

Comparative Code and Performance Analysis of Solvers for the Traveling Salesman Problem

Adelino Daniel da Rocha Vilaça

MAAI, EST-IPCA

Barcelos, Portugal

a16939@alunos.ipca.pt ORCID: 0009-0006-7501-9920

Abstract—This report describes a four-phase project that compares Picat implementations of the Traveling Salesman Problem (TSP): a SAT encoding, a Constraint Programming (CP) model, and a CP-based extension with time windows (TSPTW). The work covers theoretical background, comparative analysis, a model extension (time windows), and performance benchmarking. The report follows the project phases and summarises the modelling decisions and the executed benchmarks and results; we also briefly contrast SMT and MIP variants in Phase 4 (executing SMT where feasible).

Index Terms—Traveling Salesman Problem, Picat, Constraint Programming, SAT, Time Windows, Benchmarking

I. INTRODUCTION

This project explores different modelling paradigms for the Traveling Salesman Problem (TSP) implemented in Picat. The objectives are: (1) to analyse two reference models (a SAT encoding and a CP encoding), (2) to extend one model with time windows (Option A), (3) to benchmark performance on small/medium instances, and (4) to synthesise findings about readability, modelling effort, and solver suitability. The structure follows the four project phases.

Section II summarises Phase 1 (theory and foundational analysis). Section III presents the SAT vs CP comparative analysis. Section IV describes the model extension with time windows (Option A). Section V reports the executed benchmarks and results. The final section concludes and lists next steps to finalise the report.

II. PHASE 1 — RESEARCH AND FOUNDATIONAL CODE ANALYSIS

Phase 1 established the theoretical background of the TSP and dissected two reference Picat implementations: `tsp_1.pi` (SAT encoding) and `tsp_2.pi` (CP).

Key theoretical points:

- TSP asks for a minimum-cost Hamiltonian cycle visiting each city exactly once.
- The problem is NP-hard; typical modelling approaches trade-off verbosity and solver control.

SAT model highlights (from `tsp_1.pi`):

- Edge-based boolean matrix `NextArr[I, J]` with $N \times N$ binary variables.
- Degree constraints: exactly one outgoing and one incoming edge per city.

- An explicit ordering array `Order` with implications of the form `NextArr[I, J] #=> Order[J] #= Order[I] + 1` used for sub-tour elimination.
- Cost linking by implications (if an edge is chosen then the corresponding `CostArr` entry equals the matrix value).

Key SAT code snippets:

```
1 foreach (I in 1..N)
2     sum([NextArr[I, J] : J in 1..N]) #= 1
3 end,
```

Listing 1: Successor constraint: one outgoing edge per city

Explanation: This loop enforces the degree-1 successor condition for every city I. The sum over row I of the boolean edge matrix must equal 1, ensuring exactly one outgoing edge is chosen from each city in any feasible tour.

```
1 foreach (J in 1..N)
2     sum([NextArr[I, J] : I in 1..N]) #= 1
3 end,
```

Listing 2: Predecessor constraint: one incoming edge per city

Explanation: Symmetrically, this loop sums each column J and forces it to 1. Together with the successor constraint, this guarantees each city has exactly one predecessor and one successor, a core TSP feasibility requirement.

```
Order = new_array(N), Order :: 1..N, Order[1]
      = 1,
foreach (I in 1..N)
    NextArr[I, 1] #=> Order[I] #= N,
    foreach (J in 2..N)
        NextArr[I, J] #=> Order[J] #= Order[I
            ]+1
    end
end,
```

Listing 3: Ordering implications for sub-tour elimination

Explanation: The `Order` array assigns visit positions. If edge I-J is chosen, then J must follow I immediately in the order. This implication chain prevents disjoint cycles (sub-tours) by constructing a single sequence anchored at city 1.

```
1 CostArr = new_array(N),
2 foreach (I in 1..N, J in 1..N)
3     NextArr[I, J] #=> CostArr[I] #= M[I, J]
4 end,
```

Listing 4: Cost assignment linked to chosen edge

Explanation: When the model selects edge I-J (`NextArr[I,J]==1`), the per-node cost variable `CostArr[I]` is equated to the corresponding matrix entry `M[I,J]`. Summing `CostArr` yields the tour length used in the objective.

CP model highlights (from `tsp_2.pi`):

- Successor-based array `NextArr[I] = J` (only N variables).
- Global constraint `circuit(NextArr)` enforces a single Hamiltonian cycle and avoids sub-tours implicitly.
- Cost assigned with element(`NextArr[I]`, `M[I]`, `CostArr[I]`) and objective `TotalCost # = sum(CostArr)`.

Key CP code snippets:

```

1 NextArr = new_array(N),      % visit NextArr[I]
   after I
2 NextArr :: 1..N,
3 circuit(NextArr),

```

Listing 5: Successor representation and global circuit

Explanation: The CP model uses a single successor array, where `NextArr[I]=J` encodes edge I-J. The global constraint `circuit(NextArr)` ensures a single Hamiltonian cycle, implicitly handling all-different, in/out-degree, and subtour elimination.

```

1 CostArr = new_array(N),
2 foreach (I in 1..N)
3   element(NextArr[I], M[I], CostArr[I])
4 end,

```

Listing 6: Element constraint for per-arc cost lookup

Explanation: The element constraint binds the chosen successor of city I to its corresponding matrix entry `M[I,*]`, assigning `CostArr[I]`. This declarative lookup avoids manual implications and keeps the model concise and readable.

```

1 TotalCost #= sum(CostArr),
2 solve([$min(TotalCost), report println(cost=
   TotalCost)], NextArr),

```

Listing 7: Objective: minimize total tour cost

Explanation: The objective aggregates per-arc costs into `TotalCost` and minimizes it. The solver call uses a reporting option to print intermediate/best costs, aiding verification and benchmarking while keeping the model compact.

Other paradigms at a glance (SMT and MIP)

- **SMT (Z3 via `import smt`):** Can reuse the SAT-style edge-matrix encoding with integer sums and reified implications. Good for mixing arithmetic with boolean structure without hand-crafted CNF.
- **MIP (Gurobi via `import mip`):** Linearizes the edge-matrix model: boolean arc variables, row/column sums, MTZ (or flow) subtour cuts, and linear cost. Requires an external solver license but yields strong performance on small/medium instances.

A. Problem statement

Let the cities be $V = \{1, \dots, N\}$ and let $M \in \mathbb{R}^{N \times N}$ be the cost (distance/time) matrix. A tour is a permutation π over V with wrap-around $\pi(N+1) = \pi(1)$. The objective is to minimize total cost:

$$\min \sum_{i=1}^N M_{\pi(i), \pi(i+1)}. \quad (1)$$

Feasibility requires visiting each city exactly once and returning to the start (Hamiltonian cycle).

B. Data and scope

For Phase 1 inspection we use the small matrix returned by `data1()` (as in `tsp_2.pi`). This symmetric instance is adequate to illustrate model structure and objective definition.

C. Assumptions

Unless otherwise noted: diagonal entries are zero; costs are non-negative; matrices are symmetric (if not, treat them as directed).

The Phase 1 analysis concluded that the CP model is more concise and easier to extend, while the SAT model provides fine-grained control at the cost of verbosity and maintenance effort.

III. PHASE 2 — COMPARATIVE ANALYSIS (SAT vs CP)

This section contrasts two Picat baselines for the TSP: an edge-matrix SAT encoding (`tsp_1.pi`) and a successor-array CP encoding (`tsp_2.pi`).

A. Model contracts (inputs/vars/constraints/objective)

Common input: cost matrix $M \in \mathbb{R}^{N \times N}$ (non-negative, $M_{ii} = 0$). Objective: minimize $\sum_i M_{\pi(i), \pi(i+1)}$.

SAT (edge matrix):

- Vars: $X_{ij} \in \{0, 1\}$ for all $i, j \in \{1..N\}$ indicate using arc $(i \rightarrow j)$.
- Degree constraints: $\sum_j X_{ij} = 1$ and $\sum_i X_{ij} = 1$ (exactly one out- and in-arc per city).
- Sub-tour elimination: ordering array `Order` with implications (e.g., `X[I, J] #=> Order[J] #= Order[I]+1`).
- Cost: link chosen edges to costs and minimize the total.

CP (successor array):

- Vars: `NextArr[I] ∈ {1..N}` gives the successor of city I (only N vars).
- Tour constraint: `circuit(NextArr)` enforces a single Hamiltonian cycle.
- Cost: `element(NextArr[I], M[I], CostArr[I])`; minimize `sum(CostArr)`.

B. Encoding size and structure

- **Vars:** SAT uses N^2 booleans; CP uses N integers.
- **Constraints:** SAT has $2N$ degree equalities plus many implications for sub-tours; CP uses one global `circuit/1` and N `element/3` constraints.
- **Propagation:** `circuit/1` provides strong global propagation, often reducing branching early.

C. Mini code excerpts (side-by-side)

SAT (row/col sums + ordering)

```

1 foreach (I in 1..N) sum([NextArr[I,J] : J in
2   1..N]) #= 1 end,
3 foreach (J in 1..N) sum([NextArr[I,J] : I in
4   1..N]) #= 1 end,
5 NextArr[I,J] #=> Order[J] #= Order[I]+1.

```

These constraints make each city have exactly one successor and predecessor; the implication ties chosen edges to consecutive positions, ruling out subtours.

CP (global circuit + element)

```

1 NextArr = new_array(N), NextArr :: 1..N,
2 circuit(NextArr),
3 element(NextArr[I], M[I], CostArr[I]).

```

The global circuit enforces a single Hamiltonian cycle; element maps the chosen successor to its arc cost, keeping the model compact and declarative.

D. Backends and complexity at a glance

Backends: SAT uses Kissat via `import sat`; CP uses Picat's native CP via `import cp`.

Asymptotics: SAT has $\mathcal{O}(N^2)$ variables and many implications; CP uses $\mathcal{O}(N)$ successors, one circuit, and $\mathcal{O}(N)$ element constraints.

E. Readability, extensibility, and maintenance

- **Readability:** CP is concise and declarative; SAT exposes more low-level detail.
- **Extensibility:** Adding schedule-like features (e.g., time windows) integrates naturally with CP (see Phase IV); in SAT it requires additional layers of implications and auxiliary variables.
- **Maintenance:** Fewer moving parts in CP reduce the risk of subtle modelling bugs.

Strengths and risks (short):

- SAT: fine-grained control over edges; risk is verbosity and subtour logic maintenance.
- CP: highly declarative and quick to extend; less explicit micromanagement per edge.

F. Micro example (3 cities, conceptual)

For $N = 3$, SAT creates 9 booleans X_{ij} with row/column sums = 1 and ordering implications; CP uses 3 integers $\text{NextArr}[1..3]$ with `circuit/1`. A concrete CP tour is, e.g., $\text{NextArr}=[2, 3, 1]$ ($1 \rightarrow 2 \rightarrow 3 \rightarrow 1$). The SAT row constraint for city 1 is $X[1,1] + X[1,2] + X[1,3] = 1$. Both encodings yield the same tour; CP does so with fewer variables.

G. When to choose which

- Choose CP for rapid prototyping, readability, and adding scheduling-style constraints (e.g., time windows).
- Choose SAT when you need explicit edge-level logic or to integrate complex boolean structure.

TABLE I: SAT vs CP at a glance

Aspect	SAT (<code>tsp_1.pi</code>)	CP (<code>tsp_2.pi</code>)
Variables	$N \times N$ booleans (edges)	N integers (successors)
Core constraint	Row/column sums = 1	<code>circuit(NextArr)</code>
Sub-tours	Ordering + implications	Implicit via <code>circuit</code>
Cost	Implication linking	<code>element/3</code>
Typical LOC	Higher	Lower

H. Summary

CP achieves comparable or better performance on small/medium instances with less code and simpler extensions; SAT can be competitive with careful tuning but carries higher encoding overhead.

Extending the comparison: SMT and MIP (brief)

- **SMT:** Similar variable/constraint shape to SAT but delegated to an SMT solver (Z3). Useful when arithmetic constraints and reification are natural; avoids CNF translation overhead.
- **MIP:** Uses linear constraints with binary variables. Classic TSP uses MTZ or flow-based subtour elimination; scales well on small/medium sizes and integrates naturally with linear side constraints.

IV. PHASE 3 — MODEL EXTENSION (OPTION A: TIME WINDOWS)

For Phase 3 we extend the CP model to handle time windows (the provided `tsptw.pi`). Design choices (concise):

A. Modeling additions

- Add arrival time variables `Arrive[I]` and possibly waiting times.
- For chosen successor $j = \text{NextArr}[i]$, propagate times via: $\text{Arrive}[j] \geq \text{Arrive}[i] + \text{travel_time}(i, j) + \text{service_time}(i)$.
- Enforce window bounds: $\text{Earliest}[j] \leq \text{Arrive}[j] \leq \text{Latest}[j]$.

B. Implementation notes

- Utilities: `generate_random_distances/2`, `generate_random_times/2` for synthetic instances.
- Backend: CP backend (`import cp`) is used: `circuit`, `element`, and integer relations are native. A MIP alternative would require a linearized model (binary arcs, MTZ/flow, big-M time propagation).
- Objective variants: `min_distance` (sum Costs) or `min_time` (TravelTime).

C. Assumptions

When data is unspecified:

- Service times default to 0; waiting is non-negative.
- Travel times are symmetric if the matrix is symmetric; otherwise, use directed times.

D. Model structure and code examples

The TSPTW model keeps the CP successor representation (array `Cities`) and the explicit visit order (array `Path`). Below, three compact excerpts illustrate the core tour constraints, the order extraction, and the time-related constraints.

Model contract (inputs, variables, constraints, objective):

Inputs: distance matrix `DistanceMatrix[I, J]` and time windows `TimeWindows[I]=[LB, UB]`.

Variables:

- `Cities[1..N]`: successor array forming a Hamiltonian cycle (via `circuit`).
- `Path[1..N]`: visiting order extracted from `Cities` (with `Path[1]=1`).
- `ArrTimes[1..N]`: arrival before waiting; `WaitTimes[1..N] \geq 0`: waiting slack; `CumTimes=Arr+Wait`.
- `Costs[1..N]` and `TravelDistance=sum(Costs)`; `TravelTime=CumTimes[N]`.

Constraints:

- `Tour:circuit(Cities); circuit_path(Cities, Path); Path[1]=1.`
- *Time propagation:* $\text{Arr}[I] = \text{Cum}[I - 1] + \text{Dist}(\text{Path}[I - 1], \text{Path}[I])$ for $I \geq 2$; depot has $\text{Arr}[1] = 0$.
- *Windows:* $\text{Wait}[I] \geq \text{LB}(\text{Path}[I]) - \text{Arr}[I]$, $\text{Wait}[I] \geq 0$; $\text{Cum}[I] = \text{Arr}[I] + \text{Wait}[I]$ and $\text{Cum}[I] \leq \text{UB}(\text{Path}[I])$.

Objectives: minimize `TravelDistance` (classic) or `TravelTime` (makespan).

Tour and cost linking: .

```

1 % Decision variables
2 Len = DistanceMatrix.length,
3 Cities = new_list(Len),
4 Cities :: 1..Len,
5
6 Costs = new_list(Len),
7 Costs :: 0..max(DistanceMatrix),
8
9 % Hamiltonian cycle and arc costs
10 circuit(Cities),
11 foreach({Row,City,Cost} in
12 zip(DistanceMatrix,Cities, Costs))
13   element(City, Row, Cost)
14 end,
15
16 % Total distance objective (variant 1)
17 TravelDistance #= sum(Costs).

```

Listing 8: Tour structure and cost linking

Explanation:

- **Len:** number of cities (assumes 1-based indexing).
- **Cities:** successor array where `Cities[I]` is the next city after `I`.
- **circuit(Cities):** global constraint enforcing a single Hamiltonian cycle (no subtours, each city has one successor and one predecessor).
- **element/3:** links each chosen successor to its arc cost: for row `Row` of the distance matrix and successor `City`, `Cost` becomes `Row[City]`.

- **TravelDistance:** linear objective as the sum of per-arc costs, minimized in the solver call (see objective variants).

Order extraction (Path): .

```

1 Path = new_list(Len), Path :: 1..Len,
2 all_distinct(Path),
3 circuit_path(Cities, Path).

```

Listing 9: Extracting a linear order from the circuit

Explanation:

- **Path:** explicit visiting order (a permutation of $1..N$) extracted from the cycle in `Cities`.
- **all_distinct(Path):** ensures a permutation domain; together with `circuit_path/2` this yields a unique sequence without repetition.
- **Why Path?** Time-window reasoning is naturally expressed over a linear sequence (predecessor then successor). The CP cycle is rotation-invariant; `Path` provides a concrete order for time propagation.

```

1 % Optionally fix start at depot (anchor to
2 % city 1)
3 Path[1] #= 1,
4
5 % Arrival and cumulative times
6 ArrTimes = new_list(Len), ArrTimes :: 0...
7 MaxTimeWindows,
8
9 % First city (depot at t=0)
10 ArrTimes[1] #= 0,
11 element(Path[1], WaitTimes, WT1),
12 matrix_element(TimeWindows, Path[1], 1, LB1),
13 matrix_element(TimeWindows, Path[1], 2, UB1),
14 WT1 #>= LB1 - ArrTimes[1], WT1 #>= 0,
15 CumTimes[1] #= ArrTimes[1] + WT1,
16 CumTimes[1] #=< UB1,
17
18 % Subsequent cities
19 foreach(I in 2..Len)
20   matrix_element(DistanceMatrix, Path[I-1],
21   Path[I], T),
22   ArrTimes[I] #= CumTimes[I-1] + T,
23   element(Path[I], WaitTimes, WT),
24   matrix_element(TimeWindows, Path[I], 1, LB),
25   matrix_element(TimeWindows, Path[I], 2, UB),
26   WT #>= LB - ArrTimes[I], WT #>= 0,
27   CumTimes[I] #= ArrTimes[I] + WT,
28   CumTimes[I] #=< UB
29 end.

```

Listing 10: Depot anchoring (optional), wait semantics, and time-window enforcement

Explanation:

- **Depot anchoring:** start at city 1 at $t = 0$; wait is the non-negative slack to meet the lower bound.
- **Recurrence:** arrival at I equals previous cumulated time plus travel; cumulated time adds the wait.

- **Windows:** enforce lower/upper bounds via `matrix_element(TimeWindows, city, col, val)`.
- **Infeasibility:** if any window cannot be satisfied under any route, the model becomes unsatisfiable; this is detected during solving.

Depot anchoring (optional):

- It matches standard TSPTW practice: schedules start at a depot at a known time (here $t = 0$).
- It avoids circular definitions (e.g., tying the first visit time to the last-to-first arc), making reasoning and debugging simpler.
- It produces interpretable arrival/wait times that align directly with each city's window.

Objective variants and backend: .

```

1 Vars = Path ++ Costs ++ Cities ++ CumTimes
2   ++ WaitTimes ++ [TravelDistance,
3     TravelTime],
4 if Type == min_distance then
5   solve([$degree,split,min(TravelDistance),
6         report(println(travelDistance
7           =TravelDistance))], Vars)
8 else
9   solve([$degree,split,min(TravelTime),
10        report(println(travelTime=
11          TravelTime))], Vars)
12 end.

```

Listing 11: Objective switch (distance vs time)

Explanation:

- **Decision vector:** `Vars` collects all variables for the solver.
- **Search options:** `degree` selects high-constraint-degree variables first; `split` uses domain splitting; `report(...)` logs objective values.
- **Objectives:** `TravelDistance` sums arc costs; `TravelTime` typically denotes the tour completion time (e.g., `CumTimes[Len]` or a separate accumulator).
- **Backend:** we use the CP backend which natively supports `circuit/1`, `element/3`, and integer relations `#=`, `#>=`, `#<=`. A MIP alternative would require a linear formulation (binary arcs, MTZ/flow, big- M time propagation).

Options and extensions:

- **Return-to-depot:** if needed, add `Return=Dist(Path[N], Path[1])` and `TotalTime=CumTimes[N]+Return` to optimize/limit full completion.
- **Lexicographic objective:** first minimize `TravelDistance`, then (with distance fixed) minimize `TravelTime` to avoid gratuitous waits under a distance-first policy.
- **Service times:** if each city has `S[city]`, set `Cum[I]=Arr[I]+Wait[I]+S[Path[I]]`.
- **Tighter domains:** bound `WaitTimes` by `0..max(UB-LB)` to improve propagation.

Alternative backends for time windows (SMT and MIP)

- **SMT:** Feasible if time propagation and window bounds are expressed with linear integer arithmetic and reified constraints; reuse the successor or edge-matrix view without CNF.
- **MIP:** Requires a linear formulation: binary arc variables, MTZ/flow order variables, and big- M constraints to propagate arrival/wait within windows. Practical on small/medium instances; needs a MIP solver (e.g., Gurobi via Picat's `mip`).

V. PHASE 4 — PERFORMANCE BENCHMARKING AND ANALYSIS

Phase 4 executes the models and records time and cost metrics.

A. Experimental setup

- Environment: Windows, Picat available on PATH; timing via PowerShell Measure-Command with Tee-Object to capture outputs.
- Instances: `tsp_1.pi` (SAT) and `tsp_2.pi` (CP) run on `data1()` (6-city) and `data10()` (10-city). `tsptw.pi` runs the built-in 21-city problem2.
- Objectives: TSP uses distance minimization. TSPTW uses distance-minimization; we also report the resulting makespan (`TravelTime`).
- Start city for TSPTW: flexible (no depot anchoring) to ensure feasibility on problem2.

B. Results

TABLE II: Solving time in seconds (lower is better)

oprule	Instance	SAT	CP	TSPTW
6 cities (TSP)	0.134	0.085	—	
10 cities (TSP)	0.227	0.078	—	
21 cities (TSP)	4.743	1145.364	—	
21 cities (TSPTW)	—	—	3.350	

TABLE III: Solution quality (distance; TSPTW also shows makespan)

oprule	Instance	SAT (dist)	CP (dist)	TSPTW (dist / time)
6 cities (TSP)	106	106	—	
10 cities (TSP)	155	155	—	
21 cities (TSP)	198	198	—	
21 cities (TSPTW)	—	—	378 / 387	

C. Dataset rationale and N/A entries

We use small (6-city) and medium (10-city) symmetric TSP matrices for quick, repeatable comparisons between SAT and CP. These are deterministic test matrices defined in `data.pi` (`data1()` and `data10()`) so results are stable across runs. For a large case, we report both classic TSP and TSPTW at 21 cities. The classic 21-city TSP distances are extracted from the `problem2` matrix in `tsptw.pi`, while TSPTW

uses the same dataset with its time windows (original model adapted from H. Kjellerstrand). Since SAT and CP baselines implement classic TSP (no windows), their cells remain N/A in the 21-city time-windowed row.

D. Additional solvers (SMT and MIP)

In line with the learning objective to contrast solver paradigms:

- **SMT (Z3):** We executed the SAT-style edge-matrix model under SMT (`tsp_1_smt.pi`) on the 6- and 10-city instances. Results are summarised in Table IV. The SMT backend was enabled by placing Z3 on PATH for the session.
- **MIP (Gurobi):** We prepared a linear MTZ formulation (`tsp_1_mip.pi`) with binary arc variables, degree equalities, MTZ subtour constraints, and linear cost. This was not executed due to the lack of a Gurobi license; therefore, MIP entries are omitted.

TABLE IV: SMT (Z3) results on classic TSP

oprule	Instance	SMT time (s)	SMT distance
	6 cities (TSP)	1.115	106
	10 cities (TSP)	3.116	155
	21 cities (TSP)	634.388	198

E. Discussion

Key observations from the runs:

- Small/medium TSP (6 and 10 cities): CP and SAT reach the same optimal costs; CP is faster or comparable (0.085 vs 0.134 s at 6; 0.078 vs 0.227 s at 10).
- Large TSP on the 21-city graph (distance-only): SAT solves in 4.743 s with cost 198, while CP took 1145.364 s to reach the same cost. This highlights instance-dependent behavior and that CP's default search may struggle on some structures without additional heuristics.
- TSPTW (21-city with windows): CP solves in 3.350 s under a distance-first objective, with distance 378 and makespan 387. Time windows change both feasibility and objective trade-offs compared to pure TSP.
- SMT (Z3) on 6/10/21 cities: matched SAT/CP optimum distances (106/155/198) with times about 1.115 s, 3.116 s, and 634.388 s respectively in our environment. Useful when mixing arithmetic and boolean structure without CNF; performance can be significantly slower than SAT/CP on larger instances.

Takeaways:

- CP is concise and very effective for small/medium TSP and for adding scheduling-style constraints (TSPTW).
- SAT can be competitive on larger TSP instances, as seen on the 21-city distance-only case.
- SMT complements SAT/CP for problems that benefit from rich reification and arithmetic reasoning; performance may lag on small TSPs compared to specialized SAT/CP encodings.

- MIP (linear MTZ) is a strong option when constraints are linear and industrial solvers are available; not executed here due to licensing, but typically competitive on small/medium TSP.
- Modeling implications: prefer CP for extensibility (time windows, service times). For hard distance-only instances, consider SAT or augment CP with symmetry breaking and search heuristics (e.g., `degree`, `first_fail`, or fixing a start city/arc). Use SMT/MIP when their constraint languages better match the problem (reification or linearity).

VI. CONCLUSION AND NEXT STEPS

We compared SAT and CP encodings for TSP and extended CP to TSPTW. On 6/10-city TSP, both reached the optimal distances, with CP faster (0.085/0.078 s) than SAT (0.134/0.227 s); on 21-city TSP, SAT (4.743 s) outperformed CP (1145.364 s). The CP-based TSPTW solved the 21-city instance in 3.350 s with distance/makespan 378/387. SMT (Z3) matched the same optimal distances on 6/10/21 but ran slower (1.115/3.116/634.388 s), showing flexibility at a runtime cost on larger cases. A linear MIP (MTZ) formulation was prepared but not executed due to licensing.

Optional next steps:

- Add CP search heuristics or symmetry breaking for the 21-city TSP to reduce runtime, and report the effect.
- Fix a random seed or provide a reproducibility note for any randomized components.
- Expand to additional datasets or objectives (e.g., lexicographic distance/time) if required by the course brief.

ACKNOWLEDGMENT

The project was prepared following the guidelines and notes collected during the course. Thank you to the course instructor, Professor João Carlos Silva in Computational Tools for Data Science, for the valuable guidance and feedback throughout the project.