Computers & Operations Research 69 (2016) 40–47

Contents lists available at ScienceDirect

Computers & Operations Research

journal homepage: www.elsevier.com/locate/caor

Technical note: Split algorithm in O(n) for the capacitated vehicle routing problem

Thibaut Vidal

Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro (PUC-Rio) Rua Marquês de São Vicente, 225 – Gávea, Rio de Janeiro – RJ 22451-900, Brazil

ARTICLE INFO

Available online 11 December 2015

Keywords:
Vehicle routing problem
Large neighborhood search
Split algorithm
Cluster-first route-second heuristic

ABSTRACT

The Split algorithm is an essential building block of route-first cluster-second heuristics and modern genetic algorithms for vehicle routing problems. The algorithm is used to partition a solution, represented as a giant tour without occurrences of the depot, into separate routes with minimum cost. As highlighted by the recent survey of Prins et al. [18], no less than 70 recent articles use this technique. In the vehicle routing literature, Split is usually assimilated to the search for a shortest path in a directed acyclic graph $\mathcal{G}$ and solved in $\mathcal{O}(nB)$ using Bellman's algorithm, where n is the number of delivery points and B is the average number of feasible routes that start with a given customer in the giant tour. Some linear-time algorithms are also known for this problem as a consequence of a Monge property of $\mathcal{G}$. In this paper, we highlight a stronger property of this graph, leading to a simple alternative algorithm in $\mathcal{O}(n)$. Experimentally, we observe that the approach is faster than the classical Split for problem instances of practical size. We also extend the method to deal with a limited fleet and soft capacity constraints.

1. Introduction

The algorithm of Prins [16] was an important milestone for the vehicle routing problem (VRP): it was the first hybrid genetic algorithm with local search to outperform classical tabu searches at a time when such methods were predominant. One main ingredient of its success was its approach to solution representation and recombination. Until the 2000s, combining two solutions was considered a difficult task, because simple crossover operators had a tendency to produce infeasible and unbalanced routes. To meet this challenge, [16] represented the solution as a permutation of visits, a "giant tour", and relied on a dynamic-programming-based decoder, called *Split*, which optimally inserts depot visits to obtain complete solutions. This makes it possible to efficiently use classical crossovers for permutations, since the Split algorithm is in charge of route delimitations, and the capacity constraints are implicitly managed during solution decoding.

Ten years on, the literature on population-based methods for VRPs has grown extensively. Efficient GAs with a complete solution representation and more advanced crossover operators now exist for the capacitated VRP (e.g., [15]), a sign that the Split algorithm is useful but not a necessity. Nevertheless, the approach of [16] remains simple and generic. The representation as a giant tour enables to significantly reduce the number of distinct individuals in the GA, and many side constraints and auxiliary decisions of VRP variants, such as capacity and duration limits, time windows [22], choices of depots [8], vehicle types [9], or profitable customers in each route [25] can be handled in the Split algorithm rather than in the crossover. As such, Split has led to successful heuristics for a large number of problems, as surveyed in [9,14,18,21].

The computational efficiency of the Split algorithm for the Capacitated VRP (CVRP) is the subject of this paper. The CVRP aims to find minimum-distance routes to service n customer locations with respective demands $q_1, \ldots, q_n$, using a fleet of up to m vehicles of capacity Q located at a central depot. Here, we consider that an input solution is given, represented as a giant tour $(1, \ldots, n)$ (w.l.o.g., the visits are re-indexed by order in the tour). Let $d_{i,i+1}$ be the distance between two successive customers, and d_{0i} and d_{i0} be the distances from and to the depot. All distances and demand quantities are assumed to be non-negative. The objective of Split is to partition the giant tour into m disjoint sequences of consecutive visits. Each such sequence is associated to a route, which originates from the depot, visits its respective customers, and returns to the depot. The total distance of all routes should be minimized. Note that the algorithms of this paper do not require the symmetry of the distance matrix or the triangle inequality.

E-mail address: vidalt@inf.puc-rio.br

http://dx.doi.org/10.1016/j.cor.2015.11.012
0305-0548/

Classically, the Split algorithm is reduced to a shortest path problem between the nodes 0 and n of an acyclic graph $\mathcal{G}=(\mathcal{V},\mathcal{A})$, where $\mathcal{V}=(0,\ldots,n)$, and $\mathcal{A}$ contains one arc (i,j) with cost $c(i,j)=d_{0,i+1}+\sum_{k=i+1,\ldots,j-1}d_{k,k+1}+d_{j,0}$ for any feasible route visiting customers $i+1$ to j. In the literature, the shortest path is obtained in $\mathcal{O}(nB)$ via a variant of Bellman's algorithm, where B is the average out-degree of a node in $\{0,\ldots,n-1\}$, i.e., the average number of feasible trips from one node of the giant tour [2,16]. Moreover, for a limited fleet of m vehicles, the propagation of the labels can be iterated to produce a shortest path with at most m arcs in $\mathcal{O}(nmB)$. Such complexity is suitable for most medium-scale applications. However, Split can become a computational bottleneck for large problems with many deliveries per route, when used iteratively in a metaheuristic.

To meet this challenge, we will introduce a new *Split* algorithm in $\mathcal{O}(n)$. Note that some linear-time algorithms are already known for this shortest path (see [3,5] as the graph $\mathcal{G}$ satisfies the Monge property:

$$c(i_1,j_1)+c(i_2,j_2)\leq c(i_1,j_2)+c(i_2,j_1) \quad \text{for all } 0\leq i_1<i_2<j_1<j_2\leq n \quad \text{such that } (i_1,j_2)\in\mathcal{A}, \tag{1}$$

where $c(i,j)$ is the cost of an arc (i,j). So far, these methods were not applied in the VRP literature.

In this paper, we propose a simpler alternative which uses the fact that the auxiliary graph $\mathcal{G}$ satisfies the following stronger property:

$$\text{for all } 0\leq i_1<i_2<n, \text{there exists } K\in\mathbb{R} \quad \text{such that } c(i_1,j)-c(i_2,j)=K \text{ for all } j>i_2 \text{ such that } (i_1,j)\in\mathcal{A}. \tag{2}$$

We show that Property (2) can be used to eliminate dominated predecessors and retain only good candidates, leading to a very simple labeling algorithm in $\mathcal{O}(n)$ which performs well in practice and can be efficiently used in VRP metaheuristics. The approach is also extended to produce a solution of the Split problem with a limited number of vehicles in $\mathcal{O}(nm)$, and with soft capacity constraints in $\mathcal{O}(n)$.

Finally, we compare the practical CPU time of the proposed method with that of the classical Bellman-based algorithm, using giant tours built from TSP instances. These instances contain from $n=29$ to 71,009 nodes, and the number of deliveries per route ranges from 4 to 4000. The linear approach appears to be faster in most cases, with speedup factors ranging from 0.8 to 400. The largest speedups are achieved for instances with many deliveries per route, which can occur in courier delivery, refuse collection, and meter reading applications.

The remainder of this paper recalls the Bellman-based Split algorithm in Section 2, introduces the proposed linear Split in Section 3, discusses its generalization to limited fleets and soft capacity constraints in Section 4, and reports our computational experiments in Section 5 To facilitate the use of these algorithms in future generations of heuristics, a C++ implementation of the methods of this paper is available at https://w1.cirrelt.ca/~vidalt/en/VRP-resources.html.

2. Bellman-based split algorithm

Split is traditionally based on a simple dynamic programming algorithm, which enumerates the nodes in topological order and, for each node t, propagates its label to all successors i such that $(t,i)\in\mathcal{A}$. The presentation in Algorithm 1 is similar to that of [16]. The arc costs are not preprocessed but directly computed in the inner loop. This specific algorithm was used as a benchmark in our computational experiments in Section 5.

Algorithm 1. Classical Split algorithm.

```
1  p[0] ← 0
2  for t = 1 to n do
3  |  p[t] ← ∞
4  for t = 0 to n − 1 do
5  |  load ← 0
6  |  i ← t + 1
7  |  while i ≤ n and load + q_i ≤ Q do
8  |  |  load ← load + q_i
9  |  |  if i = t + 1 then
10 |  |  |  cost ← d_{0,i}
11 |  |  else
12 |  |  |  cost ← cost + d_{i−1,i}
13 |  |  if p[t] + cost + d_{i0} < p[i] then
14 |  |  |  p[i] = p[t] + cost + d_{i0}
15 |  |  |  pred[i] = t
16 |  |  i ← i + 1
```

At the end of each iteration t (lines 5–16 of Algorithm 1), $p[t]$ contains the cost of a shortest path from 0 to t. The array of predecessors *pred* is maintained throughout the search so that we can retrieve the solution at the end of the algorithm.

3. Split in linear time

This section will introduce a more efficient Split algorithm. As in the classical Split, the arc costs of the underlying graph are not pre-processed. We will describe, in turn, some auxiliary data structures, the data for a numerical example, and the proposed algorithm.

Preliminaries: We define for $i\in\{1,\ldots,n\}$ the cumulative distance $D[i]$ and cumulative load $Q[i]$ as follows:

$$D[i]=\sum_{k=1}^{i-1}d_{k,k+1} \tag{3}$$

$$Q[i]=\sum_{k=1}^{i}q_k. \tag{4}$$

These values can be preprocessed and stored in $\mathcal{O}(n)$ at the beginning of the algorithm. For $i<j$, the cost $c(i,j)$ of an arc (i,j) is the cost of leaving the depot, visiting customers $(i+1,\ldots,j)$, and returning to the depot, computed as

$$c(i,j)=d_{0,i+1}+D[j]-D[i+1]+d_{j,0}, \tag{5}$$

and the arc (i,j) exists if and only if the route is feasible, i.e., $Q[j]-Q[i]\leq Q$.

Our algorithm also relies on a double-ended queue, denoted Λ, that supports the following operations in $\mathcal{O}(1)$:

front–accesses the oldest element in the queue;
front2–accesses the second-oldest element in the queue;
back–accesses the most recent element in the queue;
push_back–adds an element to the queue;
pop_front–removes the oldest element in the queue;
pop_back–removes the newest element in the queue.