A toolbox for hybrid metaheuristics: use cases

Anonymous author
Anonymous affiliation

Abstract

Hybrid metaheuristics are a common approach for tackling hard combinatorial optimization problems, due to their effectiveness in finding high-quality solutions beyond what can be achieved with traditional techniques such as propagation-based constraint programming (CP), integer linear programming (ILP) and related methods. This effectiveness comes from the ability to leverage the complementary strengths of multiple metaheuristic strategies, but the design and implementation of such solvers typically requires a significant amount of effort and expertise.

To address this difficulty, we present an extended and refined framework in which such solvers may be conveniently expressed and combined.

We provide an experimental evaluation of the prototype framework on feasibility and optimisation benchmarks, using Tabu Search (TS), Simulated Annealing (SA), Stochastic Hill Climbing (SHC), and Variable Neighbourhood Search (VNS) as leashed solvers under a shared wall-clock budget, And cover constraint satisfaction problems (CSP), combinatorial optimisation problems (COP), and parallel solver compositions.

2012 ACM Subject Classification Theory of computation → Design and analysis of algorithms; Computing methodologies → Search methodologies

Keywords and phrases hybrid metaheuristics, domain-specific language, combinatorial optimization, declarative programming, solver specification

Digital Object Identifier 10.4230/LIPIcs.CP.2026.

1 Introduction

Combinatorial optimization problems are widespread in computer science and operations research, with applications spanning domains such as logistics, telecommunications, finance and scheduling. Because most of these problems are NP-hard, metaheuristics have emerged as a powerful class of algorithms, effective at finding high-quality approximate solutions where exact methods are often impractical [18]. Meanwhile, the design, implementation, and tuning of metaheuristic solvers bring additional difficulties [4, 20]. The challenge is particularly significant for hybrid metaheuristics [3], which combine multiple algorithmic paradigms to leverage their complementary strengths and thereby achieve greater performance and robustness across a wide range of problem instances [12]. Nonetheless, this added flexibility comes at the cost of increased design and coding complexity. Traditional development practices typically result in monolithic, inflexible and hard-to-maintain codebases, hindering experimentation and code reuse.

Moreover, algorithms for different metaheuristics may be combined in such a way as to enable their parallel or distributed execution, leveraging the computational efficiency of multiple concurrent executions with relatively low effort, as discussed in [9].

The specification of hybrid metaheuristics, however, comes with several challenges, in particular:

1. Combining multiple algorithmic components, each with its own parameters and control logic leads to highly complex systems which significantly depart from their original components.
2. Defining how different components exchange information (e.g., solutions, search states) and how control is transferred between them is non-trivial.

3. Ad-hoc implementations often result in components that are tightly coupled and difficult to reuse in different contexts.
4. Exploring different hybrid configurations often requires significant reimplementation effort.

Several authors have explored language-based means of specifying hybrid solvers, i.e. ones where the structure of the solver is described using a formal language. Such is the case, e.g. of POSL [16] or MoSCO [8].

2 On the Design and Specification of Hybrid Metaheuristics

While no single metaheuristic performs well across all problem instances or types, hybrid metaheuristics leverage different complementary features, by integrating components from different metaheuristics (e.g., local search, population-based methods, construction heuristics), or by combining metaheuristics with other optimization techniques like integer programming. For instance, genetic algorithms excel at diversification and exploring new regions of the search space, while local search are better for intensification because they effectively exploit promising areas of the search space; perturbation operators help escape local extrema where greedy methods struggle; and conventional techniques reduce the search space and prune infeasible regions. By strategically combining such mechanisms, one can achieve a balance between exploration and exploitation that single-strategy methods cannot. This often leads to improved solution quality, greater robustness across problem instances, and superior computational efficiency.

3 The Framework: Modular Solver Architecture

The framework is built upon a modular architecture that constructs *solvers* by connecting a series of fundamental building blocks: *operators* and *connectors*. A solver is a graph (N, A) where N is a collection of nodes which may be operators or connectors. A are the edges of the graph and represents a stream of solution candidates, i.e. configurations for a given COP or CSP. A solver may be specified according to a Domain-Specific Language (DSL); see Section 3.1 for details.

The following describes the main components of the framework. We distinguish configurations, operators, connectors and leashed solvers in general terms; formal definitions of specific operators and their signatures are collected in Appendix A and Appendix B.

Configuration Streams

Solutions or sets of solutions, termed `configurations`, flow through the solver via `Configuration Streams`. These streams act as communication channels between operators, forming a directed graph $G = (V, E)$, where V is the set of operators and E represents the streams. A Configuration Stream $S : \mathcal{O}_{\text{source}} \to \mathcal{O}_{\text{destination}}$ transmits a set of configurations $C = \{c_1, c_2, \ldots, c_n\}$.

Operators

An operator $\mathcal{O}$ is a transformation function $\mathcal{O} : \mathcal{I}_{\mathcal{O}} \to \mathcal{O}_{\mathcal{O}}$, where $\mathcal{I}_{\mathcal{O}}$ denotes its input domain and $\mathcal{O}_{\mathcal{O}}$ its output range. In other words, operators consume configurations $\mathcal{A}$, or sets thereof $\mathcal{P}(\mathcal{A})$, and produce transformed configurations.

Connectors

Connectors organize the flow of configuration streams. They express branching and merging rather than direct transformations of configurations.

- **demux**: $\mathcal{O}_{\text{demux}} : \mathcal{A} \times \mathcal{N} \times \mathcal{B} \to \mathcal{P}(\mathcal{A})$. Dispatches an input configuration c to n output streams.
- **mux**: $\mathcal{O}_{\text{mux}} : \mathcal{P}(\mathcal{A}) \times \mathcal{N} \times \mathcal{S}_{sel} \to \mathcal{A}$. Recombines n branches into a single output configuration according to a strategy $s \in \mathcal{S}_{sel}$.

Leashed Solvers

A leashed solver encapsulates a complete metaheuristic, or a composition of operators, as a single controllable unit within a larger hybrid architecture. This abstraction allows "macro combinations" in which complete solvers, potentially themselves hybrid, may be deployed and orchestrated as black-box components under explicit resource limits. For instance, we can compose proven algorithmic building blocks (*Tabu Search*, *Simulated Annealing* and *Variable Neighbourhood Search*) each operating as a self-contained unit under its own budget. The primary solver governs when the leashed solver is invoked, what inputs it receives, and which resource limits apply (e.g., time, iterations). This is essentially a form of portfolio-style constructions in which distinct search strategies explore the search space under a common orchestration layer.

Examples in section 4.1.

3.1 Sketch of the Solver Description Language

The DSL is designed to allow for complex solver logic to be textually expressed as a composition of defined operations and control flows, where operators define transformations and their sequencing defines the solver's logic.

We will not give a complete formal definition of the language here: we will, instead, just sketch it to illustrate the main concepts, which are taken from the description in section 3.

```
Solver     -> Statement { ';' Statement }...

Statement  -> Assignment | Expression

Expression -> Operator | Pipeline | Loop | Branches

Operator   -> OperatorName | OperatorName(Parameters)

Parameters -> Parameter { ',' Parameter }...

Parameter  -> Identifier '=' Value

Value      -> Number | Boolean | Identifier | Operator | List

Pipeline   -> Expression { '|' Expression }...

Loop       -> loop(Expression)'*' |
              loop(Expression)'*'(Parameters)

Branches   -> '[' Expression { ',' Expression }... ']'
```

```
Assignment -> Identifier '=' Expression |
              Expression '>' Identifier
```

Operators are chained using '|' which designates a pipeline, i.e. a pipeline carrying a stream of configurations. Parentheses '()' group expressions, and square brackets '[]' denote branched sub-expressions (implicitly using demux/mux), with commas separating the branches; which may as well be executed in parallel. Parameters are passed as comma-separated key-value pairs (*Identifier* = *Value*), where values can be primitives, identifiers, lists, or nested operators, and variable binding allows explicit data flow management.

4 Use-cases

4.1 Designing New Hybrid Solvers

The DSL also allows new hybrid structures to be expressed by composing search strategies. For example, one may define a solver that switches local search strategies after stagnation, or a population-based algorithm where individuals are improved by different leashed solvers.

We designed pipelines for the QAP, TSP, and Costas-style permutation problems evaluated in Section 5. Three leashed solvers—SA, TS, and VNS—run in parallel and the best one is retained at a `mux`. Full DSL specifications are given in Appendix C.4.

For the QAP, each branch loops over a shared mixed neighbourhood that cycles through `swap`, `shift`, and `block` moves:

```
mixed_neighbourhood = demux(strategy=round_robin) [
    neighbourhood(move_type=swap),
    neighbourhood(move_type=shift),
    neighbourhood(move_type=block)
] | mux(strategy=best)

sa_solver = loop(mixed_neighbourhood | accept(criterion=temperature))
ts_solver = loop(mixed_neighbourhood | accept(criterion=tabu))
vns_solver = loop(mixed_neighbourhood | accept(criterion=improvement))

leashed_hybrid_solver = demux(strategy=parallel) [
    sa_solver, ts_solver, vns_solver
] | mux(strategy=best)

solver = init | gen(n=1) | leashed_hybrid_solver
```

The TSP solver has the same structure. The neighbourhood moves change to `two_opt`, `or_opt`, and `relocate`. The search starts from a nearest-neighbor heuristic and ends with a full-pass `two_opt` optimization:

```
sa_solver  = loop(neighbourhood(move_type=two_opt) | accept(criterion=temperature))

polish_step = loop(
    neighbourhood(move_type=two_opt(strategy=all_pairs)) |
    select(strategy=first_improvement)
)*(iterations=post_processing_iterations)

leashed_hybrid_solver = demux(strategy=parallel) [
```

```
    sa_solver, ts_solver, vns_solver
] | mux(strategy=best)

solver = init | gen(strategy=nearest_neighbor) | leashed_hybrid_solver
              | polish_step
```

5 Experimental Evaluation

We evaluate the prototype in three ways. First, CSP feasibility benchmarks to validate that the same solver description can drive a search toward a zero-violation assignment. Second, combinatorial optimisation pipelines on QAP and TSP instances. Third, studying the effect of parallel composition by keeping the leashed solvers fixed and changing only the way their branches communicate.

All runs use a time budget of 60 s unless otherwise stated. For feasibility problems, a run is successful when the violation-based fitness reaches 0. For optimisation problems, gaps are measured against the optimum when known, otherwise against the lower bound (best-known value). The local search parameters were fixed before the experiments and were not tuned per instance.

5.1 CSP Feasibility

The first group of experiments uses Magic Square, N-Queens, and Costas arrays as feasibility problems and small checks that the framework can run different constraint structures through the same search interface. The initial configuration does not necessarily need to be feasible, the fitness counts the remaining violations.

Table 1 CSP-style feasibility results. Values are success rates over repeated runs.

Problem	Instances	Success rate (60 s)	Success rate (300 s)
Magic Square	$N = 10$	100%	100%
	$N = 20$	78%	100%
	$N = 30$	0%	41%
N-Queens	$N = 1024$	100%	100%
	$N = 2048$	100%	100%
	$N = 4096$	0%	—
Costas arrays	CAP 15	100%	100%
	CAP 16	100%	100%
	CAP 17	100%	100%
	CAP 18	92%	—

Note that the Costas problem has a different feasibility structure from Magic Square and N-Queens while still using a permutation representation.

5.2 Optimisation problems

The second group of experiments compares the Hybrid Leashed solver with Hexaly [11], a strong commercial solver, on permutation optimisation benchmarks. The QAP instances are Taillard instances from QAPLIB [19]; the TSP instances are from TSPLIB [15]. The table reports average gaps after one minute, grouped by size.

On QAP, the hybrid solver is in the same range as Hexaly on these grouped instances. On TSP, Hexaly is clearly stronger, especially as the instance size grows. This is still useful

Table 2 Average gaps (%) after 60 s.

Problem	Size group	Hexaly	Hybrid Leashed
QAP	12–19	0.11	0.00
	20–30	2.09	1.73
	31–50	1.66	1.33
	51–80	1.93	1.56
	81–256	2.32	1.73
TSP	1–100	0.00	1.56
	101–1000	0.17	16.31
	1001–10000	1.29	70.92

information: the same DSL pattern gives a competitive QAP solver, while the TSP results show where the current move set and polishing steps built into the TSPspecific pipeline are not enough.

5.3 Parallel Behaviour

The last group of experiments is focused on the effect of the parallel composition topology. We use QAP because it is permutation-based, standard in the metaheuristics literature, and already used in studies of cooperative parallel search [19, 12]. We use the Taillard instances from QAPLIB, grouped by size n: $n \in [12, 19]$, $[20, 30]$, $[31, 50]$, $[51, 80]$, and $[81, 256]$. The same leashed solvers, the same instances, and the same seeds are used in every cell; only the way in which the parallel branches communicate is changed.

The portfolio contains four leashed solvers: SA, TS, SHC and VNS. Threads are assigned cyclically to this list, so that $T = 4$ runs one copy of each strategy and larger values repeat the same pattern with different seeds. Each branch runs under the same time budget and uses the same restart schedule. We compare three topologies. In the *independent* topology, branches do not communicate and the final *stream* keeps the best result. In the *cooperative* topology, all branches periodically publish their current best solution to a shared elite pool and may replace their current solution by a better pool member. In the *hierarchical* topology, the T branches are split into disjoint groups; each group has its own elite pool and the outer *stream* keeps the best group result.

Figure 1 gives a time-based view of runs. The logging records final gaps and time-to-target events, so the plot is a time-to-target figure rather than a full gap trajectory. Notice that there is a sharp rise after a plateau around the 60 seconds mark; at this stage we do not know whether this reflects solver dynamics, the restart schedule, the instance mix, or the logging and aggregation procedure. It requires further investigation.

Table 3 Topology effect for the hybrid leashed portfolio on QAP. Negative mean gap difference favours the first topology in the comparison.

Comparison	Threads	Pairs	Cliff's δ	BH q	Mean Δ gap
hierarchical vs. independent	8	248	-0.438	$< 10^{-6}$	-0.766
	16	249	-0.442	$< 10^{-6}$	-0.772
	32	220	-0.424	$< 10^{-6}$	-0.885
cooperative vs. independent	8	184	$+0.006$	$< 10^{-6}$	$+0.0005$
	16	182	$+0.003$	$< 10^{-6}$	$+0.0005$
	32	157	$+0.004$	$< 10^{-6}$	$+0.0001$

Table 3 shows that the hierarchical topology improves over an independent portfolio at

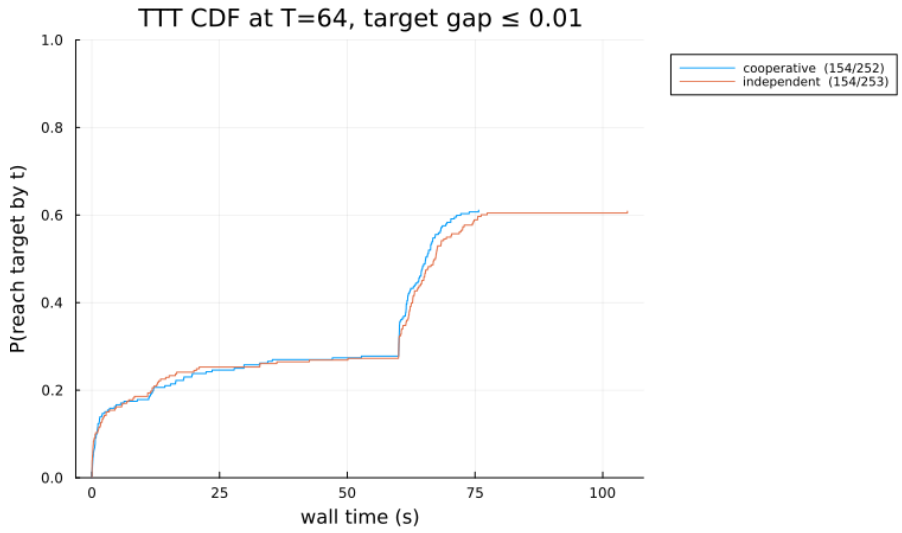

Figure 1 Time-to-target view of the parallel QAP runs on 64 threads.

every reported thread count, with a medium effect size under the Cliff's δ thresholds. A single shared elite pool, by contrast, gives no practically relevant improvement over independent search: the observed effect sizes are negligible and the mean gap differences are near zero. This suggests that, for this portfolio, the useful structure is not merely running more leashed solvers in parallel, but controlling the scope of cooperation. Group-local sharing preserves more diversity than a single global pool while still allowing the best candidate in the current search to circulate among related branches.

5.4 Prototype Implementation

A prototype of this framework is being developed in Julia. We chose Julia because of its strong metaprogramming capabilities, its flexibility and strong performance, which allowed us to rapidly prototype/implement the DSL parser, where the high-level syntax can be translated into executable solver components. While Julia itself is not strictly a declarative programming language, it allows for embedding the DSL and may capture a logic-based specification. The prototype comprises a DSL parser, a solver engine to interpret the parsed structure, a library of core components, and a problem interface to decouple the engine from specific problem details. Notably, the prototype is designed to parse XCSP3 [5] and MiniZinc [13] instances, to provide a bridge to existing constraint satisfaction problem benchmarks.

6 Related Work

The prototype just described sits within a broader line of frameworks for programming and composing metaheuristic search, each making different choices about how much of the search process is exposed as reusable structure. Since the prototype is designed to read XCSP3 and MiniZinc instances, it's worth noting that the closest CP-modelling side comparison is OscaR/CBLS [2], which connects a modelling front-end (i.e. MiniZinc) to a local-search engine. From the metaheuristic-framework side, **ParadisEO** [7] offers a white-box object-oriented framework for parallel and distributed metaheuristics, with support

for both population-based and trajectory-based methods. **jMetal** [10], **OptFrame** [14], **HeuristicLab** [17] and **MAHF** [21] provide modular libraries in which algorithms are assembled from reusable components. **POSL** [16] is closer in spirit to our work, since it uses a solver language to describe parallel combinations of metaheuristic and constraint-based search processes. Hyper-heuristic systems such as MatHH [1] also operate above the level of a single neighbourhood or acceptance rule by selecting or generating heuristics during search [6].

Our work is positioned between these lines of work. We do not propose a new metaheuristic, nor a new modelling language for constraints. Its focus is the explicit composition of solvers at two granularities in the same language: operators such as neighbourhood generation and acceptance, and *leashed solvers*, i.e. complete metaheuristics packaged as resource-bounded components. This makes portfolio-style, cooperative and hierarchical combinations expressible as solver graphs rather than as ad-hoc control code. The same constructs used to write a local-search pipeline can also be used to compose complete solvers and to vary their cooperation topology.

7 Conclusion and Directions for Future Work

In this paper, we presented a general framework for specifying and constructing hybrid metaheuristics, with the concept of *leashed solvers* as a key abstraction that enables "macro combinations" of partial usages of existing metaheuristic algorithms. This abstraction allows complete independent solvers to be encapsulated, controlled, and hierarchically composed within larger hybrid architectures. This ability to model sophisticated algorithms as reusable black-box components promotes code reuse, rapid experimentation, and better structural integrity compared to ad-hoc implementations. One expected benefit of such an approach is that end-users can focus on the conceptual and logical architecture of the solvers at various abstraction levels without concern for low-level details—bringing us one step closer to the holy grail ideal of Constraint Programming: the user specifies the problem, and the computer solves it.

Moving forward, our immediate next step is to further benchmark the framework on various problems, against various battle-tested solvers. Then to support alternative hardware architectures beyond shared-memory parallel processors, starting with distributed architectures and then GPU-accelerated environments. Another promising direction is to explore automated program generation LLM-assisted solver design, where the proposed DSL would serve as a constrained and executable target language for generating solver pipelines, leashed-solver portfolios, and cooperation topologies from high-level problem descriptions. Such generated programs could then be validated and executed directly by the framework, then compared across benchmark instances.

References

1 Ivan Amaya, Jorge M. Cruz-Duarte, and José C. Ortiz-Bayliss. MatHH: A Matlab-based hyper-heuristic framework. *SoftwareX*, 18:101047, 2022. doi:10.1016/j.softx.2022.101047.

2 Gustav Björdal, Jean-Noël Monette, Pierre Flener, and Justin Pearson. A constraint-based local search backend for MiniZinc. *Constraints*, 20(3):325–345, 2015.

3 Christian Blum, Maria Jose Blesa Aguilera, Andrea Roli, Michael Sampels, MariaJose Blesa Aguilera, Michael Samples, and Maria J Blesa Aguilera. *Hybrid metaheuristics*, volume 114. Springer, 2008.

4 Christian Blum and Andrea Roli. Metaheuristics in combinatorial optimization: Overview and conceptual comparison. *ACM computing surveys (CSUR)*, 35(3):268–308, 2003.

5 Frédéric Boussemart, Christophe Lecoutre, Gilles Audemard, and Cédric Piette. Xcsp3: an integrated format for benchmarking combinatorial constrained problems. *arXiv preprint arXiv:1611.03398*, 2016.

6 Edmund K. Burke, Michel Gendreau, Matthew Hyde, Graham Kendall, Gabriela Ochoa, Ender Özcan, and Rong Qu. Hyper-heuristics: A survey of the state of the art. *Journal of the Operational Research Society*, 64(12):1695–1724, 2013.

7 Sébastien Cahon, Nordine Melab, and E-G Talbi. Paradiseo: A framework for the reusable design of parallel and distributed metaheuristics. *Journal of heuristics*, 10(3):357–380, 2004.

8 Khalil Chrit, Jean-François Baffier, Pedro Patinho, and Salvador Abreu. An architecture for composite leashed combinatorial optimization solvers. In *14th Symposium on Languages, Applications and Technologies, SLATE 2025*, OASIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2025. (forthcoming).

9 Philippe Codognet, Danny Munera, Daniel Diaz, and Salvador Abreu. *Parallel Local Search*, pages 381–417. Springer International Publishing, Cham, 2018. URL: `https://doi.org/10.1007/978-3-319-63516-3_10`, `doi:10.1007/978-3-319-63516-3_10`.

10 Juan J Durillo and Antonio J Nebro. jmetal: A java framework for multi-objective optimization. *Advances in engineering software*, 42(10):760–771, 2011.

11 Hexaly Optimizer. Hexaly optimizer (formerly LocalSolver). `https://www.hexaly.com/`, 2024.

12 Danny Munera, Daniel Diaz, and Salvador Abreu. Hybridization as cooperative parallelism for the quadratic assignment problem. In *Hybrid Metaheuristics: 10th International Workshop, HM 2016, Plymouth, UK, June 8-10, 2016, Proceedings 10*, pages 47–61. Springer, 2016.

13 Nicholas Nethercote, Peter J Stuckey, Ralph Becket, Sebastian Brand, Gregory J Duck, and Guido Tack. Minizinc: Towards a standard cp modelling language. In *International Conference on Principles and Practice of Constraint Programming*, pages 529–543. Springer, 2007.

14 OptFrame. OptFrame: Optimization framework. `https://optframe.github.io/`, 2024.

15 Gerhard Reinelt. TSPLIB—a traveling salesman problem library. *ORSA Journal on Computing*, 3(4):376–384, 1991.

16 Alejandro Reyes-amaro, Éric Monfroy, and Florian Richoux. Posl: A parallel-oriented metaheuristic-based solver language. *Recent Developments in Metaheuristics*, pages 91–107, 2018.

17 Andreas Scheibenpflug, Andreas Beham, Michael Kommenda, Johannes Karder, Stefan Wagner, and Michael Affenzeller. Simplifying problem definitions in the heuristiclab optimization environment. In *Proceedings of the Companion Publication of the 2015 Annual Conference on Genetic and Evolutionary Computation*, page 1101–1108. Association for Computing Machinery, 2015.

18 Kenneth Sorensen, Marc Sevaux, and Fred Glover. A history of metaheuristics. *arXiv preprint arXiv:1704.00853*, 2017.

19 Éric D. Taillard. Robust taboo search for the quadratic assignment problem. *Parallel Comput.*, 17(4-5):443–455, 1991. `doi:10.1016/S0167-8191(05)80147-4`.

20 El-Ghazali Talbi. *Metaheuristics: from design to implementation*, volume 74. John Wiley & Sons, 2009.

21 Jonathan Wurth, Helena Stegherr, Michael Heider, Leopold Luley, and Jörg Hähner. Fast, flexible, and fearless: A rust framework for the modular construction of metaheuristics. In *Proceedings of the Companion Conference on Genetic and Evolutionary Computation*, page 1900–1909, New York, NY, USA, 2023. Association for Computing Machinery, Association for Computing Machinery. `doi:10.1145/3583133.3596335`.

A Operator Definitions

Operators

- **init**: $\mathcal{O}_{\text{init}} : \Theta \to \mathcal{S}$. Builds a solver-compatible representation $\mathcal{S}$ from the problem definition Θ (variables, domains, constraints and objective function).
- **generate**: $\mathcal{O}_{\text{generate}} : \mathcal{S} \to \mathcal{P}(\mathcal{A})$. Produces an initial set of configurations from $\mathcal{S}$.
- **neighbourhood**: $\mathcal{O}_{\text{neighbourhood}} : \mathcal{A} \times \mathcal{M} \to \mathcal{P}(\mathcal{A})$. Produces candidate neighbours of a configuration $c \in \mathcal{A}$ using a move $m \in \mathcal{M}$.
- **select**: $\mathcal{O}_{\text{select}} : \mathcal{P}(\mathcal{A}) \times \mathcal{S}_{sel} \to \mathcal{A}$. Chooses one configuration from the candidate pool according to a selection strategy $s \in \mathcal{S}_{sel}$.
- **accept**: $\mathcal{O}_{\text{accept}} : \mathcal{A} \times \mathcal{A} \times \mathcal{Q} \to \mathcal{A}$. Decides whether a candidate configuration replaces the current one according to an acceptance criterion $a \in \mathcal{Q}$.
- **loop**: $\mathcal{O}_{\text{loop}} : (\mathcal{A} \to \mathcal{A}) \times \mathcal{T} \times \mathcal{A} \to \mathcal{A}$. Repeats an operator or operator sequence Op until a termination condition $\tau \in \mathcal{T}$ is met.

B Moves and operator parameters

Moves

A move $m \in \mathcal{M}$ specifies how a configuration $c \in \mathcal{A}$ is transformed into one or more neighbouring configuration(s). Each move may be parameterized by a sampling strategy $s \in \mathcal{S}_{\text{samp}}$.

The default move is **assign**, which sets a variable x_i to a value v drawn from its domain D_i. The rest of the moves listed below preserve the validity of the solution structure and are better suited for permutation-based problems.

- **swap**: exchange the values at two positions (i, j).
- **two-opt**: reverse the sub-tour between positions i and j.
- **relocate**: remove the city at position *from* and reinsert it at position *to* (equivalent to Or-opt-1).
- **or-opt**: relocate a chain of 1–3 consecutive cities to a new position.
- **shift**: circularly rotate a random subsequence of length 2–5.
- **block**: swap two equal-size contiguous blocks of positions.

Note that the moves by default, will generate a single candidate, for example, the positions i and j for a **swap** move, or selected variable and a corresponding value for an **assign** move are chosen at random. Unless a sampling strategy $s \in \mathcal{S}_{\text{samp}}$ is provided.

- **single**: generate one candidate at random.
- **randomK(k)**: sample k candidates uniformly at random.
- **scanPairs**: fix one element at random, enumerate all pairings with it.
- **scanTargets**: fix a target slot or value at random, enumerate all sources that could fill it.
- **allPairs**: enumerate the full neighborhood in a single fused pass; no candidate buffer is used and improvements are applied in-place.

Parameters

Each operator and move type exposes parameters that control its behaviour. Parameters may be overridden to fine-tune search behaviour in any given part of the pipeline. Each parameter has a default value.

- **generate**:
  - `strategy` ∈ {`random`, `sequential`, `nearest_neighbor`} (default: `random`).
- **neighbourhood**:
  - `move_type`: a move $m \in \mathcal{M}$, possibly parameterized by a sampling strategy $s \in \mathcal{S}_{\text{samp}}$.
- **select**:
  - `strategy` ∈ {`best`, `random`} (default: `best`).
- **accept**:
  - `criterion` ∈ {`always`, `improvement`, `probability`, `temperature`, `tabu`}.
    - `probability`: parameter `p` (default 0.5).
    - `temperature`: parameters `t0` (default 1000.0) and `alpha` (default 0.95); non-improving moves are accepted with probability $\exp(\Delta/T)$.
    - `tabu`: parameter `tabu_size` (default 7).
- **loop**:
  - `iterations`: Maximum number of iterations (required).
  - `target_fitness`: Fitness value at which the loop terminates early (optional; enables race-to-solution semantics in parallel contexts).
  - `time_limit`: Wall-clock time limit in seconds (optional).
  - `store_history`: Whether to record the fitness trace (default: `false`).
  - `restart`: optional restart schedule, e.g. `luby(base_unit = b)`.
- **demux**:
  - `strategy` ∈ {`parallel`, `round_robin`, `random`, `weighted`, `adaptive`, `indexed`}.
    - `weighted`: uses user-defined weights.
    - `adaptive`: adjusts branch selection online according to a success metric, e.g. frequency of improvement.
    - `indexed`: routes to the branch designated by an index variable.
- **mux**:
  - `strategy` ∈ {`best, random, indexed`}.

C Benchmark Problem Definitions

C.1 Costas arrays

A Costas array may be encoded by a permutation π of $\{1, \ldots, n\}$ (one mark per row and column on an $n \times n$ board) such that for every pair of marks (i, π_i) and (j, π_j) with $i < j$, the displacement $(j - i,\ \pi_j - \pi_i)$ is unique. Equivalently, the multiset of positive differences $|\pi_j - \pi_i|$ at each offset $j - i$ must not repeat. Infeasibility is measured by counting constraint clashes; the search operators are the same permutation moves as for QAP.

C.2 The Quadratic Assignment Problem

The Quadratic Assignment Problem (QAP) models the following: there are a set of n facilities and a set of n locations. For each pair of locations, a distance is specified and for each pair of facilities a weight or flow is specified. The problem is to assign all facilities to different locations with the goal of minimizing the sum of the distances multiplied by the corresponding flows. Given two sets, P ("facilities") and L ("locations"), of equal size, together with a weight function $w : P \times P \to \mathbb{R}$ and a distance function $d : L \times L \to \mathbb{R}$. Find the bijection $f : P \to L$ such that:

$$\sum_{a,b \in P} w(a,b) \cdot d(f(a), f(b))$$

is minimized.

C.3 The Travelling Salesman Problem

The Travelling Salesman Problem (TSP) is one of the most studied combinatorial optimization problems. Given a set of n cities and a distance function $d : C \times C \to \mathbb{R}^+$ where $C = \{c_1, \ldots, c_n\}$, the goal is to find a Hamiltonian cycle of minimum total length, i.e., a permutation π of $\{1, \ldots, n\}$ that minimizes:

$$\sum_{i=1}^{n} d(c_{\pi(i)}, c_{\pi(i \bmod n+1)})$$

C.4 Hybrid Solver Specifications

QAP-specific pipeline

```
mixed_neighbourhood =
    demux(strategy=round_robin) [
        neighbourhood(move_type=swap),
        neighbourhood(move_type=shift),
        neighbourhood(move_type=block)
    ] | mux(strategy=best)

sa_solver  = loop( mixed_neighbourhood
                 | accept(criterion=temperature,
                          t0=10000, alpha=0.995)
             )*(iterations=it, restart=luby(base_unit=50k))

ts_solver  = loop( mixed_neighbourhood
                 | accept(criterion=tabu, tabu_size=10)
             )*(iterations=it, restart=luby(base_unit=50k))

vns_solver = loop(
    demux(strategy=indexed, index=k) [
        neighbourhood(move_type=swap),
        neighbourhood(move_type=shift),
        neighbourhood(move_type=block)
    ] | accept(criterion=improvement) > success_flag;
```

```
    k < if(success_flag, 1, min(k+1, 3))
)*(iterations=it, restart=luby(base_unit=50k))

leashed_hybrid_solver =
    demux(strategy=parallel) [
        sa_solver, ts_solver, vns_solver
    ] | mux(strategy=best)

solver = init | gen(n=1) | leashed_hybrid_solver
```

TSP-specific pipeline

```
sa_solver  = loop( neighbourhood(move_type=two_opt)
                 | accept(criterion=temperature,
                          t0=10000, alpha=0.995)
             )*(iterations=it, restart=luby(base_unit=20k))

ts_solver  = loop( neighbourhood(move_type=two_opt)
                 | accept(criterion=tabu, tabu_size=10)
             )*(iterations=it, restart=luby(base_unit=20k))

vns_solver = loop(
    demux(strategy=indexed, index=k) [
        neighbourhood(move_type=two_opt),
        neighbourhood(move_type=or_opt),
        neighbourhood(move_type=relocate)
    ] | accept(criterion=improvement) > success_flag;
    k < if(success_flag, 1, min(k+1, 3))
)*(iterations=it, restart=luby(base_unit=20k))

polish_step = loop(
    neighbourhood(move_type=two_opt(strategy=all_pairs))
    | select(strategy=first_improvement)
)*(iterations=post_processing_iterations)

leashed_hybrid_solver =
    demux(strategy=parallel) [
        sa_solver, ts_solver, vns_solver
    ] | mux(strategy=best)

solver = init
       | gen(n=1, strategy=nearest_neighbor)
       | leashed_hybrid_solver
       | polish_step
```

CA-specific pipeline

```
mixed_neighbourhood = neighbourhood(move_type=swap)
```

```
509  sa_solver  = loop(mixed_neighbourhood | accept(criterion=temperature))
510  ts_solver  = loop(mixed_neighbourhood | accept(criterion=tabu))
511  vns_solver = loop(mixed_neighbourhood | accept(criterion=improvement))
512
513  leashed_hybrid_solver =
514      demux(strategy=parallel) [ sa_solver, ts_solver, vns_solver ]
515      | mux(strategy=best)
516
517  solver = init | gen(n=1) | leashed_hybrid_solver
```