Available online at www.sciencedirect.com

ScienceDirect

Transportation Research Procedia 78 (2024) 361–368

Transportation Research Procedia

www.elsevier.com/locate/procedia

25th Euro Working Group on Transportation Meeting (EWGT 2023)

Enhanced bi-directional dynamic programming algorithm for the resource constrained shortest path problem

Matteo Salani[a,]* Saverio Basso[a] Giovanni Righini[b]

[a]*Dalle Molle Institute for Artificial Intelligence - IDSIA USI/SUPSI, Lugano, Switzerland*
[b]*Università degli Studi di Milano, Milan, Italy*

Abstract

In this paper we propose an enhanced bi-directional dynamic programming algorithm for the Resource Constrained Shortest Path, Righini and Salani (2006). In particular we study the join procedure, one of the time consuming components of the algorithm, and propose a dominance based join that uses binary search and Pareto frontier exploration. The proposed method can be used in both node-join and arc-join procedures. The computational experiments are performed on two classes of problems both describing complete graphs with negative cost cycles. On the first set, with a single resource consumption, the proposed procedure slightly improves the overall computational time with savings up to 8.4% for the B class and just 0.4% for the most time consuming class M. On the second set, with multiple resources consumption, the proposed procedure exhibits a remarkable improvement with an average speedup of 39.2%.

Keywords: Resource Constrained Shortest Path; Dynamic Programming

1. Introduction

We consider Resource Constrained Shortest Path Problem (RCSPP), a fundamental combinatorial problem that appears in most challenging and practically relevant decision problems in transportation, telecommunication, and scheduling. RCSPPs also appear as subproblems in routing and workforce planning problems solved by column generation, Desaulniers et al. (2005).

The RCSPP is defined on a graph *G(N, A)*, which we assume directed, composed of a node set N and an arc set A. The problem asks to find a minimum cost elementary path, i.e. a finite sequence of consecutive arcs in which every

* Corresponding author. Tel.: +41.58.666.66.70.
E-mail address: matteo.salani@supsi.ch

10.1016/j.trpro.2024.02.046

node $n \in N$ appears at most once, from a source node $s \in N$ to a destination node $d \in N$. The cost is accumulated when traversing arcs along the path. We remark that no assumptions are taken on the cost of the arcs and the graph may possess negative cost cycles.

The RCSPP problem has one or more resources that are consumed while traversing arcs. For example, elapsed time, transported load, etc., and their availability is constrained. More involving resources can be considered, but for the sake of simplicity we limit our discussion to monotone accumulated resource consumption.

The RCSPP appears in a large variety of real-life applications such as vehicle and crew scheduling problems, Desaulniers et al. (1998); Haase et al.(2001), rostering, Gamache et al. (1999), military aircraft management systems, Zabarankin et al.(2002), railroad management, Halpern and Priess (1974), telecommunication network design, Cabral et al.(2007), green vehicle routing problems, Erdogan and Miller-Hooks (2012) and many others. Surveys on the RCSPP and related solution methods have been published by Irnich and Desaulniers (2005), Pugliese and Guerriero (2013) and Madkour et al. (2017).

The most effective algorithms for the RCSPP rely on dynamic programming methods, Mehlhorn and Ziegelmann, (2000). Those are based on labels that encode partial path information from the source node to another node of the network. In these algorithms, propagation is restricted to Pareto-optimal labels associated at every node. Some notable improvements can be obtained by using bi-directional search, Righini and Salani (2006), by propagating labels from both the source to the destination and backward, from the destination to the source, and by joining partial paths. Recent techniques exploit bucket based approaches, Pecin et al. (2017); Sadykov et al. (2020) with the effect of decreasing the number of comparisons for dominance, resulting in significant improvements in running time.

A more complex variant of the problem, the Resource Constrained Elementary Shortest Path Problem (RCESPP), arises when negative cost cycles appears in the network. The problem requires to find elementary paths, i.e. with no repeating nodes. For example in Orienteering problems, Golden et al. (1987); Gunawan et al. (2016), prize collection Travelling Salesman Problems, Laporte and Martello (1990); Feillet et al. (2005) and pricing problems arising in column generation applied to routing problems, Desaulniers et al. (2005).

While approaches based on Branch-and-Cut exist, Jepsen et al. (2008), the most effective algorithms are based on relaxation solved by dynamic programming. Among the top techniques, in Righini and Salani (2008) the authors propose to progressively enlarge the state space, by iteratively populating a set of nodes necessary to compute an elementary solution. Similarly, ng-path relaxations have been proposed in Baldacci et al. (2011): they define, for each node, a set of neighbor nodes that must not contain cycles. More recently, variants of such relaxations based on managing sets of neighbor arcs instead of nodes have been introduced in Bulhoes et al. (2018) and Costa et al. (2021). Other relevant approaches such as Irnich and Villeneuve (2006) and Desaulniers et al. (2008), rely on forbidding cycles of small length and on relaxing elementarity requirements for specific nodes.

2. Exact algorithm for the RCESPP

The exact algorithm used in this paper is based on the bi-directional dynamic programming algorithm proposed by Righini and Salani (2006). In Dynamic Programming (DP), states represent partial paths from the source node s to the nodes in the network. Different states can be associated with the same node and they correspond to different partial paths. DP iteratively extends states until no further extensions are possible. Each state is encoded in a label, in bi-directional DP called forward and backward labels. A forward label associated with node $i \in N$ is a tuple:

$$l_i^f = (i, c_i, S, R) \tag{1}$$

where i is the last node visited in the partial path, c_i is the accumulated cost, S is a binary vector that keep tracks of the visited nodes in the partial path and R is the so called resource vector that keeps track of the consumption of each resource. Backward labels are similar and correspond to paths from nodes to the destination d. Labels ending at the same node i can be compared and those that will not lead to an optimal solution, i.e. those with a larger cost and larger consumption of resources, are said to be *dominated* and can be safely discarded.

The DP algorithm extends all feasible non dominated forward and backward labels. The extension of a forward label corresponds to appending an additional arc (i, j) to a path from s to i, obtaining a path from s to j, while the extension of a backward label corresponds to pre-pending an additional arc (j,i) to a path from i to d, obtaining a path from j to d. Labels are stored in convenient data structures, referred as *label pools*.

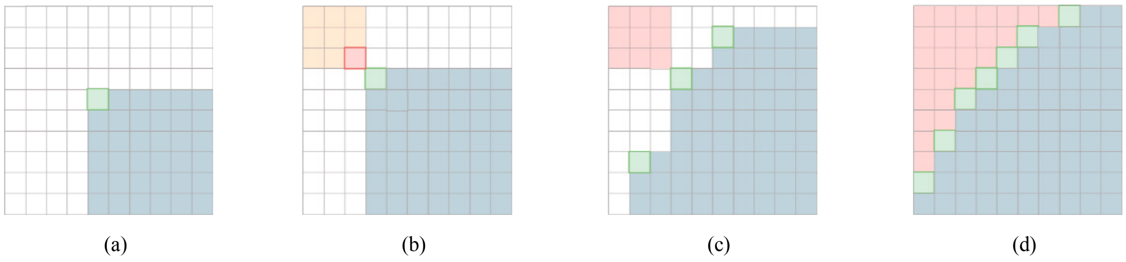

Fig. 1. Illustration of the pareto-join procedure. (a) The binary search explores the area. A feasible join is found and the joins in the shaded area are identified as suboptimal. (b) The binary search converges. (c) The binary search is recursively called on the unexplored joins in the white areas. (d) The procedure terminates when no potential join areas left unexplored.

Once extension is completed, forward and backward are joined to produce complete paths from node s to node d, ensuring that the final path contains no cycles nor violates resource constraints. Indeed, among the feasible paths that are generated by the join operation, we compute the optimal path: its existence is guaranteed by the domination criteria. The join operation can be performed considering the sets of forward and backward labels ending at a node i (node join) or, for any pair of nodes (i, j), the set of forward labels ending at node i and the set of backward labels starting from node j (arc join).

Let Fw and Bw be the set of forward and backward labels, respectively. Regular implementations check for label joining by inspecting the label pools extensively starting from lower cost labels, although it is less likely that a low cost forward label combines feasibly with a low cost backward label. We propose an enhanced join method that we call *pareto-join* and that works for both types of join procedures (node and arc join) and that aims at minimizing the number of pairwise feasibility checks. We assume that forward and backward labels are ordered by non-decreasing cost so that we can index the $i-th$ forward label and $j-th$ backward label to form the joining pair (fw_i, bw_j). The optimal solution, if exists, lies on the Pareto frontier of the non dominated feasible label pairs.

We want to find the Pareto-frontier as quickly as possible. We illustrate the procedure in Figure 1 in which the set of label pairs is explored with binary search to find a feasible solution. Any feasible pair (fw_i, bw_j) dominates all pairs (fw_k, bw_z) with $k \geq i$ and $z \geq j$. When the RCSPP is defined on a single resource and an infeasible pair (fw_i, bw_j) is found, then all pairs (fw_k, bw_z) with $k \leq i$ and $z \leq j$ are infeasible as well. Instead, for multi-resource RCSPPs we lose this property and an infeasible label pair does not guarantee that cheaper and feasible label pairs do not exist.

Figures 2 and 3 report the pseudo-code of the Pareto-Join algorithm. The algorithm is recursive and its presentation is divided in the search phase in figure 2 and the recursive calls phase in figure 3. The algorithm searches for the optimal feasible join of two labels belonging to the ordered sets of labels Fw and Bw, it also receives the value of the best feasible solution found so far and possibly updates it.

Preprocessing. The algorithm first tests the potential of the entire set of labels by performing a preprocessing step (lines 6-7). Then, three special cases are evaluated: when there are just one forward and one backward labels and their join is feasible the process can immediately terminate (lines 10-12). When there is just one forward (resp. backward) label, the join is tested with all the labels in the opposite direction using a linear search. As soon as the pair is suboptimal (line 17 and 26) or the join is feasible (line 19 and 28), the process can immediately terminate.

Binary Search. The binary search procedure starts at line 33 and to explore it the algorithm uses two pairs of indexes *(s_fw, e_fw) (s_bw, e_bw)* (lines 33-34). The region is explored by binary search (lines 38-39). If the current pair *(Fw[i], Bw[j])* is suboptimal, then all pairs *(Fw[k], Bw[z])* with $k \geq i$ and $z \geq j$ are suboptimal and the search region is reduced decreasing the upper limits (lines 41-42). Some special care is needed when the boundary of the region is reached, the procedure anyway guarantees that at each iteration the search region is reduced. Similarly, when a feasible pair is found then a new incumbent solution is found too. Its value is updated and the search region is reduced decreasing the upper limits (lines 45-46). Instead, if the current pair is not suboptimal but infeasible, the search region is reduced increasing the lower limits (lines 48-49). The main while loop converges as at each iteration the search region is decreased by at least one unit. The loop terminates when *s_fw = e_fw* and *s_bw = e_bw*. The last label pair in *(Fw[s_fw], Bw[s_bw])* is either the best feasible solution found, the last suboptimal solution (regardless if feasible or not) or an unfeasible label pair.

Recursive calls. Figure 3 reports how the remaining search space is split in portions according the the status of the last evaluated label pair. The recursive calls consider subsets of the *Fw* and *Bw* sets by defining the two new extremes using the following notation: *Fw[i:j]* means that the subset of forward labels from the index *i* to the index *j* is considered in the recursive call. When the last pair is feasible or suboptimal two recursive calls are made (lines 6-9). When the join on the last pair is infeasible but the cost is lower than the current incumbent solution, some more involving recursive calls are necessary. The procedure guarantees that no potential feasible pair is lost during the search.

3. Implementation

The algorithm has been implemented using *PathWise*, a flexible (soon-to-be) open-source library for the solution of the RCSPP, Salani and Basso (2023). PathWise can solve a variety of standard RCSPPs with an off-the-shelf implementation of state-of-the-art algorithms and allows experienced users to develop their algorithmic components of PathWise while taking advantage of the framework thanks to clear interfaces and well-defined hook points.

Pathwise implements several relaxation schemes such as Decremental state space relaxation, DSSR, Righini and Salani (2008) and *ng-path* relaxation, Baldacci et al. (2011) as well as hybridizations inspired by Martinelli et al. (2014) and different techniques that guarantee complete or partial elementarity, i.e. techniques that return solutions where cycles are forbidden on all or subsets of nodes, respectively. All these relaxations share the idea that only a fraction of the nodes are relevant to compute the optimal solution without cycles or a strong relaxation with cycles and, in the label definition, the vector *S* is restricted to those nodes only. In iterative algorithms, such as DSSR and its hybridizations, the set is empty at the start of the algorithm and it is iteratively enlarged until a solution without cycles is found. Attempts to initialize the set *S* have been explored in Righini and Salani (2009).

Pathwise implements an ad-hoc technique called *semi-dynamic half way point* inspired by the work recently proposed by Tilk et al. (2017). The main idea is to adjusts the critical resource splitting threshold during the label extension procedure in order to keep the dimensions of the forward and backward sets of labels as balanced as possible.

PathWise has a flexible architecture (see figure 4) built around a central solver unit that manages and interacts with other 5 major modules: configuration, problem, algorithm, solution and data collection. In order to implement the ParetoJoin algorithm we extended the unit called *Label Manager* in the algorithmic module. This module unit handles labels, while performing core operations and providing encapsulation.

4. Computational Experiments

We perform computational experiments on 2 classes of problems including both instances found in the literature and newly generated ones. Both classes represent cyclic problem, meaning that each instance present cycles of negative costs.

SPPRCLIB. The first dataset consists of 45 instances with a single capacity constraint. They present a complete graph with up to 262 nodes, and positive distances on arcs and negative costs on nodes (i.e., prizes), making the problem cyclic. This dataset was taken from the repository found in (SPPRCLIB, 2008), where instances were generated from VRP problems. They are organized in subsets A, B, E, G, M and P, that differ in how nodes are positioned and possibly clustered, along with different capacities. A detailed description of these sets can be found in (Uchoa et al., 2017).

Prize Collecting (PC). The second dataset, was generated ad-hoc starting from CVRPLIB instances (CVRPLIB, 2014). It consists of 48 instances with multiple resource consumptions: the problem has two capacity bounds, a node limit and time windows, therefore it is a multi-resource problem. Graphs are complete and up to 100 nodes. In this case, every arc of the network presents a negative distance and the optimal solution maximizes the collection of prizes within feasible resource consumption. Similar problems are present in the literature: we cite, as a reference, arc orienteering problems (Gavalas et al., 2015) and prize collecting TSPs (Balas, 1989).

PathWise is developed in C++20. Compilation was performed through GCC 11.3 with the “O3” optimization flag and the experiments were executed on a machine running Kubuntu 22.04, with an eight-core Intel i9-11900 @ 2.50

GHz and 32GB RAM. We compare the proposed Pareto Join procedure with the currently best performing join procedure available in PathWise called *Ordered Join*.

```
ParetoJoin (Fw, Bw, incumbent)
 // Fw: set of forward labels ordered by non decreasing cost
 // Bw: set of backward labels ordered by non decreasing cost
 // incumbent: best solution found so far

 if cost(Fw[1], Bw[1]) >= incumbent
    return // Any possible join of Fw and Bw labels is suboptimal

  // Join single labels
  if Fw.size == 1 and Bw.size == 1 and isJoinFeasible(Fw[1] , Bw[1])
    incumbent = cost(Fw[1], Bw[1])
    return

  // Join with single forward label
  if Fw.size == 1
    for (j in 1..Bw.size)
       if cost(Fw[1], Bw[j]) >= incumbent
          return // All remaining joins are suboptimal
       if isJoinFeasible(Fw[1], Bw[j])
          incumbent = cost(Fw[1], Bw[j])
          return // Found the best possible join, all the remaining joins are suboptimal

  // Join with single backward label
  if Bw.size == 1
    for (i in 1.. Fw.size)
       if cost(Fw[i], Bw [1]) >= incumbent
          return // All remaining joins are suboptimal
       if isJoinFeasible(Fw[1], Bw[i])
          incumbent = cost(Fw[1], Bw[1])
          return // Found the best possible join, all the remaining joins are suboptimal

   // Sets Fw and Bw have at least two labels each
   s_fw = s_bw = 1
   e_fw = Fw.size ; e_bw = Bw.size

   // Perform binary search
   while ( s_fw < e_fw or s_bw < e_bw )
     i = floor(( s_fw + e_fw ) / 2)
     j = floor(( s_bw + e_bw ) / 2)
     if cost(Fw[i], Bw[j]) >= incumbent
       if ( s_fw != e_fw ) e_fw = i - 1
       if ( s_bw != e_bw ) e_bw = j - 1
     else if isJoinFeasible(Fw[i], Bw[j])
       incumbent = cost (Fw[i], Bw[j])
       if ( s_fw != e_fw ) e_fw = i - 1
       if ( s_bw != e_bw ) e_bw = j - 1
     else
       if ( s_fw != e_fw ) s_fw = i + 1
       if ( s_bw != e_bw ) s_bw = j + 1

   // Test the status of the last pair of labels
   if cost(Fw[ s_fw ], Bw[ s_bw ]) >= incumbent
     feasible_or_suboptimal = true
   else if isJoinFeasible(Fw[ s_fw ], Bw[ s_bw ])
     incumbent = cost(Fw[ s_fw ], Bw[ s_bw ])
     feasible_or_suboptimal = true
   else
     feasible_or_suboptimal = false

(cont.)
```

Fig. 2. Multi resource ParetoJoin algorithm - Searching Phase

```
ParetoJoin (Fw, Bw, incumbent )
(cont.)

  if feasible_or_suboptimal
     // The last tested labels were feasible or suboptimal
     if e_fw > 1
        ParetoJoin (Fw[ 1:e_fw − 1 ], Bw, incumbent)
     if e_bw > 1
        ParetoJoin (Fw[ e_fw:Fw.size ], Bw[ 1:e_bw − 1 ], incumbent)
   else
     // The last tested labels were unfeasible but super − optimal
     if e_fw == Fw. size
        ParetoJoin (Fw[ 1:e_fw − 1 ], Bw , incumbent )
        if e_bw > 1
           ParetoJoin (Fw[ e_fw:e_fw ], Bw [ 1:e_bw − 1 ], incumbent)
     else
        ParetoJoin (Fw[ 1:e_fw ], Bw, incumbent )
        if e_fw < Fw.Size
           ParetoJoin (Fw[ e_fw + 1:Fw.size ], Bw[ 1:e_bw ], incumbent)
```

Fig. 3. Multi resource ParetoJoin algorithm - Recursive calls

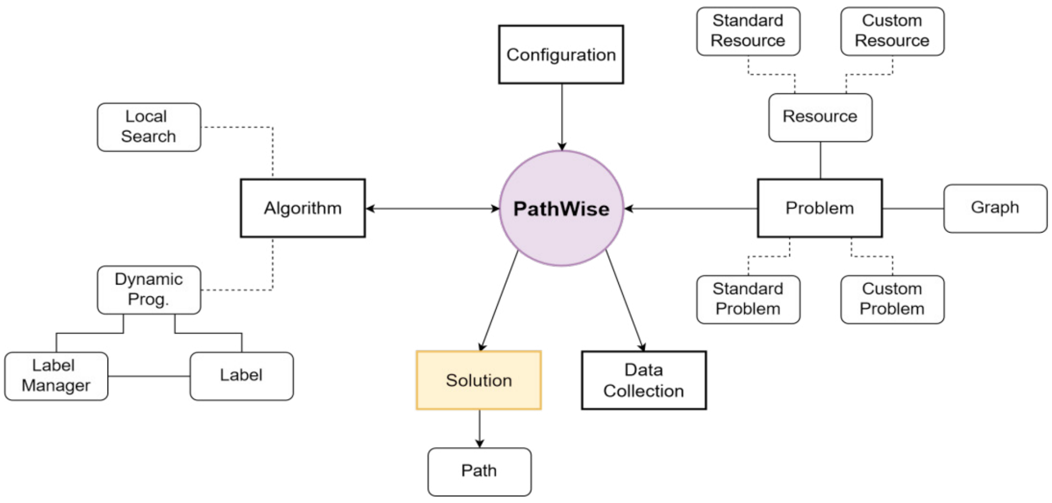

Fig. 4. PathWise architecture

Table 1 reports experimental results related to SPPRCLIB instances. For each class of instances we report the averages over the set. We detail computational overall time dedicated to the join procedure and the global computational time to converge to proven optimality. We observe that the pareto join procedure dominates the ordered join procedure and is 76.6% faster on average. This major speedup only slightly reflect on overall computational times with an average speedup of only 0.7%. This is due to the fact that the time taken by the join procedure represents a small fraction of the overall computational time. Three instances do not converge within 1h of computation and the pareto-join procedure does not help to reach convergence. The set G is composed by just one instance which did not converge in 1h of computation. Therefore, this set does not appear in the table.

Table 2 reports experimental results related to PC instance. The table is organized as table 1. We observe that the overall join time for this set of instances constitutes the largest contribution. We observe that the proposed pareto-join procedure dominates the ordered-join procedure for the majority of instances with a speedup of 45.3% on average. In this set of instances pareto-join is effective in improving the overall computational time, saving 39.2% of global computational time on average.

	Join Time [s]			Global Time [s]		
Instance Class	Pareto Join	Ordered Join	Speedup	Pareto Join	Ordered Join	Speedup
A	0.018	0.044	60.4%	0.976	1.022	4.5%
B	0.049	0.131	62.9%	1.142	1.247	8.4%
E	0.007	0.012	41.0%	0.995	1.007	1.2%
M	3.254	14.169	77.0%	283.674	284.802	0.4%
P	0.066	0.274	75.8%	6.569	6.779	3.1%
Average	0.347	1.481	76.6 %	29.927	30.140	0.7%

Table 1. Comparison on SPPRClib instance classes. We report the average Join Time (s) and the average Global Time (s) along with the percentage speedup.

Instance Class			Join Time [s]			Global Time [s]		
Nodes	Capacity	Node Limit	Pareto Join	Ordered Join	Speedu p	Pareto Join	Ordered Join	Speedup
50	25	8	0.057	0.080	29.0%	0.140	0.164	14.6%
50	25	18	0.106	0.167	36.3%	0.261	0.321	18.8%
50	40	8	4.020	4.240	5.2%	4.811	5.032	4.4%
50	40	18	6.388	12.710	49.7%	9.257	15.594	40.6%
50	25	8	1.131	1.517	25.4%	2.161	2.541	14.9%
50	25	18	1.215	2.842	57.2%	3.450	5.124	32.7%
50	40	8	61.539	62.716	1.9%	70.142	71.324	1.7%
50	40	18	178.419	378.044	52.8%	233.807	433.064	46.0%
Average			31.609	57.790	45.3 %	40.504	66.654	39.2%

Table 2. Comparison on PC instance classes. We report the average Join Time (s) and the average Global Time (s) along with the percentage speedup

5. Conclusions

We proposed an enhancement for a bi-directional dynamic programming algorithm for the RCSPP focusing on the join procedure. We introduce the Pareto-join procedure, a recursive search procedure based on binary search. The proposed procedure has proven effective on two classes of problems. The improvements on the first set with a single resource consumption are modest with an overall speedup of just 0.7%. On the second set with multiple resources consumption, the proposed procedure improved the computational time by 39.2% in average. The paretojoin procedure becomes a candidate to be the standard procedure released with PathWise.

References

Balas, E., 1989. The prize collecting traveling salesman problem. Networks 19, 621–636.

Baldacci, R., Mingozzi, A., Roberti, R., 2011. New route relaxation and pricing strategies for the vehicle routing problem. Operations Research 59, 1269–1283.

Bulhoes, T., Sadykov, R., Uchoa, E., 2018. A branch-and-price algorithm for the minimum latency problem. Computers & Operations Research˜ 93, 66–78.

Cabral, E.A., Erkut, E., Laporte, G., Patterson, R.A., 2007. The network design problem with relays. Eur. J. Oper. Res. 180, 834–844. Costa, L., Contardo, C., Desaulniers, G., Pecin, D., 2021. Selective arc-ng pricing for vehicle routing. International Transactions in Operational Research 28, 2633–2690.

CVRPLIB, 2014. http://vrp.galgos.inf.puc-rio.br [Online; accessed 24-Feb-2023].

Desaulniers, G., Desrosiers, J., loachim, I., Solomon, M.M., Soumis, F., Villeneuve, D., 1998. A unified framework for deterministic time constrained vehicle routing and crew scheduling problems, in: Crainic, T.G., Laporte, G. (Eds.), Fleet Management and Logistics. Springer US, Boston, MA, pp. 57–93.

Desaulniers, G., Desrosiers, J., Solomon, M.M., 2005. Column generation. volume 5. Springer Science & Business Media.

Desaulniers, G., Lessard, F., Hadjar, A., 2008. Tabu search, partial elementarity, and generalized k-path inequalities for the vehicle routing problem with time windows. Transportation Science 42, 387–404.

Erdogan, S., Miller-Hooks, E., 2012. A green vehicle routing problem. Transportation Research Part E: Logistics and Transportation Review˘ 48, 100–114. Select Papers from the 19th International Symposium on Transportation and Traffic Theory.

Feillet, D., Dejax, P., Gendreau, M., 2005. Traveling salesman problems with profits. Transportation Science 39, 188–205.

Gamache, M., Soumis, F., Marquis, G., Desrosiers, J., 1999. A column generation approach for large-scale aircrew rostering problems. Operations Research 47, 247–263.

Gavalas, D., Konstantopoulos, C., Mastakas, K., Pantziou, G., Vathis, N., 2015. Approximation algorithms for the arc orienteering problem.

Information Processing Letters 115, 313–315.

Golden, B.L., Levy, L., Vohra, R., 1987. The orienteering problem. Naval Research Logistics (NRL) 34, 307–318.

Gunawan, A., Lau, H.C., Vansteenwegen, P., 2016. Orienteering problem: A survey of recent variants, solution approaches and applications. European Journal of Operational Research 255, 315–332.

Haase, K., Desaulniers, G., Desrosiers, J., 2001. Simultaneous vehicle and crew scheduling in urban mass transit systems. Transportation Science 35, 286–303.

Halpern, J., Priess, I., 1974. Shortest path with time constraints on movement and parking. Networks 4, 241–253.

Irnich, S., Desaulniers, G., 2005. Shortest Path Problems with Resource Constraints. Springer US, Boston, MA. pp. 33–65.

Irnich, S., Villeneuve, D., 2006. The shortest-path problem with resource constraints and k-cycle elimination for $k \geq 3$. INFORMS Journal on Computing 18, 391–406.

Jepsen, M., Petersen, B., Spoorendonk, S., 2008. A branch-and-cut algorithm for the elementary shortest path problem with a capacity constraint. Technical Report 08/01, DIKU, University of Copenhagen .

Laporte, G., Martello, S., 1990. The selective travelling salesman problem. Discrete Applied Mathematics 26, 193–207.

Madkour, A., Aref, W.G., Rehman, F.U., Rahman, M.A., Basalamah, S.M., 2017. A survey of shortest-path algorithms. CoRR abs/1705.02044. arXiv:1705.02044.

Martinelli, R., Pecin, D., Poggi, M., 2014. Efficient elementary and restricted non-elementary route pricing. European Journal of Operational Research 239, 102–111.

Mehlhorn, K., Ziegelmann, M., 2000. Resource constrained shortest paths, in: Paterson, M.S. (Ed.), Algorithms - ESA 2000, Springer Berlin Heidelberg, Berlin, Heidelberg. pp. 326–337.

Pecin, D., Pessoa, A., Poggi, M., Uchoa, E., 2017. Improved branch-cut-and-price for capacitated vehicle routing. Mathematical Programming Computation 9, 61–100.

Pugliese, L.D.P., Guerriero, F., 2013. A survey of resource constrained shortest path problems: Exact solution approaches. Networks 62, 183–200.

Righini, G., Salani, M., 2006. Symmetry helps: Bounded bi-directional dynamic programming for the elementary shortest path problem with resource constraints. Discrete Optimization 3, 255–273. Graphs and Combinatorial Optimization.

Righini, G., Salani, M., 2008. New dynamic programming algorithms for the resource constrained elementary shortest path problem. Networks 51, 155–170.

Righini, G., Salani, M., 2009. Decremental state space relaxation strategies and initialization heuristics for solving the orienteering problem with time windows with dynamic programming. Computers & Operations Research 36, 1191–1203.

Sadykov, R., Uchoa, E., Pessoa, A., 2020. A bucket graph–based labeling algorithm with application to vehicle routing. Transportation Science 55, 4–28.

Salani, M., Basso, S., 2023. Pathwise: a flexible, open-source library for the resource constrained shortest path. arXiv:2306.08622.

SPPRCLIB, 2008. http://hjemmesider.diku.dk/~spooren/spprclib.htm [Online; accessed 24-Feb-2023].

Tilk, C., Rothenb acher, A.K., Gschwind, T., Irnich, S., 2017. Asymmetry matters: Dynamic half-way points in bidirectional labeling for solving shortest path problems with resource constraints faster. European Journal of Operational Research 261.

Uchoa, E., Pecin, D., Pessoa, A., Poggi, M., Vidal, T., Subramanian, A., 2017. New benchmark instances for the capacitated vehicle routing problem. European Journal of Operational Research 257, 845–858.

Zabarankin, M., Uryasev, S.P., Pardalos, P.M., 2002. Optimal risk path algorithms.