Multiple Couriers Planning Problem

Leonardo Mannini - leonardo.mannini@studio.unibo.it

Lorenzo Sansone - lorenzo.sansone3@studio.unibo.it

January 2025

1 Introduction

The following report contains the strategies, the results and the models for solving the MCP problem. Our project lasted for six months. We have split the work in two parts: Lorenzo Sansone handled the CP and SAT part and Leonardo Mannini took care of SMT and MIP part. We faced the main difficulties with the most complex instances. In particular, memory consumption was a big issue for SAT and CP models.
In the next paragraph are presented the main commonalities between the models:

1.1 Variable

The input variables are the number of items n, the number of couriers m, the array of items' size s, the array of couriers' load l and the matrix of distances D.

2 CP model

The following section is dedicated to the Constraint Programming (CP) technique. Several encodings of the problem have been tested but the best models were chosen. The models were developed with MiniZinc and are tested with Gecode and Chuffed solver.

2.1 Decision variables

We used the following decision variables:

- $path_{ik}$ is a 2D array of size $[m, n+2]$ with domain $[1..n+1]$: it represents the path of the couriers and which points are visited. If $path_{ik} = j$ means that the courier i delivers the item j as the k-th point in the tour. If the courier i doesn't deliver any item at the k-th point the values assigned is $n+1$ (deposit). The *path* has $n+2$ columns because it has to represent the worst case, namely the case when a courier has to deliver all the n items.

- x_i is an array of size n with domain $[1..m]$: it represents the assignment of each item to a specific courier. If $x_i = j$ means that the item i is assigned to courier j.
- $load_i$ is an array of size m with domain $[0..max\_load]$: it represents the load of each courier. It depends on the number of the items assigned to a specific courier. If the $load_i = v$ means that the courier i has to load a total weight equal to v. The upper bound max_load is calculated as $\sum_{i=1}^{m} l_i$. $load_i$ is involved just in the models where symmetry breaking constraints are used.

2.2 Objective function

The objective variable is represented by y_i an array of size $[1..\ m]$ with domain $[0..max\_dist]$. It represents the total distance that each courier has to travel in order to deliver the assigned items. It is defined as follows:

$$y_i = \sum_{j=1}^{n+1} D[path_{i,j}, path_{i,j+1}]$$

The upper bound max_dist represents the maximum distance that a single courier can cover, namely when one courier has to deliver all the items.

$$max\_dist = D_{n+1,1} + \sum_{j=1}^{n} D_{i,i+1}$$

The lower bound is set to 0 because the courier can be remain in the deposit.
The objective function is to minimize the maximum distance of the couriers.

$$minimize \max_{i \in m} y_i$$

The lower bound of the objective function is equal to the maximum distance in the case a courier has to deliver one item $(deposit(n+1) \rightarrow item \rightarrow deposit(n+1))$

$$lower\_bound\_obj = \max \sum_{j=1}^{n} D_{n+1,i} + D_{i,n+1}$$

The upper bound of the objective function is equal to max_dist

2.3 Constraints

The constraints are listed as follows:

- **Couriers doesn't exceed their maximum load capacity**. Given each item i with size s_i, put into x_i such that the sum of the sizes of the items for each courier c does not exceed the capacity l_c. It is imposed via a global constraint:

$$bin\_packing\_capa(l, x, s)$$

- **Couriers deliver only the own items assigned**. The constraint checks if an item assigned in *path* belongs to the correct courier according to x. The first and last step in path are not checked because they are always equal to *n+1*.

$$x[path_{i,k}] == i \quad \forall i \in [1..m], \forall k \in [2..n+1], path_{i,k} \neq n+1$$

- **An item is carried by only one courier and is delivered one time**. Every value in *path* is assigned just one time (except for the deposit's value) and it is imposed via a global constraint.

$$alldifferent\_except(path, n+1)$$

- **Couriers start and end at the deposit**.

$$path_{i,1} = n+1 \quad path_{i,n+2} = n+1 \quad \forall i \in [1..m]$$

- **Checks if all the packages that are assigned to a courier are delivered by him**. This constraint also assures that all the items are assigned in *path*.

$$\exists k \; path[x_j, k] == j \quad \forall k \in [2..n+1] \quad \forall j \in [1..n]$$

- **Couriers can't return to the deposit if they have to deliver other items and force to starts immediatly if the courier has, at least one, items assigned**.

$$path_{i,j} == n+1 \rightarrow \nexists k \; path_{i,k} != n+1 \quad \forall i \in [1..m], \forall j \in [2..n+1], \forall k \in [j..n+1]$$

- **All couriers have to deliver at least one item**. A courier is forced to be assigned to, at least, one item in x. It is added just when the implied constraints are used:

$$\exists i \; x_j == i \quad \forall j \in [1..n], \forall i \in [1..m]$$

- **Compute the load of each courier**. This constraint is used when the symmetry breaking constraints are considered:

$$load_i = \sum_{j=1}^{n} (if \; x_j == i \; then \; s_j \; else \; 0) \quad \forall i \in [1..m]$$

The implied constraints are listed as follows:

- **Every courier has to deliver one item at the second time step**. In this way, it is explicit that every courier doesn't remain in the deposit. This constraint should help the solver during serch.

$$path_{i,2} \neq n+1 \quad \forall i \in [1..m]$$

The symmetry breaking constraints are listed as follows:

- **If two couriers have the same load capacity then they are interchangeable**.To break the symmetry we impose an order (for the package they pick up) betweem them.

$$l[courier1] == l[courier2] \rightarrow lex\_less(row(path, courier1), row(path, courier2))$$

$$\forall courier1 \in [1..m], \quad \forall courier2 \in [1..m], \quad courier1 < courier2$$

- **Two couriers path are interchangeable if the maximum item's weight of them is less than the minimum loading capacity**. In this way, the path are exchangeable so we impose an ordering between them.

$$l[courier1] > l[courier2] \rightarrow load[courier1] \geq load[courier2]$$

$$\forall courier1 \in [1..m], \quad \forall courier2 \in [1..m], \quad courier1 \neq courier2$$

2.4 Validation

Experimental design

The solver involved are *Chuffed* and *Gecode*. The models were tested with and without *implied constraint* and with and without *symmetry breaking constraints*. The execution time was set to 300s (the time to compute the parameter of the model is included). Several search strategies were explored but the most effective was *dom_w_deg* and *indomain_min* with *restart_luby* so this configuration was used for all the experiment except for the *bs* model where it was used the *first_fail* in order to highlight the performance differences. In the case of Gecode, *relax and reconstruct* (neighbourhood search strategy) was applied with 85% of probability of the variable being fixed to the previous solution for each restart. In order to take full advantage of Chuffed's performance, the *free_search* parameter was set to true to let the solver to choose its own search or user-defined search strategies. The experiments were run on an Asus Vivobook with an *Intel(R) Core(TM) i7-8565U CPU @1.80GHz*, *4 Cores*, *8* logical processor, *1.99 GHz* of Clock speed and *8GB* of RAM.

Experimental result

	Gecode					Chuffed				
Ins	bs	bs_heu	bs_heu impl	bs_heu sym	bs_heu sym impl	bs	bs_heu	bs_heu impl	bs_heu sym	bs_heu sym impl
1	**14**	15	14	14	14	**14**	**14**	**14**	**14**	**14**
2	**226**	**226**	**226**	**226**	**226**	**226**	**226**	**226**	**226**	**226**
3	**12**	12	12	12	12	**12**	**12**	**12**	**12**	**12**
4	**220**	**220**	**220**	**220**	**220**	**220**	**220**	**220**	**220**	**220**
5	**206**	206	206	206	206	**206**	**206**	**206**	**206**	**206**
6	**322**	**322**	**322**	**322**	**322**	**322**	**322**	**322**	**322**	**322**
7	**167**	**167**	**167**	**167**	**167**	**167**	**167**	**167**	**167**	**167**

8	**186**	**186**	**186**	**186**	**186**	**186**	**186**	**186**	**186**	**186**
9	**436**	**436**	**436**	**436**	**436**	**436**	**436**	**436**	**436**	**436**
10	**244**	**244**	**244**	-	-	**244**	**244**	**244**	**244**	**244**
11	-	952	613	-	-	-	-	-	-	-
12	1741	**346**	**346**	-	-	1589	1587	-	-	-
13	1932	484	492	490	468	1932	1932	1932	1932	1932
14	-	-	-	-	-	-	-	-	-	-
15	-	-	-	-	-	-	-	-	-	-
16	1045	**286**	**286**	-	-	1030	**286**	-	-	-
17	-	-	-	-	-	-	-	-	-	-
18	-	-	-	-	-	-	-	-	-	-
19	1495	**334**	**334**	-	-	1332	897	-	-	-
20	-	-	-	-	-	-	-	-	-	-
21	-	942	868	-	-	-	-	-	-	-

The table shows the results of the models solved with Gecode and Chuffed solver. Moreover, it is shown symmetry breaking constraint models(*sym*), implied constraint models (*impl*) and the combination of both. In the Gecode section, the model *bs_heu*, *bs_heu_impl*, *bs_heu_sym*, *bs_heu_sym_impl* were run with *relax and reconstruct*

Other experimental design: Several configuration and strategies were tested but the results didn't improve. The size of the matrix *path* was reduced because if all the couriers deliver at least one item it can't happen that all the columns are used. Moreover, the first and last column were removed because they have to be equal to *n+1* and this constraint can be managed during the computation of the objective variable. The upper and lower bounds of the objective function and of variable function(y) were explored deeply: the minimum distance (originally zero) was increased especially for the model where all the couriers have to deliver at least one item and the maximum distance was reduced using a small minizinc model to compute a maximum efficient path. All these experiments should reduce the search space but the final result was worse. The upper bound of the array *load* (*max_load*) was reduced to max(l) but the outcomes weren't changed, especially for the instances with no results.

3 SAT model

In this section, the MCP problem is solved with SAT theory which uses only boolean variable and the constraints are encoded with Propositional Logic.

3.1 Decision variables

The main decision variables are listed as follows:

- $path_{ijk}$ is a 3D array of size [m, *n+1*, *n+2*]. It represents the path of the couriers and which points are visited. $path_{ijk} = 1$ if the courier i delivers the package j at

the k-th step. $path_{i,n+1,k} = 1$ means that the courier i is in the deposit at the time step k

$$path_{ijk} \in [0, 1] \quad \forall m \in [1..m] \quad \forall j \in [1..n+1] \quad \forall k \in [1..n+2]$$

- $courier\_weights_{ij}$ is 2D array of size $[m, n]$. $courier\_weights_{ij} = 1$ if the courier i takes the package j, otherwise 0.
- $courier\_loads_i$ is an array of size m. It stores the binary encoding of load carried by each courier. The maximum value that can be stored is $max\_load = \sum_{i=1}^{n} s_i$ because it is the case where one courier has to deliver all the items.
- The input variables l, s and D are binary encoded accordingly (l_b, s_b, D_b).

3.2 Objective function

The objective variable is represented by $courier\_dists_i$, is an array of size m. It stores the binary encoding of the distances travelled by each courier. The maximum value that can be stored is equal to the sum of the n longest distances of D. The objective function is to minimize the maximum distance of the couriers

$$minimize \max_{i \in m} courier\_dists_i$$

The *upper_bound_obj* of the objective function is equal to the one of CP theory (Section 2.2). The *lower_bound_obj* is equal to zero for linear search (it is explained in Section 3.4) and is equal to the *lower_bound_obj* of CP theory for binary search (Section 3.4)

3.3 Constraints

The main constraints are listed as follows:

- **Every courier delivers exactly one package at each step**. It also imposes that every courier visits exactly one location at each step (at least the deposit).

$$\bigwedge_{i=1}^{m} \bigwedge_{k=1}^{n+2} exactly\_one(path_{ijk} \mid \forall j \in [1..n+1])$$

- **Each package is delivered only once**.

$$\bigwedge_{j=1}^{n} exactly\_one(path_{ijk} \mid \forall i \in [1..m],\ \forall k \in [1..n+2])$$

$$\bigwedge_{j=1}^{n} exactly\_one(courier\_weights_{ij} \mid \forall i \in [1..m])$$

- **Couriers start and end at the deposit**.

$$path_{i,n+1,1} = True \quad path_{i,n+1,n+2} = True \quad \forall i \in [1..m$$

- Every courier has a **maximum load capacity to respect** (*geq* means greater or equal).

$$geq(l\_b_i, courier\_loads_i) \quad \forall i \in [1..m]$$

- **If a courier doesn't take the a pack at position j, also at position j+1 doesn't take any pack**. It also means that the courier can't come back to the deposit if he has to deliver other packagages

$$\bigwedge_{i=1}^{m} \bigwedge_{k=2}^{n} path_{i,n+1,k} \rightarrow path_{i,n+1,k+1}$$

The implied constraints are listed as follows:

- All the couriers have to **start immediatly and deliver one item**

$$\bigwedge_{i=1}^{m} at\_least\_one\_(path_{i,j,2} \mid \forall j \in [1..n])$$

In the symmtry breaking constraints is used an ordered array of load called *l_sorted*. It is calculated by sorting in descending order *l*.

- if **the load of two couriers are equal** then define the lexicographical order:

  $l\_sorted_i == l\_sorted_{i+1} \rightarrow lex\_leq(courier\_weights_{i1}, courier\_weights_{i2})) \quad \forall i \in [1..m]$

  The *i1* and *i2* are the index of the courier that correspond with the index *i* and *i+1* of the array *l_sorted*

- if **the load of the first courier is greater than the load of the second courier** then impose an order:

  $l\_sorted_i \geq l\_sorted_{i+1} \rightarrow geq(courier\_loads_{i1}, courier\_loads_{i2}) \quad \forall i \in [1..m]$

3.4 Validation

Experimental design
The models and experiment were built using Z3 python library. Two different search strategies were implemented:

- linear search: The search starts with a fixed value for the upper bound of the objective function and at each iteration it is stricted until the problem becomes unsatisfiable. At this point the last SAT solution is considered.

- binary search: At each iteration the upper and lower bound of the objective function are dinamically updated dividing the search interval in half. This approach significantly reduceds the number of comparisons compared to linear search.

The model were run with and without *symmetry breaking constraints* and using *linear* and *binary* seaerch. The experiments were run on an Asus Vivobook with an *Intel(R) Core(TM) i7-8565U CPU @1.80GHz*, *4 Cores*, *8* logical processor, *1.99 GHz* of Clock speed and *8GB* of RAM.

Experimental result

Inst	LNS	LNS_SYM	BNS	BNS_SYM
1	**14**	**14**	**14**	**14**
2	**226**	**226**	**226**	**226**
3	**12**	**12**	**12**	**12**
4	220	**220**	**220**	**220**
5	**206**	**206**	**206**	**206**
6	**322**	**322**	**322**	**322**
7	193	170	169	**167**
8	**186**	**186**	**186**	**186**
9	436	436	**436**	**436**
10	244	244	**244**	**244**

The table shows the results of the SAT model with and without symmetry breaking constraints and with linear and binary search. The instances from 11 to 21 haven't any results.

4 SMT model

The SMT (Satisfiability Modulo Theories) approach was implemented using the Z3 solver through its Python API. Four different variants of the model were developed and tested:

- Base model (Z3_SMT_Base_Solver)
- Model with symmetry breaking constraints (Z3_SMT_SymBrk_Solver)
- Model with both symmetry breaking and implied constraints (Z3_SMT_SymBrk_ImplConstr_Solver)
- Binary search variant with symmetry breaking (Z3_SMT_SymBrk_BinarySearch)

4.1 Decision variables

The main decision variables are:

- $path_{ik}$ is a 2D array of size [m, $n+2$] with domain [1..$n+1$]: represents the sequence of locations visited by each courier. If $path_{ik} = j$ means courier i visits location j at step k. Value $n+1$ represents the depot.

- $courier\_assignments_{ij}$ is a 2D boolean array of size $[m, n]$: represents the assignment of items to couriers

$$courier\_assignments_{ij} \in \{0, 1\} \quad \forall i \in [1..m], \forall j \in [1..n]$$

  where $courier\_assignments_{ij} = 1$ means courier i is assigned to deliver item j

- $courier\_loads_i$ is an array of size m with domain $[0..max\_load]$: represents the total load carried by each courier

$$courier\_loads_i \in [0..\sum_{j=1}^{n} s_j] \quad \forall i \in [1..m]$$

- $courier\_distances_i$ is an array of size m: stores the total distance traveled by each courier

$$courier\_distances_i \in [0..max\_dist] \quad \forall i \in [1..m]$$

  where max_dist is calculated as in the CP model

4.2 Objective function

The objective is to minimize the maximum distance traveled by any courier:

$$minimize \max_{i \in [1..m]} courier\_distances_i$$

The distance for each courier is calculated as:

$$courier\_distances_i = \sum_{k=1}^{n+1} D[path_{i,k}, path_{i,k+1}] \quad \forall i \in [1..m]$$

Two different search strategies were implemented for finding the optimal solution:

- Linear search: Iteratively reducing the upper bound until finding the minimum feasible solution
- Binary search: Using a divide-and-conquer approach to find the optimal solution by maintaining both upper and lower bounds

4.3 Constraints

The core constraints implemented in the SMT models include:

- **Load capacity**: Each courier's total load must not exceed their capacity

$$courier\_loads_i = \sum_{j=1}^{n} (courier\_assignments_{ij} \cdot s_j) \leq l_i \quad \forall i \in [1..m]$$

- **Path validity**: Each location must be visited exactly once, and all paths must start and end at the depot

$$path_{i,1} = n + 1 \land path_{i,n+2} = n + 1 \quad \forall i \in [1..m]$$

$$\bigwedge_{j=1}^{n} \sum_{i=1}^{m} \sum_{k=2}^{n+1} (path_{i,k} = j) = 1$$

- **Assignment consistency**: Items must be delivered by the courier they are assigned to

$$path_{i,k} = j \rightarrow courier\_assignments_{ij} = 1 \quad \forall i \in [1..m], \forall k \in [2..n+1], \forall j \in [1..n]$$

- **Assignment completeness**: Each item must be assigned to exactly one courier

$$\sum_{i=1}^{m} courier\_assignments_{ij} = 1 \quad \forall j \in [1..n]$$

The symmetry breaking variant adds constraints to:

- **Load ordering**: For couriers with equal capacity, enforce lexicographic ordering

$$l_i = l_j \rightarrow lex\_less(courier\_assignments_i, courier\_assignments_j) \quad \forall i, j \in [1..m], i < j$$

- **Load balance**: For couriers with different capacities, maintain load relationship

$$l_i > l_j \rightarrow courier\_loads_i \geq courier\_loads_j \quad \forall i, j \in [1..m], i \neq j$$

The implied constraints variant further adds:

- **Immediate start**: Couriers must start delivering immediately if they have assignments

$$(\exists j : courier\_assignments_{ij} = 1) \rightarrow path_{i,2} \neq n + 1 \quad \forall i \in [1..m]$$

- **No return until complete**: Couriers cannot return to depot until all their assigned items are delivered

$$path_{i,k} = n + 1 \rightarrow \bigwedge_{j=k+1}^{n+1} path_{i,j} = n + 1 \quad \forall i \in [1..m], \forall k \in [2..n + 1]$$

4.4 Validation

Experimental design
The experiments were conducted using the Z3 SMT solver with a timeout of 5 minutes per instance. All four model variants were tested on the benchmark instances. The implementation uses Z3's Python API for model construction and solving.

The experiments were run with the following specifications:

- MacBook Air M1, 2020
- RAM: 8GB
- Processor: Apple M1
- Clock speed: Up to 3.2 GHz (for performance cores; efficiency cores run at lower frequencies)
- Cores: 8 (4 high-performance cores + 4 high-efficiency cores)
- Logical processors: 8

Experimental Results

Ins	Base	SymBrk	SymBrk ImplConstr	SymBrk BinSearch
1	**14**	**14**	**14**	**14**
2	**226**	**226**	**226**	**226**
3	**12**	**12**	**12**	**12**
4	**220**	**220**	**220**	**220**
5	**206**	**206**	**206**	-
6	**322**	**322**	**322**	**322**
7	270	258	186	255
8	**186**	**186**	**186**	**186**
9	**436**	**436**	**436**	**436**
10	-	**244**	323	**244**
11	-	-	-	-
12	-	-	-	-
13	-	-	-	-
14	-	-	-	-
15	-	-	-	-
16	-	-	-	-
17	-	-	-	-
18	-	-	-	-
19	-	-	-	-
20	-	-	-	-
21	-	-	-	-

5 MIP model

The Mixed Integer Programming (MIP) approach was implemented using Gurobi through its Python API. The model was designed to efficiently handle the routing and assignment aspects of the Multiple Couriers Planning problem.

5.1 Decision variables

The main decision variables in the MIP model are:

- x_{ij} is a binary 2D array of size [m, n]: represents the assignment of items to couriers

  $$x_{ij} \in \{0, 1\} \quad \forall i \in [1..m], \forall j \in [1..n]$$

  where $x_{ij} = 1$ means courier i delivers item j

- y_{ijk} is a binary 3D array of size [m, *locations*, *locations*]: represents the travel paths of couriers

  $$y_{ijk} \in \{0, 1\} \quad \forall i \in [1..m], \forall j, k \in [1..locations]$$

  where $y_{ijk} = 1$ means courier i travels from location j to location k

- $Distance_i$ is an integer array of size m: represents the total distance traveled by each courier

- *max_distance* is an integer variable: represents the maximum distance among all couriers

- u_{ij} is a continuous 2D array of size [m, n]: auxiliary variables for subtour elimination

5.2 Objective function

The objective is to minimize the maximum distance traveled by any courier:

$$minimize\ max\_distance$$

where *max_distance* is constrained by:

$$max\_distance \geq Distance_i \quad \forall i \in [1..m]$$

The distance for each courier is calculated as:

$$Distance_i = \sum_{j1=1}^{locations} \sum_{j2=1}^{locations} y_{i,j1,j2} \cdot D_{j1,j2} \quad \forall i \in [1..m]$$

5.3 Constraints

The core constraints in the MIP model include:

- **Load capacity**: Each courier's total load must not exceed their capacity

$$\sum_{j=1}^{n} x_{ij} \cdot s_j \leq l_i \quad \forall i \in [1..m]$$

- **Single courier assignment**: Each item must be delivered by exactly one courier

$$\sum_{i=1}^{m} x_{ij} = 1 \quad \forall j \in [1..n]$$

- **Depot constraints**: Each courier must start and end at the depot (location n)

$$\sum_{j=1}^{locations} y_{i,n,j} = 1 \quad \forall i \in [1..m]$$

$$\sum_{j=1}^{locations} y_{i,j,n} = 1 \quad \forall i \in [1..m]$$

- **Self-loop prevention**: Couriers cannot travel from a location to itself

$$y_{i,j,j} = 0 \quad \forall i \in [1..m], \forall j \in [1..locations]$$

- **Flow conservation**: For each location visited, a courier must also leave it

$$\sum_{k=1}^{locations} y_{i,k,j} = \sum_{k=1}^{locations} y_{i,j,k} \quad \forall i \in [1..m], \forall j \in [1..locations]$$

- **Visit-delivery consistency**: If a courier delivers an item, they must visit and leave its location

$$\sum_{k=1}^{locations} y_{i,j,k} = x_{ij} \quad \forall i \in [1..m], \forall j \in [1..n]$$

$$\sum_{k=1}^{locations} y_{i,k,j} = x_{ij} \quad \forall i \in [1..m], \forall j \in [1..n]$$

- **Subtour elimination**: Using Miller-Tucker-Zemlin formulation

$$u_{ij} - u_{ik} + (n-1)y_{i,j,k} \leq n-2 \quad \forall i \in [1..m], \forall j,k \in [1..n], j \neq k$$

with bounds:

$$1 \leq u_{ij} \leq n-1 \quad \forall i \in [1..m], \forall j \in [1..n-1]$$

5.4 Validation

Experimental design
The experiments were conducted using Gurobi 11.0.3 with a timeout of 5 minutes per instance.

The experiments were run on:

- MacBook Air M1, 2020
- RAM: 8GB
- Processor: Apple M1
- Clock speed: Up to 3.2 GHz
- Cores: 8
- Logical processors: 8

Experimental Results
The MIP model was able to solve smaller instances (1-10) to optimality within the time limit, while larger instances (11-21) proved more challenging, as in other implementations.

Ins	gurobipy
1	**14**
2	**226**
3	**12**
4	**220**
5	**206**
6	**322**
7	**172**
8	**188**
9	**436**
10	**244**
11	-
12	503
13	444
14	2214
15	1589
16	**286**
17	-
18	1893
19	**334**
20	-
21	1168

6 Conclusion

The Multiple Couriers Planning (MCP) problem was solved using four different solving approaches: Constraint Programming (CP), Boolean Satisfiability (SAT), Satisfiability Modulo Theories (SMT), and Mixed Integer Programming (MIP).

For small instances (1-10), all approaches performed remarkably well, consistently finding optimal solutions. The MIP implementation using Gurobi proved to be the most robust, solving these instances efficiently and extending its effectiveness to several larger instances. Notably, it found optimal solutions for instances 16 and 19, which proved challenging for other approaches.

The CP models, implemented with both Gecode and Chuffed solvers, showed strong performance with various enhancements. The addition of symmetry breaking constraints and implied constraints improved solution quality, particularly evident in instances 11-13. However, the CP approach struggled with memory consumption in larger instances, highlighting a common limitation of this method.

The SAT approach, while effective for small instances (1-10), demonstrated limitations in scalability. The binary search variant with symmetry breaking (BNS_SYM) showed the best performance among SAT implementations, particularly in finding optimal solutions for instances 7-10. However, its effectiveness diminished significantly for larger instances.

The SMT approach, implemented with Z3, showed promising results for small and medium-sized instances. The addition of symmetry breaking constraints and implied constraints improved solution quality, but the approach still struggled with larger instances (11-21), often failing to find solutions.

The MIP-gurobipy approach emerged as the most effective, managing to find solutions (though not always optimal) for several larger instances where other approaches failed. This suggests that the mathematical programming approach, combined with Gurobi's sophisticated solving techniques, is particularly adapt for handling the routing and assignment aspects of the MCP problem.

An observation across all approaches is the impact of problem size on solution quality and solving time. The "turning point" appears to be around instance 11, where the problem's complexity increases substantially, challenging all solving methods.

In conclusion, while each approach has its merits, the MIP implementation stands out as the most practical choice for real-world applications of the MCP problem, offering a good balance between solution quality and computational efficiency. For smaller instances, any of the approaches would be suitable, with CP and SMT offering particularly good performance when enhanced with symmetry breaking and implied constraints.