**SUMMARY OF THE MATHEMATICAL BACKGROUND**

# Mathematical formulations

In this section, we present the two formulations used in this article to model and solve the MDVRP, namely a vehicle-flow and a set-partitioning formulation. These two formulations of the problem are exploited at different stages of the algorithm. More precisely, the vehicleflow formulation is used to derive a lower bound quickly and perform variable fixing. The set-partitioning formulation is then considered using the reduced network produced by the variable fixing procedure performed before.

## Vehicle-flow formulation of the MDVRP

For every depot *i* ∈ D and customer *j* ∈ C, let *yij* be a binary variable equal to 1 iff *j* is served by a single-vehicle route departing from depot *i*. For every *e* ∈ *E*, we let *xe* be a binary variable equal to 1 iff edge *e* is used by a vehicle route visiting at least two customers. For a customer set *S* ⊆ C, we let *r*(*S*) = ⌈*d*(*S*)*/Q*⌉ be a lower bound on the number of vehicles needed to serve the customers in *S* due to the capacity constraint, and we let *ρ*(*S*) be a lower bound on the number of vehicles needed to serve the customers in *S* due to the route length constraint. Note that while *r*(*S*) can be computed in constant time, *ρ*(*S*) can be difficult to compute as it involves the solution of a *m*-TSP with route length constraints, which is strongly NP-hard. For a node subset *U* ⊂ *V* we let *δ*(*U*) be the cutset of *U*, or equivalently the subset of edges with exactly one extremity in *U*. For an edge subset *F* ⊆ *E* we also the MDVRP is as follows:P P

define *x*(*F*) = *e*∈*F xe*, and if *F* ⊆ *δ*(D), *y*(*F*) = *e*∈*F ye*. The vehicle-flow formulation of

|  |  |  |  |
| --- | --- | --- | --- |
| min | X*texe* + 2 X *tijyij*  *e*∈*E i*∈D*,j*∈C |  | (1) |
|  | *x*(*δ*{*j*}) + 2*y*(*δ*({*j*}) ∩ *δ*(D)) = 2 | *j* ∈ C | (2) |
|  | *x*(*δ*{*i*}) + 2*y*(*δ*({*i*}) ≤ 2*mi* | *i* ∈ D | (3) |
|  | *x*(*δ*{*S*}) + 2*y*(*δ*(*S*) ∩ *δ*(D)) ≥ 2max{*r*(*S*)*,ρ*(*S*)} | *S* ⊆ C | (4) |
|  | *x*(*δ*(*S*)) ≥ 2[*x*(*δ*({*h*}) ∩ *δ*(D′)) + *x*(*δ*({*j*}) ∩ *δ*(D \ D′))] | *S* ⊆ C*,h,j* ∈ *S*  D′ ⊂ D | (5) |
|  | *ye* ∈ {0*,*1} | *e* ∈ *δ*(D) | (6) |
|  | *xe* ∈ {0*,*1} | *e* ∈ *E* | (7) |

The objective function aims to minimize the total traveling time. Constraints (2) are the degree constraints that impose each customer be visited exactly once. Constraints (3) is the fleet size constraint. They impose that at most *mi* vehicles are used at each depot. Constraints (4) are the capacity and route length constraints. They impose that at least max{*r*(*S*)*,ρ*(*S*)} vehicles are used to visit the customers of set *S*. Constraints (5) are the path constraints. They forbid routes to have the starting and ending points at two different depots. Finally, (6)-(7) impose that variables are indeed binary. This formulation is a particular case of the one introduced by Belenguer et al. [8] for the CLRP, with the only addition of the route length constraint represented by the constants *ρ*.

## Set-partitioning formulation of the MDVRP

For each *i* ∈ D and *j* ∈ C we let *yij* be a binary variable equal to 1 iff customer *j* is served alone in a route departing from depot *i*, with cost equal to 2*tij*. We now let Ω be the set of routes visiting at least two customers and respecting the capacity and route length constraints. For a depot *i* ∈ D, we denote by Ω*i* the subset of routes starting and ending *i*. For each *l* ∈ Ω, we let *θl* be a binary variable equal to 1 iff route *l* is selected, and we denote by *tl* its cost, which is equal to the sum of the traveling times along the edges used by *l*. For each customer *j* ∈ C and route *l* ∈ Ω we let *alj* be the number of times that customer *j* is visited by *l*. For each depot subset *D* ⊆ D and customer subset *C* ⊆ C we let *y*(*D* : *C*) = P*i*∈*D* P*j*∈*C yij*. The set-partitioning formulation of the MDVRP is as follows:

|  |  |  |
| --- | --- | --- |
| min | X*tlθl* + 2 X *tijyij*  *l*∈Ω *i*∈D*,j*∈C | (8) |

X

*aljθl* + *y*(D : {*j*}) = 1 *j* ∈ D (9)

*l*∈Ω

|  |  |  |
| --- | --- | --- |
| X*θl* + *y*({*i*} : C) ≤ *mi*  *l*∈Ω*i* | *i* ∈ D | (10) |
| *yij* ∈ {0*,*1} | *i* ∈ *δ*(D) | (11) |
| *θl* ∈ {0*,*1} | *l* ∈ Ω | (12) |

Once again, the objective function aims to minimize the total traveling cost. Constraints

(9) are the degree constraints that impose each customer be visited exactly once. Constraints (11)-(12) state the binary nature of the variables. Note that the capacity and the route length constraints are embedded into the definition of the route set Ω, and therefore do not need to be explicitely included in the formulation of the problem. For the same reason, the path constraints (5) are also not needed.

# Valid inequalities

In this section we present the valid inequalities used to strengthen both formulations presented earlier. For the first families of inequalities, we refer to them as “weak” because their inclusion does not impose the addition of an extra resouce in the labeling algorithm. For the last five classes of inequalities introduced, we refer to them as “strong”, because their addition imposes the use of additional resource during the labeling algorithm.

## Weak valid inequalities

We call weak valid inequalities to all those inequalities that are valid for formulation (1) and that can be used in both formulations. These inequalities have the particularity that the contribution of their duals to the computation of the reduced costs of paths can be decomposed along the edges defining them, and therefore included in the pricing algorithm without compromising its performance. We consider some of the valid inequalities available in the CVRPSEP package [27], namely the framed capacity inequalities, strengthened comb inequalities, multistar inequalities and hypotour inequalities. We also use some of the inequalities introduced by Belenguer et al. [8], Contardo et al. [12], namely the *y*-capacity cuts, degree constraints and co-circuit inequalities using the separation algorithms of Contardo et al. [12].

## Strong degree constraints

The strong degree constraints (SDC) were originally introduced by Contardo et al. [13] for the CLRP, and they are also valid for the CVRP and the MDVRP. Before presenting the inequality, let us define some notation. Given a customer *j* ∈ C and a route *l* ∈ Ω, we let  be a binary constant equal to 1 iff route *l* visits node *j*. For a given customer *j* ∈ C we define. The SDC associated to node *j* is

*ξθ*(*j*) + *y*(D : {*j*}) ≥ 1*.* (13)

Contardo et al. [13] proved that this constraint imposes partial elementarity on node *j*, this is, that no variable *θl* visiting node *j* twice or more will take a positive value in the solution of the linear relaxation of the set-partitioning formulation.

## *k*-Cycle elimination constraints

We now introduce a new family of valid inequalities that can be seen as a weaker form of the strong degree constraints. Let *k* ≥ 1 be an integer constant. Let *j* ∈ C be a customer and let *l* ∈ Ω be a route. Let us define *νjkl* as the number of times that route *l* visits customer *j* with at least *k* nodes between two consecutive appearances of *j* in the route. In Figure 1 we illustrate by means of an example the behavior of the values  increases. The following *k*-cycle elimination constraint (*k*-CEC) is valid for the MDVRP:

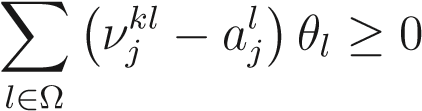
X*νjklθl* + *y*(D : {*j*}) ≥ 1*.* (14)

*l*∈Ω

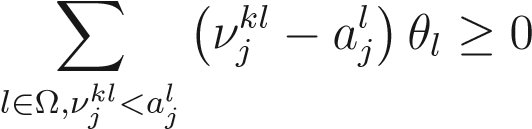
Because *ξjl* ≤ *νjkl* ≤ *alj* it follows that the *k*-CEC is a strengthening of the degree constraint (9) but a weaker form of the SDC (13). The following proposition shows that the *k*-CEC for a given cycle length *k* and customer *j* is effective to forbid a cycle of length *k* or less from visiting customer *j* twice.

**Theorem 3.1.** *Suppose that a k-CEC has been added to problem* (8)*-*(12) *for customer j and cycle length k. All routes l* ∈ Ω *visiting node j twice or more and such that* *will be non-basic in the linear relaxation of problem* (8)*-*(12)*. In particular, no route visiting customer j two consecutive times with k* − 1 *or less intermediate nodes will take a positive value.*

**Proof** If we consider the *k*-CEC and subtract from it the regular degree constraint (9) we obtain



The summation above only has interest for those *l* such that *νjkl < ali*. In that case, it becomes



This last summation implies that *θl* = 0 for all *l* such that *νjkl < alj*.



0

1

2

1

3

4

1

5

1

6

0

Figure 1: Route *l* with = 2 and

## *y*-Strong capacity constraints

The *y*-strong capacity constraints (*y*-SCC) were also introduced by Contardo et al. [13]. Given a customer subset *S* ⊆ C and a route *l* ∈ Ω, we let *ξSl* be a binary constant equal to 1 iff route *l* visits at least one customer in *S*. We define *ξθ*(*S*) = *l*∈Ω *ξSl θl*. Also, let *r*(*S*) = ⌈*d*(*S*)*/Q*⌉ be a lower bound on the number of vehicles that are needed to visit allP customers in *S*, with *d*(*S*) = *i*∈*S di*. Let *S*′ ⊆ *S* be a subset satisfying *r*(*S* \ *S*′) = *r*(*S*). The following *y*-strong capacity constraintsP (*y*-SCC) is valid for the MDVRP:

*ξθ*(*S*) + X X *yij* ≥ 1*.* (15)

*i*∈D *j*∈*S*\*S*′

This inequality dominates the SCC introduced by Baldacci et al. [3] and the *y*-capacity constraints (*y*-CC) introduced by Belenguer et al. [8]. Its addition, however, imposes a modification in the pricing algorithm to properly handle the associated dual variable.

## Strong framed capacity inequalities

The strong framed capacity inequalities (SFCI) are a lifted form of the original FCI and were introduced by Contardo et al. [13]. Given a customer subset *S* ⊆ C (the frame) and a partition of it (*Sk*)*k*∈*K*, let *r*(*S,*(*Sk*)*k*∈*K*) be the solution of the following bin-packing problem. For each set *Sk* with accumulated demand *d*(*Sk*) ≤ *Q*, consider one object of size *d*(*Sk*). For each set *Sk* with accumulated demand *d*(*Sk*) ≥ *Q* consider *nk* = ⌊*d*(*Sk*)*/Q*⌋ objects of size *Q*, plus at most one object of size *d*(*S*)−*Q*×*nk* (this last object does not appear if *d*(*S*) divides *Q*). Set the bins to have capacity *Q*. If *r*(*S,*(*Sk*)*k*∈*K*) *> r*(*S*) then the following inequality is valid for the MDVRP:

*ξθ*(*S*) + X *ξθ*(*Sk*) + 2XX*yij* ≥ X *r*(*Sk*) + *r*(*S,*(*Sk*)*k*∈*K*) (16)

*k*∈*K i*∈D *j*∈*S k*∈*K*

Again, the addition of a SFCI requires a modification of the labeling algorithm during the recursion of the dynamic programming. Indeed, |*K*|+ 1 additional resources are needed to properly handle the dual variable associated to a such constraint.

## Subset-row inequalities

We consider two particular cases of subset-row inequalities [24]. Given a customer subset *C* of size *n* odd (we consider *n* = 3*,*5), for every route *l* ∈ Ω we let *nlC* be the number of times that *l* visits the customers in *C*. The following inequality is a valid subset-row inequality for

MDVRP:

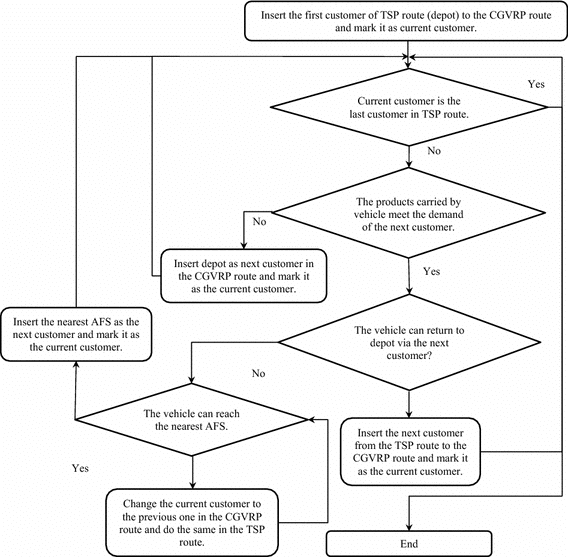
X⌊*nlC/*2⌋*θl* ≤ ⌊*n/*2⌋ (17)

*l*∈Ω

The addition of a subset-row inequality of this form again forces the addition of an additional resource in the dynamic programming recursion.

## Separation algorithms

For the weak constraints, we make use of the separation routines introduced in Lysgaard et al. [28] and Contardo et al. [12]. For the strong constraints, we use the same strategy as in Contardo et al. [13]. Constraints SDC and *k*-CEC are polynomial in number and can be easily separated by simple inspection. To find violated constraints *y*-SCC and SFCI, we verify for each weak constraint *y*-CC and FCI, if the strong version of such inequality is violated, and add it to the problem. Finally, for constraints SRI, we do the following. For *n* = 3, we check for every triplet (*i,j,k*) with *i < j < k* if the corresponding SRI is violated and add it to the problem. For *n* = 5, this same procedure becomes impractical. Thus, we heuristically select the 30 customers with more appearances in the basic solutions of the current linear problem. We then consider all possible 5-tuples restricted to these 30 customers.



CODE STRUCTURE

col-gen-vrptw.py

from utilities import \*

from optimization import \*

from impact import initializePathsWithImpact, computeRouteCost

from time import process\_time

import os

### TODOS ###

# Create Timer to clean the code

# Move file writes of results in a dedicated function

# Take instance and customer number from CLI

# See if it is possible to extend master model each iteration instead of

#   recreate it each time (model.addVar(..., \*column = ...\*))

# Use numpy where possible (use np array for x, y, a, b, ... if possible)

#   -> fun. readData

# Plot routes

######

if \_\_name\_\_ == '\_\_main\_\_':

    INSTANCE\_NAME, INSTANCE\_FILENAME, n = readInstanceN()

    # Read data from file and create distance matrix

    Kdim, Q, x, y, q, a, b = readData(INSTANCE\_FILENAME, n)

    d = createDistanceMatrix(x, y)

    print("Number of customers:", n, "\n")

    print("Start")

    start = process\_time()

    # Initialize routes with IMPACT heuristic and dummy paths

    impactSol = initializePathsWithImpact(d, n, a, b, q, Q)

    routes = impactSol[:]

    #print("Impact solution:", routes)

    impactCost = sum([computeRouteCost(route, d) for route in routes])

    print("Impact cost:", impactCost)

    # A[i,p] = times that path p visits customer i

    A = np.zeros((n, len(routes)))

    c = np.zeros(len(routes))   # routes costs

    addRoutesToMaster(routes, A, c, d)

    rc = np.zeros((n+2,n+2))    # reduced costs

    iter = 1

    while True:

        print("Iter", iter, flush=True)

        # Create master problem model

        masterModel = createMasterProblem(A, c, n, Kdim)

        masterModel.optimize()

        # Compute reduced costs

        constr = masterModel.getConstrs()

        pi\_i = [0.] + [const.pi for const in constr] + [0.]

        for i in range(n+2):

            for j in range(n+2):

                rc[i,j] = d[i,j] - pi\_i[i]

        if not np.where(rc < -1e-9):

            break

        newRoutes = subProblem(n, q, d, a, b, rc, Q)

        # Exit condition

        if not newRoutes:

            break

        for route in newRoutes:

            if route in routes:

                print("\nDUPLICATE PATH\n", flush=True)

                break

        # Add new routes to master problem

        newMat = np.zeros((n, len(newRoutes)))

        newCosts = np.zeros(len(newRoutes))

        addRoutesToMaster(newRoutes, newMat, newCosts, d)

        routes += newRoutes

        c = np.append(c, newCosts)

        A = np.c\_[A, newMat]

        iter += 1

        # Print partial time

        sc = int(process\_time()-start)

        mn = int(sc / 60)

        sc %=  60

        print("Partial time:", mn, "min", sc, "s")

    end = process\_time()

    print("+++RESULTS+++")

    sec = int(end-start)

    min = int(sec / 60)

    sec %=  60

    print("Time Elapsed:", min, "min", sec, "s")

    print("Impact solution cost:", impactCost)

    print("Exact solution cost:", masterModel.getAttr("ObjVal"))

    # Write results on file in directory "results"

    if not os.path.exists(os.path.join(os.getcwd(), "results")):

        os.mkdir(os.path.join(os.getcwd(), "results"))

    filenameOut = os.path.join("results", \

                               "results-"+INSTANCE\_NAME+"-"+str(n)+".txt")

    fout = open(filenameOut, "w")

    fout.write("Impact solution cost: " + str(impactCost) + "\n")

    fout.write("Impact solution: " + str(impactSol) + "\n")

    fout.write("Exact solution cost: "+str(masterModel.getAttr("ObjVal"))+"\n")

    for i in range(len(routes)):

        var = masterModel.getVarByName("y["+str(i)+"]")

        if var.x > 0.:

            print(round(var.x, 3), "   ", routes[i])

            fout.write(str(round(var.x, 3)) + "   " + str(routes[i]) + "\n")

    fout.close()

    # Write generated routes on file for CoverCost heuristic

    if not os.path.exists(os.path.join(os.getcwd(), "routes")):

        os.mkdir(os.path.join(os.getcwd(), "routes"))

    outfile = os.path.join("routes", \

                            INSTANCE\_NAME+"-"+str(n)+"-customers-routes.txt")

    with open(outfile, "w") as fout:

        fout.write(INSTANCE\_NAME + "\n")

        for route in routes:

            fout.write(str(route) + "\n")

COST FUNCTION

coverCost.py

from utilities import \*

from collections import Counter

import os

from sys import exit

def coverCostHeuristic(bestIndex, allRoutes, allCosts, allCoverCost):

    nodes = set(range(1,n+1))

    routes = allRoutes[:]

    costs = allCosts[:]

    coverCost = allCoverCost[:]

    for node in routes[bestIndex][1:-1]:

        nodes.remove(node)

    sol = [routes[bestIndex]]

    solCost = costs[bestIndex]

    # Start loop

    while nodes:

        filteredRoutes = []

        filteredCosts = []

        filteredCoverCost = []

        for i in range(len(routes)):

            feasible = True

            for node in routes[i][1:-1]:

                if not node in nodes:

                    feasible = False

                    break

            if feasible:

                filteredRoutes.append(routes[i])

                filteredCosts.append(costs[i])

                filteredCoverCost.append(coverCost[i])

        if not filteredRoutes:

            return None

        bestIdx = filteredCoverCost.index(max(filteredCoverCost))

        sol.append(filteredRoutes[bestIdx])

        solCost += filteredCosts[bestIdx]

        for node in filteredRoutes[bestIdx][1:-1]:

            nodes.remove(node)

        routes = filteredRoutes[:]

        costs = filteredCosts[:]

    return (sol, solCost)

if \_\_name\_\_ == "\_\_main\_\_":

    # Take in input instance and number of costumers

    INSTANCE\_NAME, INSTANCE\_FILENAME, n = readInstanceN()

    ROUTES\_FILENAME = os.path.join("routes", \

                            INSTANCE\_NAME+"-"+str(n)+"-customers-routes.txt")

    if not os.path.exists(ROUTES\_FILENAME):

        print("No routes generated for this instance and number of customers.")

        print("Run: 'python col-gen-vrptw.py' and input desired instance and" +\

                "number of customers.")

        print("Exit")

        exit(1)

    # Read CG generated routes

    lines = []

    with open(ROUTES\_FILENAME, "r") as fin:

        lines = fin.readlines()

    allRoutes = [list(map(int, line[1:-2].split(", "))) for line in lines[1:]]

    # Discard all routes that visit a node more than one time

    nodes = set(range(1,n+1))

    routes = []

    for route in allRoutes:

        occ = Counter(route)

        if max(occ.values()) == 1:

            routes.append(route)

    # Create distance matrix and routes costs

    Kdim, Q, x, y, q, a, b = readData(INSTANCE\_FILENAME, n)

    d = createDistanceMatrix(x, y)

    costs = []

    for route in routes:

        cost = 0.

        for i in range(len(route)-1):

            cost += d[route[i], route[i+1]]

        costs.append(cost)

    # Find coverage/cost ratio off the routes

    coverCost = [(len(routes[i])-2)/costs[i] for i in range(len(routes))]

    idxBestCoverCost = sorted(coverCost, reverse = True)

    # if len(idxBestCoverCost) > 300:

    #    idxBestCoverCost =idxBestCoverCost[:300]

    # Find indexes of the 10 paths with the best cover cost

    idxBestCoverCost = set([coverCost.index(idxBestCoverCost[i]) \

                            for i in range(len(idxBestCoverCost))])

    solutions = []

    solutionCosts = []

    for bestIndex in idxBestCoverCost:

        info = coverCostHeuristic(bestIndex, routes, costs, coverCost)

        if info:

            solutions.append(info[0])

            solutionCosts.append(info[1])

    solCost = min(solutionCosts)

    sol = solutions[solutionCosts.index(solCost)]

    print("+++RESULTS+++")

    print("Solution cost:", solCost)

    print("Solution:", sol)

MODEL STRUCTURE

ESPmodel.py

import gurobipy as gp

import numpy as np

def computeMaxCost(d, a, b, n):

    # TODO: Use distance mat. to find an equivalent to infinity for subproblem

    return max([b[i] + d[i,j] - a[j] for i in range(n+2) for j in range(n+2)])

def setESPModelFO(model, x\_vars, pi, n, d):

    rc = np.zeros((n+2,n+2))

    for i in range(n+2):

        for j in range(n+2):

            if (i==0) or (i==n+1):

                rc[i,j] = d[i,j]

            else:

                rc[i,j] = d[i,j] - pi[i-1]

    model.setObjective(gp.quicksum(x\_vars[i,j]\*rc[i,j] for i in range(n+2) \

                                                       for j in range(n+2)), \

                       gp.GRB.MINIMIZE)

    return

def createESPModel(d, pi, q, Q, a, b, n):

    M = computeMaxCost(d, a, b, n)

    model = gp.Model("ESPModel")

    x\_vars = model.addVars(n+2, n+2, vtype=gp.GRB.BINARY, name="x")

    s\_vars = model.addVars(n+2, vtype=gp.GRB.CONTINUOUS, name="s")

    setESPModelFO(model, x\_vars, pi, n, d)

    # R0: capacity constraint

    model.addConstr(sum([q[i] \* \

                    gp.quicksum(x\_vars[i,j] for j in range(n+2)) \

                    for i in range(1,n+1)]) <= Q)

    # R1: depot start constraint

    model.addConstr(gp.quicksum(x\_vars[0,j] for j in range(n+2)) == 1)

    # R2: depot finish constraint

    model.addConstr(gp.quicksum(x\_vars[i,n+1] for i in range(n+2)) == 1)

    # R3-R53: flow costraints

    for h in range(1,n+1):

        model.addConstr(gp.quicksum(x\_vars[i,h] for i in range(n+2)) - \

                        gp.quicksum(x\_vars[h,j] for j in range(n+2)) == 0)

    # Time windows contraints

    for i in range(n+2):

        for j in range(n+2):

            if j!=i:

                model.addConstr(s\_vars[i] + d[i,j] - M\*(1-x\_vars[i,j]) <= \

                                s\_vars[j])

    # Service time constraints

    model.addConstrs(s\_vars[i] >= a[i] for i in range(1,n+1))

    model.addConstrs(s\_vars[i] <= b[i] for i in range(1,n+1))

    # Goodsense constraints:

    # Must not exist an arc that connects a customer with himself

    model.addConstr(gp.quicksum(x\_vars[i,i] for i in range(n+2)) == 0)

    # No arc can enter in the first node

    model.addConstr(gp.quicksum(x\_vars[i,0] for i in range(n+2)) == 0)

    # No arc can exit from the last node

    model.addConstr(gp.quicksum(x\_vars[n+1,j] for j in range(n+2)) == 0)

    # model.write("ESPModel.lp")

    return model