often preferred. Thus, the first neighbor found that is better than the current one is accepted. In order to hopefully explore more diversified solutions, we will explore the neighborhoods randomly. If no improving solution is found in the neighborhood of the current solution, a *local optimum* for that neighborhood (i.e. local search operator) has been reached.

Local Search for the CVRP In general local search operators for the CVRP can be distinguished between operators for *intra-route optimization* and operators for *inter-route optimization.* These two operators types reflect the two tasks that one has to solve in a CVRP:

- The optimization of each route in itself (intra-route optimization);
- The allocation of customers to routes (inter-route optimization).

Intra-route optimization can be executed rather efficiently, since it corresponds to solving a Travelling Salesman Problem (a particular case of the VRP but with only one route), with relatively few customers. Therefore, it seems sensible to optimize the routes themselves, before they are optimized jointly. The *Lin-Kernighan* operator [18] (LK) has proven to be very efficient to solve the TSP, and it is still used to solve very large instance of the problem. This operator generalizes the k-opt operators. A k-opt operator tries to replace k edges with k new edges to improve the current solution. It is very hard to efficiently implement a LK operator, and such an implementation is described by Helsgaun [10]. However to simplify our algorithm we will only consider a 2-opt operator.

An important observation is that a local optimum for one local search operator is generally not a local optimum for another one. For this reason, it is sensible to use several local search operators in a single algorithm. However the usage of several local search operators is only beneficial as long as they explore different neighborhoods. In the best case, the pairwise intersections of the considered neighborhoods are empty, i.e, given two operators o_1 and o_2 we have $\mathcal{N}^{o_1}(s) \cap \mathcal{N}^{o_2}(s) = \varnothing$ for every solution s.

Considering the previous remark, we will use two complementary operators to perform an inter-route optimization. Many effective local search operators have been developed over the years, and have been condensed in an online library [9]. These two operators are the Cross-Exchange (CE) and the Relocation-Chain (RC). An efficient implementation of these two operators is described in the thesis of F. Arnold [1].

The CE operator is a generic local search operator that tries to exchange two substrings $\hat{r}_i$ and $\hat{r}_j$ of two different routes r_i and r_j [21].

A RC starts with a relocation of a customer node from route r_i into r_j. This relocation is followed by a relocation of a customer node from route r_j into

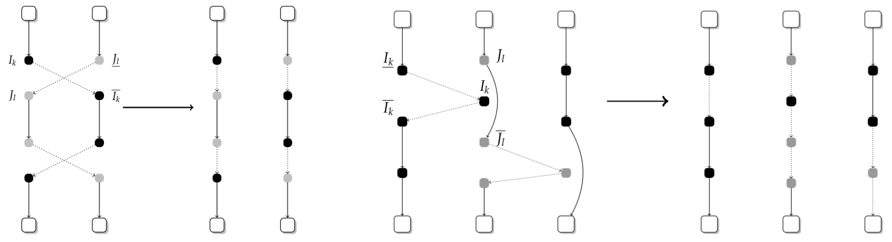

Fig. 2. (Left) Illustration of a CE move with substrings of two customers. (Right) Illustration of a relocation chain with two customers.

route r_k (where $i = k$ is possible). This process can be repeated until an upper bound of relocations is reached. Note that a relocation of a node from r_i into r_j might improve the solution, but exceed the capacity constraints of r_j. However, the move might become feasible, if we make space, by simultaneously relocating a node from r_j into another route. An example of these two operators can be visualized in figure 2.

Complexity of Local Search A last point about local search that we need to discuss is its complexity. In general, the computational complexity of a local search operator depends on the size of its neighborhood. The larger the neighborhood, the more solutions need to be generated and evaluated. On the other hand, larger neighborhoods also come with a larger probability of finding improvements. Consequently, there is a trade-off between computational complexity and the probability of improvement. This trade-off presents one of the greatest challenges in the design of an efficient local search. For larger instances (about hundreds of customers), a quadratic complexity might already be computationally too expensive. A commonly used tool to reduce the complexity of local search operators is *heuristic pruning*. Rather than generating the entire neighborhood for each operator, pruning tries to limit the considered neighborhood to promising options. Thus, according to the results of F. Arnold, we will only consider moves among the 30 nearest neighbors of the initial customer to generate neighborhoods. Moreover as explained before, we will not generate the entire neighborhood for each operator. We apply the first improvement move found during the search (note that neighborhoods are explored randomly to not favour certain solutions), contrarily to F. Arnold who entirely explores the neighborhoods, with efficient pruning methods, and then applies the best possible move.

The complexity of operators used is the following (n refers to the average length of the routes). The 2-opt operator has a complexity of $O(n^2)$. For the CE operator, the generation of all substrings of a route has a theoretical complexity of $O(n^2)$, and thus matching two substrings amounts to $O(n^4)$ its complexity. The complexity of a RC is relatively high, and grows exponentially with the number of relocations we make: with k relocations we have a complexity of

$O(n^{2^{k-1}})$. Thus, like F. Arnold we take an upper bound of 3 relocations to limit the computational time.

2.2 Guided Local Search

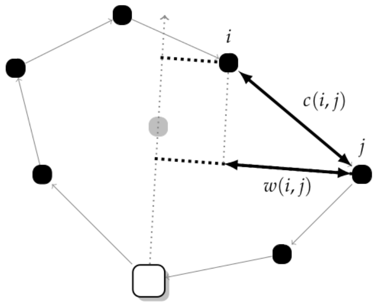

Fig. 3. Metrics determining the "badness" of an edge (i, j). The gray dot reflects to the gravity center of the route.

During his thesis F. Arnold evaluated the influence of several metrics to find which ones are more relevant to characterize good and bad edges. Finally he found that two metrics were particularly interesting to study edges: their cost (smaller edges appear with higher probability in better solutions) noted c, and their *width* (routes tend to be more compact in better solutions) noted w. The width of an edge is linked to the route in which it belongs. It is computed as the distance between nodes i and j measured along the axis perpendicular to the line connecting the depot and the route's center of gravity. The x-coordinate of the gravity center of a route is simply the average of x-coordinates of all nodes on the route (customers and depot), y-coordinate is computed accordingly. These two metrics are illustrated in figure 3. Then one can obtain functions that measure the "badness" of an edge (i, j) [1]:

$$b(i,j) = \frac{\lambda_w w(i,j) + \lambda_c c(i,j)}{1 + p(i,j)}$$

Variables λ_w and λ_c only indicate if the metric is taken into account in the function, hence they are equals to 0 or 1. Thus we can use 3 different functions to measure the "badness" of an edge. The function $p(i, j)$ refers to the number of times that edge (i, j) has been penalized. This value is incremented each time that (i, j) maximizes the function b, that is, when (i, j) is considered to be the *worst* edge of the solution.

The idea behind guided local search is to change the evaluation cost of s, and then improve s by using this other evaluation cost, rather than directly change s itself. Features that are considered bad are penalized. In the context of the CVRP, those features are edges, hence we should increase the cost of bad edges [1]. More formally, we change the cost $c(i,j)$ of an edge in the current solution between customers i and j to

$$c^g(i,j) = c(i,j) + \lambda p(i,j)L$$

where L is the average cost of an edge in the current solution and λ controls the impact of penalties. Kilby [14] experimentally found $\lambda \in [0.1, 0.3]$ to be a good choice, so we keep $\lambda = 0.1$. We use this guided local search in section 4.

3 Prediction of good arcs

According to the results obtained last year [17], it seemed to be promising to predict with a high accuracy edges that appear in good solutions. With such predictions, we could simply create a good initial solution for any instance, or improve our guided local search. In this section we present the learning phase, and how we use the prediction to generate an initial solution using the Clarke & Wright algorithm.

3.1 Learning phase

Features At first look, many features could be relevant to describe an edge in a solution (for instance the placement in a route, the distance to the depot or simply the cost). However if we consider features that depend on routes, we should first compute a route that contains the edge we want to evaluate, and then the probability of the edge will depend on the tour created. The edge could belong to the optimal solution, if we create a very bad route, the probability of the edge will be very low. Thus we can not consider features that depend on other edges. This condition drastically reduces the set of features that we can consider. As we will only consider symmetric instances (edges (i,j) and (j,i) have the same cost) in the following, we will only study edges (i,j) where $j > i$. According to the thesis of F. Arnold [1], for an edge (i,j) we will simply consider the following features that appeared as the most relevant: the cost (euclidean distance between i and j), the demand of i (resp. j), the distance between i (resp. j) and its nearest neighbor, the demand of the nearest neighbor of i (resp. j), the distance between i (resp. j) and the depot, and finally the angle $\widehat{iv_0j}$ (v_0 is the depot). Most of these features are common to describe an edge and it seems reasonable to consider them. Features like distance between a customer and the depot or the angle, have been showed [1] relevant to distinguish optimal and near-optimal solutions. Now as we want our neural network to be applicable on several instances, we have to normalize all previous features to make them independent from the instance. To normalize demands we simply divide them by the capacity of the instance, and to normalize costs we divide them by the highest cost of the instance between two customers.

Preparation of Data Initially [17], we generated a set of good solutions, and then we computed a frequency matrix, such that a coefficient in the matrix refers to the number of times that the corresponding edge appeared in the generated set. Then we found that keeping edges that have a coefficient in the matrix of at least 0.5 gave the best results. However this technique had disadvantages like an important computational time for large instances (to generate the solution set we used the Clarke and Wright algorithm, which will be detailed in section 3.2). On the other hand, we considered that edges that appear in very good solutions and bad solutions had the same weight, but it should not be the case. To fix this latter problem we introduce a formula which attributes a weight to each edges in a solution. The weight varies from 0 to 100 and depends on the quality of the solution. Given a *cost of reference*, the better will be the solution considering the reference cost, the higher will be the weight of each edge. All solutions that have a cost higher than an upper bound will have a weight of 0. We take 0 as a lower bound (we can not easily know if a solution is far or not from the optimal cost). The reference cost and the upper bound are computed as follows. The upper bound is defined as the cost of the worst solution we generated during the algorithm. However for the reference cost we can not simply take the best solution obtained. Indeed there will not be big gaps between weights of good and bad solutions and it is what we want to avoid. That's why we take a reference cost which corresponds to the mean between the cost of the best solution and the cost of the worst one. In the following, the variable z referes to the cost of the solution we consider. Similarly z_{ref} (resp. z_{UB}, z_{LB}) refers to the reference cost (resp. upper bound, lower bound). The weight associated to a solution of cost z is noted $\hat{\Delta}(z)$. We introduce the gap between z and the reference cost as $\Delta(z) = 100 \times \frac{z - z_{ref}}{z_{ref}}$. We then scale $\Delta(z)$ between 0 and 100, to obtain the weight.

$$\hat{\Delta}(z) = \frac{100 - 1}{\Delta(z_{UB}) - \Delta(z_{LB}) - 0} \times ((\Delta(z_{UB}) - \Delta(z)) - (\Delta(z_{UB}) - \Delta(z_{LB})))$$

Let N_s the number of solutions generated during our algorithm. Then the edge (i, j) will have a coefficient in the matrix which is the sum of the weights of the edge in all N_s solutions. We divide by $N_s \times 100$ to obtain a coefficient between 0 and 1.

To feed the neural network, we used the best-known solutions of the Uchoa set of instances. The Uchoa set is very challenging, mixes various instances, and it is commonly used to test new algorithms for the CVRP. To compute the frequency matrix starting with the best-known solutions, we generate a slightly worse solution using a perturbation method (described in section 4), and we use algorithm 2 in annex without the Restart phase). Then we do as described in the latter section. We have simply randomly chosen a first half of the instances for the training set, the second half is for the validation set. To generate data, we compute for each edge of the instance all its features, and we assign the label "good" if the weight of the edge in the corresponding matrix exceeds 0.5, otherwise we assign the label "bad". We have to keep in mind that any instance of N

customers add $\frac{N(N+1)}{2}$ new examples. However negative examples (edges labelled "bad") are much more plentiful than positive examples (edges labelled "good", that we want to predict). We find three different ways to solve the imbalanced data problem:

- Oversample the minority class: duplicate each positive example;
- Downsample the majority class: keep each negative example with some probability;
- Generate artificial positive examples with the SMOTE technique.

Note that oversample introduces an important bias in our data. Moreover creating artificial examples for our data does not seem to be a good idea either, because "good" and "bad" edges can be very similar. Finally, after some experiments, we find that the downsampling method gives the best results.

To perform the learning phase we use the module scikit-learn of python and a MLP classifier as neural network. Parameters are chosen according to those described in the book Deep Learning with Python of François Chollet[3], to realize a binary classification. Besides to realize our downsampling, we simply keep each negative example with different probabilities. The results we obtained are presented in table 2 in annex. We notice that a low ratio gives a very high accuracy, but it is due to the imbalanced data. Moreover in these cases, precision and recall for "good" edges is very low. On the other hand, a high ratio decreases the accuracy, but we have more balanced statistics with the two classes. That's why in the following section, we will only consider neural networks obtained with probabilities 0.02, 0.01 and 0.005 (we tried also with the other networks but it did not give better results).

In the following the notation $neuralNetwork(i, j)$ will refer to the probability of the edge (i, j) to be "good", according to the neural network. If we want to precise the probability p used to generate the data with which we obtained the neural network, we will note $neuralNetwork^p$.

3.2 Application to the Construction of an Initial Solution

A common strategy to generate an initial solution to a routing problem is the Clarke & Wright algorithm (CW). The first version of the algorithm dates back to 1964 [6], but since then many variants have been suggested. Initially, there are as many routes as customers and each route serves exactly one customer. The general principle of the algorithm is based on computing *savings*, and then merge routes until the saving become negative. Algorithm 1 in annex gives a general framework of the algorithm. Usually, the formula to compute the savings differs from a variant to another. In general, the algorithm does not generate good solutions, but it is a good starting point for a local search procedure.

We suggest here other ways to generate an initial solution, by using three savings formulas. The first one is simply the formula used in the basic CW algorithm, in the second one we replaced the savings by the probability of being a "good" edge, in the last one we multiply the two latter formula:

- Basic saving: $saving^{basic}(i,j) = c(i,0) + c(0,j) - c(i,j)$;
- Probability: $saving^{proba}(i,j) = neuralNetwork(i,j)$
- Product: $saving^{product}(i,j) = saving^{basic}(i,j) \times saving^{proba}(i,j)$

These three formulas define three distinct CW algorithms. Another CW variant, noted *SubLists*, is defined as follows. We use the basic saving formula to compute all the savings. Then, we sort the saving list and we divide it into sublists of size l, but we keep the order of the savings (i.e. the first sublist contains the l higher savings, the following sublist the next l higher and so on). In each sublist we associate to a saving, the probability of the corresponding edge, and we sort the sublist by decreasing probability. Finally, we concatenate all the sublists by conserving their initial macro order. The rest of the algorithm is the same as a basic CW algorithm. Note that if we take $l = 1$, we retrieve the *Basic saving* algorithm, and with $l = |allSavings|$ we retrieve the *Probability* variant.

By experimenting the *SubLists* variant, we have noticed that we cannot fix a value of l such that we obtain the best initial solution with this method for all instances. To fix this problem, we consider a last variant *Best SubLists*, in which we define a list L_l of value for l, and we keep the best solution obtained among the $|L_l|$ solutions obtained. Experimentally we found that taking $l = 1$ and $l = N$ gave good results, so we decide to keep them in L_l. We have also added in L_l all integers between 2 and $N - 1$ with a step of $N//10$ to keep a good trade-off between computational time and results.

We compare the results obtained with the five versions of CW algorithm described (and applying on the computed solutions the 2-opt operator, to obtain an improved solution) on the 50 first instances of the Uchoa set. We compute for each solution obtained its gap from the best-known solution, and we keep the average gap over the 50 instances. Results can be visualized in table 3 in annex. The variants *Probability* and *Product* compute worse solutions than the *Basic* CW. With the variant *SubLists*, for all values of l chosen, we obtain in average better solutions than the *Basic* CW, and if we preprocess the training of the neural network and the prediction probability of each edge, we have even a similar time to the *Basic* CW. Finally we obtained best results with the variant *Best SubLists*, which improves in average solutions by almost 2.5%. However with this variant we have to sacrifice a little more time in average. Finally, according to the results, we will only keep the neural network $neuralNetwork^{0.005}$ and use the variant *SubLists* with $l = N$, to have a lower computational time.

4 Description of our Algorithm and Results

This section compares the results obtained with two algorithms: one designed last year [17] and a new one described in the section.

Algorithm We tried to improve our last year algorithm by bringing some modifications and improving the local search. Local search has been improved by fixing some issues, and using efficient pruning for the operators. The major modifications can be seen in red on algorithm 2 in annex. We detailed in the following the two major modifications: the restart and the perturbation, which are efficient methods to perform an iterative local search [19].

New "Restart" Phase The restart phase occurs when no more improvement are found for a fixed number of iterations. It simply consists in creating a new solution of lower quality than the best solution we found. Jan Christiaens and Greet Vanden Berghe have suggested [4] an efficient way to compute good solutions for the CVRP by using a ruin and recreate method, called *SISRs*. So we decided to implement the ruin method of *SISRs* to improve our restart. We recreate then a solution with the method used for generating our initial solution. More precisely we generate (nbRuins) solutions (we empirically chose $nbRuins = 20$ solutions to have a good trade-off between computation time and exploration) which have at least a gap of 1% from the best solution or improve the best solution. We only keep the best generated solution according to the latter condition. However in their paper Christiaens and Berghe choose the starting customer of the ruin phase randomly. Therefore to explore a bigger neighborhood we compute a list of starting points (of length nbRuins) as follows. The first point is randomly chosen among all possible customers of the instance. Then we choose the next point randomly among customers that are not in any nearest neighbors set of our starting points (but we consider only those which have been generated from the last time we reset). If there are no such customers we reset by choosing a random customer of the instance.

New Perturbation Phase The major loss of time when a local search is performed, is in general due to local optimum. To avoid this loss of time, we decide to apply a perturbation each time that a local optimum is detected. We consider that a local optimum has been reached if the current solution cannot be improved during a small amount of iterations, or whether our current solution has already been considered to be a local optimum. When a solution is considered to be a local optimum, we authorize the local search operators to return a slightly worse solution than the current solution to avoid the local optimum. The perturbation consists in a small number of iteration (according to the results of F. Arnold, we choose 10 iterations), during which we iteratively apply the local search operators with the modified cost function defined in section 2.2. Operators are applied around the *worst* edge as defined in section 2.2. Note that an improving move with c^g is in general not an improving move for c. After the perturbation we change the penalization function by picking another pair (λ_c, λ_w).

Results obtained We first compare our last year results and those obtained with the new algorithm, on sets A, B and P. All the results can be visualized in tables 4, 5 and 6 in annex. Most of the results obtained with our new algorithm are better (in time and quality) than the previous one. We then execute our algorithm on the 50 first instances of Uchoa set (called X set) and results can be visualized in tables 7 and 8 in annex. Most results are good (below 1%), however the computation time is quite high. Moreover we have still a progress margin, according to the results of F. Arnold. All the results are summarized in table 1.

Set	Size	LearnHeurisrtic		MachineLearning	
		Average Gap (%)	Total Time (m)	Average Gap (%)	Total Time (m)
A	27	0.23	234.1	0.14	27.5
B	20	0.37	187.9	0.17	23.4
P	21	0.24	176.1	0.12	38.9
X	50	NA	NA	1.13	1329.2

Table 1. Synthesis of results obtained for instances A, B, P and X.

Conclusion

In this article we have performed a binary classification thanks to a neural network to predict the probability that has an edge to appear in good solutions. Due to the imbalanced data, we have decided to downsample the majority class (i.e. "bad" edges) to improve ours results. We have then used the results of the prediction to improve the basic Clarke & Wright algorithm. In addition, we improved our last year algorithm with some corrections and modifications. Indeed we have added a perturbation phase and modified the restart phase. All the python code designed is available on gitlab `https://gitlab.com/Lixonclem/master_internship`.

Results obtained are promising, but we are still far from the results of F. Arnold. First our code is in python so it is slower than a C++ code. Moreover it is possible that our implementation is not as efficient as that of F. Arnold, and that the chosen parameters for the neural network are not the bests.

As future work we could try to improve the range of our prediction to set of customers and not only edges. We could also analyze the (potential) correlation between the savings and the probabilities predicted by the neural network. Moreover the CW variant *SubLists* may could be improved by using variable steps instead of a constant step.

Acknowledgments

I thank professor Daniele Vigo at DEI laboratory of university of Bologna (Italy) who supervised this project. I thank Luca Accorsi (phd student) who worked with me on this project. I would also thank Laetitia Jourdan, Marie-Eléonore Kessaci and Diego Cataruzza from CRIStAL laboratory of Lille (France) who helped me on this project.

References

1. Florian Arnold. *Efficient heuristics for routing and integrated logistics.* PhD thesis, University of Antwerp, 2018.
2. Kris Braekers, Katrien Ramaekers, and Inneke Van Nieuwenhuyse. The vehicle routing problem: State of the art classification and review. *Computers & Industrial Engineering*, 99:300–313, 2016.
3. Francois Chollet. Deep learning with python. 2018.
4. Jan Christiaens and Greet Vanden Berghe. Slack induction by string removals for vehicle routing problems. 2018.
5. Nicos Christofides. The vehicle routing problem. *Combinatorial optimization*, 1979.
6. Geoff Clarke and John W Wright. Scheduling of vehicles from a central depot to a number of delivery points. *Operations research*, 12(4):568–581, 1964.
7. George B Dantzig and John H Ramser. The truck dispatching problem. *Management science*, 6(1):80–91, 1959.
8. Bruce L Golden, Subramanian Raghavan, and Edward A Wasil. *The vehicle routing problem: latest advances and new challenges*, volume 43. Springer Science & Business Media, 2008.
9. Chris Groër, Bruce Golden, and Edward Wasil. A library of local search heuristics for the vehicle routing problem. *Mathematical Programming Computation*, 2(2):79–101, 2010.
10. Keld Helsgaun. An effective implementation of the lin–kernighan traveling salesman heuristic. *European Journal of Operational Research*, 126(1):106–130, 2000.
11. John H Holland. Genetic algorithms. *Scientific american*, 267(1):66–73, 1992.
12. Holger H Hoos and Thomas Stützle. *Stochastic local search: Foundations and applications.* Elsevier, 2004.
13. Sahbi Ben Ismail, François Legras, and Gilles Coppin. Synthèse du problème de routage de véhicules. 2011.
14. Philip Kilby, Patrick Prosser, and Paul Shaw. Guided local search for the vehicle routing problem with time windows. In *Meta-heuristics*, pages 473–486. Springer, 1999.
15. Rahma Lahyani, Mahdi Khemakhem, and Frédéric Semet. Rich vehicle routing problems: From a taxonomy to a definition. *European Journal of Operational Research*, 241(1):1–14, 2015.
16. Gilbert Laporte. Fifty years of vehicle routing. *Transportation Science*, 43(4):408–416, 2009.
17. Clément Legrand-Lixon. Création d'une learning heuristic pour résoudre le capacitated vehicle routing problem. 2018.
18. Shen Lin and Brian W Kernighan. An effective heuristic algorithm for the traveling-salesman problem. *Operations research*, 21(2):498–516, 1973.
19. Helena R Lourenço, Olivier C Martin, and Thomas Stützle. Iterated local search. In *Handbook of metaheuristics*, pages 320–353. Springer, 2003.
20. Nenad Mladenović and Pierre Hansen. Variable neighborhood search. *Computers & operations research*, 24(11):1097–1100, 1997.
21. Éric Taillard, Philippe Badeau, Michel Gendreau, François Guertin, and Jean-Yves Potvin. A tabu search heuristic for the vehicle routing problem with soft time windows. *Transportation science*, 31(2):170–186, 1997.
22. Paolo Toth and Daniele Vigo. *The vehicle routing problem.* SIAM, 2002.
23. Christos Voudouris and Edward PK Tsang. Guided local search. In *Handbook of metaheuristics*, pages 185–218. Springer, 2003.

Annex

Probability	Ratio	Accuracy	Precision (“good”, “bad”)	Recall (“good”, “bad”)	Time
1	0.003	0.997	0.650, 0.998	0.407, 0.999	243
0.5	0.007	0.995	0.719, 0.997	0.521, 0.998	142
0.25	0.014	0.992	0.817, 0.994	0.554, 0.998	69
0.1	0.035	0.987	0.825, 0.992	0.762, 0.994	47
0.05	0.069	0.980	0.884, 0.987	0.808, 0.993	27
0.02	0.172	0.972	0.910, 0.982	0.899, 0.985	18
0.01	0.345	0.967	0.933, 0.979	0.939, 0.977	12
0.005	0.678	0.966	0.961, 0.969	0.957, 0.973	11

Table 2. Statistics obtained after training the neural network with different down-sampling. A low probability means that we keep less negative examples. The number of positive examples over the number of negative examples corresponds to the ratio. Then we give the precision and the recall for each class: “good” and “bad”. The time is in seconds.

Algorithm 1: Clarke & Wright algorithm

```
   Input: A CVRP instance
   Output: A solution of the instance
1  Compute the savings of all edges of the instance
2  $s \leftarrow$ Initialization
3  while $\max_{(i,j)} saving(i,j) > 0$ do
4      $(i,j) \leftarrow \text{argmax}_{(i,j)} saving(i,j)$
5      $r_i \leftarrow findRoute(s,i)$                    // Route in which is $i$
6      $r_j \leftarrow findRoute(s,j)$
7      if $r_i$ and $r_j$ can merge then
8          Remove $r_i$ and $r_j$ from $s$
9          Merge $r_i$ and $r_j$ such that $i$ and $j$ are connected in the new route
10         Add the new route in $s$ and put $saving(i,j) = 0$
11 return $s$
```

Variant	$neuralNetwork^{0.005}$	$neuralNetwork^{0.01}$	$neuralNetwork^{0.02}$	Time
Basic	7.84%	7.84%	7.84%	0.30
Probability	20.10%	18.71%	18.95%	0.17
Product	14.11%	14.41%	16.32%	0.28
SubLists				
$l = N$	**7.16%**	**7.22%**	**7.42%**	0.18
$l = N//2$	7.63%	7.48%	7.70%	0.19
$l = N//4$	7.37%	7.63%	7.71%	0.19
$l = N//8$	7.44%	7.67%	7.55%	0.20
$l = 10$	7.70%	7.65%	7.59%	0.20
Best SubLists	**5.60%**	**5.50%**	**5.38%**	1.92

Table 3. Average gap obtained on the 50 first instances of the Uchoa set, with different variants of the CW algorithm and different neural networks. The gap is computed between the solution obtained and the best-known. The time is in seconds. It does not include the training time of the neural network, and the time to predict the probability of each edge (average time of 2.6 s).

Algorithm 2: Proposition of algorihtm to solve the CVRP

```
   Input: A CVRP instance, A neural network
   Output: A solution of the instance
 1 $s^* \leftarrow$ Initial solution                                  // Construction
 2 $s \leftarrow s^*$
 3 $allSolutions \leftarrow \{s^*\}$
 4 while Last improvement obtained during the 2500 last iterations do
 5     $worstEdge \leftarrow argmax_{(i,j)} b(i,j)$
 6     $s \leftarrow$ apply the local search around the $worstEdge$        // Optimization
 7     $allSolutions \leftarrow allSolutions \cup \{s\}$
 8     if $s$ is better than $s^*$ then
 9         $s^* \leftarrow s$
10     if No improvement for 450 iterations then
11         Use the SISR's method to ruin $s^*$ and recreate a new solution with the
             method used to generate an initial solution              // Restart phase
12     if No improvement for 150 iterations and $s^*$ improved for the last time then
13         $s^* \leftarrow$ apply local search on each route                 // Global phase
14     if localOptimum then
15         Change the cost function
16         for 10 iterations do
17             $worstEdge \leftarrow argmax_{(i,j)} b(i,j)$                    // Perturbation
18             Apply a perturbation method on $s$ around $worstEdge$
19         Change penalization function by picking another pair $(\lambda_w, \lambda_c)$
20         Restore the initial cost function
21 Sort $allSolutions$ by increasing cost
22 return $allSolutions$
```

Instance	Optimal	LearnHeuristic-2018			MachineLearning-2019		
		Best	Gap (%)	Time (s)	Best	Gap (%)	Time (s)
A-n32-k05	784	784	**0**	351	784	**0**	45
A-n33-k05	661	661	**0**	381	661	**0**	32
A-n33-k06	742	742	**0**	412	742	**0**	43
A-n34-k05	778	779	0.13	331	778	**0**	37
A-n36-k05	799	799	**0**	464	799	**0**	38
A-n37-k05	669	671	0.15	421	669	**0**	45
A-n37-k06	949	949	**0**	465	949	**0**	31
A-n38-k05	730	730	**0**	361	730	**0**	50
A-n39-k05	822	822	**0**	405	822	**0**	63
A-n39-k06	831	831	**0**	382	831	**0**	23
A-n44-k06	937	937	**0**	500	937	**0**	41
A-n45-k06	944	950	0.63	487	948	0.42	59
A-n45-k07	1146	1149	0.26	500	1148	0.17	42
A-n46-k07	914	914	**0**	498	914	**0**	45
A-n48-k07	1073	1073	**0**	508	1073	**0**	76
A-n53-k07	1010	1014	0.39	515	1011	0.1	65
A-n54-k07	1167	1167	**0**	521	1167	**0**	105
A-n55-k09	1073	1073	**0**	506	1073	**0**	61
A-n60-k09	1354	1354	**0**	547	1354	**0**	101
A-n61-k09	1034	1035	0.09	562	1035	0.09	91
A-n62-k08	1288	1308	1.53	624	1302	1.09	71
A-n63-k09	1616	1627	0.68	641	1627	0.68	108
A-n63-k10	1314	1320	0.45	662	1314	**0**	72
A-n64-k09	1401	1416	1.06	684	1414	0.93	58
A-n65-k09	1174	1176	0.17	704	1178	0.34	48
A-n69-k09	1159	1164	0.43	768	1159	**0**	102
A-n80-k10	1763	1767	0.23	843	1763	**0**	97

Table 4. Results obtained for the set A.

Instance	Optimal	LearnHeuristic-2018			MachineLearning-2019		
		Best	Gap (%)	Time (s)	Best	Gap (%)	Time (s)
B-n31-k05	672	672	**0**	319	672	**0**	42
B-n34-k05	788	788	**0**	371	788	**0**	54
B-n35-k05	955	955	**0**	388	955	**0**	36
B-n38-k06	805	806	0.12	413	805	**0**	35
B-n39-k05	549	549	**0**	436	549	**0**	45
B-n41-k06	829	831	0.24	491	829	**0**	81
B-n43-k06	742	742	**0**	534	742	**0**	53
B-n44-k07	909	910	0.11	463	909	**0**	45
B-n45-k05	751	751	**0**	461	751	**0**	154
B-n50-k07	741	741	**0**	604	741	**0**	51
B-n50-k08	1312	1320	0.61	587	1314	0.15	92
B-n52-k07	747	747	**0**	562	747	**0**	58
B-n56-k07	707	710	0.42	473	707	**0**	55
B-n57-k09	1598	1599	0.06	523	1599	0.06	68
B-n63-k10	1496	1533	2.41	766	1516	1.34	89
B-n64-k09	861	865	0.46	787	862	0.12	72
B-n66-k09	1316	1321	0.38	770	1319	0.23	75
B-n67-k10	1032	1040	0.77	784	1035	0.29	67
B-n68-k09	1272	1289	1.32	666	1288	1.26	90
B-n78-k10	1221	1231	0.57	881	1221	**0**	141

Table 5. Results obtained for the set B.

Instance	Optimal	LearnHeuristic-2018			MachineLearning-2019		
		Best	Gap (%)	Time (s)	Best	Gap (%)	Time (s)
P-n016-k08	450	450	**0**	132	450	**0**	63
P-n019-k02	212	212	**0**	163	212	**0**	43
P-n020-k02	216	216	**0**	203	216	**0**	52
P-n021-k02	211	211	**0**	168	211	**0**	66
P-n022-k02	216	216	**0**	227	216	**0**	97
P-n023-k08	529	529	**0**	205	529	**0**	106
P-n040-k05	458	458	**0**	383	458	**0**	55
P-n045-k05	510	510	**0**	580	510	**0**	58
P-n050-k07	554	554	**0**	596	556	0.36	65
P-n050-k08	631	631	**0**	485	631	**0**	87
P-n050-k10	696	699	0.43	584	701	0.72	79
P-n051-k10	741	741	**0**	479	741	**0**	54
P-n055-k07	568	568	**0**	641	568	**0**	78
P-n055-k10	694	694	**0**	632	698	0.58	67
P-n060-k10	744	750	0.8	513	746	0.27	79
P-n060-k15	968	975	0.72	596	968	**0**	124
P-n065-k10	792	792	**0**	668	792	**0**	119
P-n070-k10	827	839	1.43	715	828	0.12	126
P-n076-k04	593	596	0.50	754	593	**0**	280
P-n076-k05	627	630	0.48	821	628	0.16	270
P-n101-k04	681	686	0.73	1023	683	0.29	371

Table 6. Results obtained for the set P.

Instance	Best-Known	MachineLearning-2019		
		Best	Gap (%)	Time (m)
X-n101-k25	27591	27833	0.88	3.41
X-n106-k14	26362	26443	0.31	3.75
X-n110-k13	14971	14971	**0**	2.48
X-n115-k10	12747	12751	0.03	6.27
X-n120-k06	13332	13342	0.08	10.2
X-n125-k30	55539	56390	1.53	9.83
X-n129-k18	28940	28970	0.10	4.28
X-n134-k13	10916	10938	0.20	13.5
X-n139-k10	13590	13670	0.59	5.62
X-n143-k07	15700	15769	0.44	14.0
X-n148-k46	43448	43765	0.73	6.2
X-n153-k22	21220	22082	4.06	18.17
X-n157-k13	16876	16925	0.29	7.5
X-n162-k11	14138	14264	0.89	10.1
X-n167-k10	20557	20622	0.32	11.3
X-n172-k51	45607	45988	0.84	13.1
X-n176-k26	47812	48912	2.30	25.3
X-n181-k23	25569	25718	0.58	6.63
X-n186-k15	24145	24326	0.75	11.7
X-n190-k08	16980	17033	0.31	26.9
X-n195-k51	44225	44603	0.85	17.8
X-n200-k36	58578	60018	2.46	14.83

Table 7. Results obtained for the set X (1).

Instance	Best-Known	MachineLearning-2019		
		Best	Gap (%)	Time (m)
X-n204-k19	19565	19689	0.63	16.0
X-n209-k16	30656	30909	0.83	25.7
X-n214-k11	10856	11036	1.66	17.8
X-n219-k73	117595	117796	0.17	16.9
X-n223-k34	40437	40752	0.78	14.3
X-n228-k23	25742	25922	0.70	25.4
X-n233-k16	19230	19680	2.34	56.9
X-n237-k14	27042	27129	0.32	45.2
X-n242-k48	82751	83470	0.87	22.9
X-n247-k50	37274	38832	4.27	31.7
X-n251-k28	38684	39070	0.99	20.4
X-n256-k16	18839	18994	0.82	23.0
X-n261-k13	26558	27145	2.21	80.2
X-n266-k58	75478	76869	1.84	22.9
X-n270-k35	35291	35618	0.93	18.8
X-n275-k28	21245	21539	1.38	20.1
X-n280-k17	33503	33840	1.01	50.2
X-n284-k15	20215	20786	2.82	64.6
X-n289-k60	95151	96365	1.28	41.0
X-n294-k50	47161	47859	1.48	17.6
X-n298-k31	34231	34420	0.55	31.8
X-n303-k21	21744	21980	1.09	40.4
X-n308-k13	25859	26258	1.54	93.2
X-n313-k71	94044	95862	1.93	80.9
X-n317-k53	78355	78527	0.22	37.8
X-n322-k28	29834	30454	2.08	31.1
X-n327-k20	27532	28025	1.79	53.9
X-n331-k15	31102	31589	1.57	85.7

Table 8. Results obtained for the set X (2).