

EasyLocal – Reference

A C++23 framework for local search and metaheuristics

Contents

Reference	4
Header layout	4
Problem model	5
Contract	5
Optional hooks	5
Reading and writing	6
Convenience bases	6
Design choices	7
SolutionManager	8
Contract	8
Solution identity	8
Convenience base	9
The services of a run	9
Recipes	9
Design choices	10
Cost	11
Cost components	11
Cost expressions	11
Cost models	13
Cost semantics	15
Delta cost components	16
Design choices	16
NeighborhoodExplorer	18
Contract	18
Parameters	19
Convenience base	19
Recipes	20
Neighborhood unions	20
Design choices	20
Runners	21
Composing a runner	21
Built-in algorithms	22
Results	28
Writing an algorithm	28
Run options	31
Design choices	32
Solvers	33
Construction	33
Built-in solvers	33
Pipeline	34
Exceptions	37
Initialization	37
Design choices	37
Configuration	38
Parameter blocks	38
Parameter sets	41
Configurable objects	42
Values as text	44
Frontends	46

Design choices	46
Apps and tools	48
Building an app	48
Tools	51
Sessions	53
Design choices	57
Testing	58
Fixture	58
Checks	59
Design choices	59

Reference

One page per component: its contract (what is required, what is optional and who uses it), its API, and the design choices behind it. The [tutorial](#) introduces the same components by example. The [generated API reference](#) lists every public class, function and concept, generated by MrDocs from the `///` comments of the headers.

Page	Header(s)	Namespace
Problem model	—	your code
SolutionManager	helpers/solution_manager.hpp	easylocal
Cost	cost.hpp, helpers/recipes.hpp	easylocal::cost, easylocal
NeighborhoodExplorer	helpers/ neighborhood_explorer.hpp, helpers/ neighborhood_union.hpp	easylocal
Runners	runners.hpp, runners/*.hpp	easylocal, easylocal::runners
Solvers	solvers.hpp, solvers/*.hpp	easylocal::solvers, easylocal::initialization
Configuration	config/*.hpp, adapters/ toml.hpp	easylocal::config
Testing	testing.hpp	easylocal::testing
Apps and tools	app/*.hpp, adapters/tui.hpp, adapters/rest.hpp	easylocal, easylocal::tui, easylocal::rest

Topic guides: [tracing](#), [logging](#), [REST](#), [dependency policy](#).

Header layout

easylocal/ easylocal.hpp	Core umbrella (all but adapters/, app/cli.hpp, app/tuning.hpp and the component checks of testing.hpp, whose report and options types come with check(app))
utils/	generator, limit, termination_reason, solution hashing, logging; internal utilities
config/	typed parameters, parameter sets, CLI/file frontends
trace/	semantic search events, tracer protocol, recorders
cost/	cost models, cost expressions, semantics, costs as text
helpers/	SolutionManager, NeighborhoodExplorer, neighborhood_union, recipes
runners/	Runner, search_run, run_control, search algorithms
solvers/	solvers
testing/	component contract checks
app/	App and BoundApp, check(app), Session, I/O hooks, RunParameters, cli::run and irace tuning
adapters/	optional components: toml.hpp, tui/, rest/

A directory includes only directories listed above it (utils/, config/ and trace/ share the lowest layer; testing/ sits beside runners/); the architecture test enforces it. detail/ subdirectories are implementation headers. Catalogs of interchangeable implementations have a namespace (runners, solvers, cost, trace, config, testing); the vocabulary used everywhere (Runner, make_runner, solution_manager, component, neighborhood, delta, app, ...) is in easylocal.

Problem model

The problem is described by three value types you define: **Input**, **Solution** and **Move**. The framework imposes no base class and no member on them; the hooks below are optional and used only by the tools.

Contract

Type	Requirements	Notes
Input	none	built once, immutable while services are bound to it
Solution	copyable, movable	does not reference the Input
Move	copyable	does not reference a Solution; default-constructible if the explorer uses a cursor

Optional hooks

Each kind of hook may be written in three ways, tried in this order:

Hook	Forms, in order	Used by
read an Input	static Input::read(std::istream&); read_input(std::type_identity<Input>, TextUI std::istream&) by ADL; operator>> on a default-constructed Input	read_input, load_input, Session, TextUI
read a Solution	static Solution::read(const Input&, std::istream&); read_solution(const Input&, std::istream&) by ADL; operator>> on Solution{input}	read_solution, load_solution, Session, TextUI
write a Solution	Solution::write(const Input&, std::ostream&) const; write_solution(const Input&, const Solution&, std::ostream&) by ADL; operator<<	write_solution, save_solution, Session, TextUI
describe a value	describe() const member; describe(const T&) by ADL; operator<<; a std::variant, or a union move, by what it holds	describe, TextUI
read a cost	read_cost(const Input&, std::string_view) by ADL; else cost::from_text	read_cost, Session, TextUI, RunParameters (see Cost)
compare solutions	the SolutionManager's equal(const Solution&, const Solution&); operator== on Solution (solution identity)	the checks of Session::testing, Session::check_move_independence, TextUI, reactive tabu list
compare moves	operator== on Move	tests, Session and TextUI

The TextUI shows a Solution without a describe hook as its write hook writes it.

Reading and writing

<easylocal/app/io.hpp> reads and writes values through the hooks, in any program:

Function	
<code>read_input<Input>(std::istream&), load_input<Input>(path)</code>	an Input
<code>read_solution<Solution>(input, std::istream&), load_solution<Solution>(input, path)</code>	a Solution of input
<code>write_solution(input, solution, std::ostream&), save_solution(input, solution, path)</code>	writes a Solution
<code>describe(value) -> std::string</code>	the text of a value, for people

The concepts `readable_input<Input>`, `readable_solution<Input, Solution>`, `writable_solution<Input, Solution>` and `describable<T>` tell whether a hook exists; `has_describe<T>` whether T has a describe of its own, member or free, not just operator<<. The functions throw `std::runtime_error` when a stream fails (or what a hook throws); the file functions report errors as `std::runtime_error` naming the file, for `save_solution` those of the final flush and close too. `write_solution` and `describe` are function objects, so the argument-dependent lookup of a problem's own hook never finds them.

Convenience bases

Two optional, non-virtual bases give a class what it needs to receive the Input and its parameters, so that every service does it the same way:

Base	Header	Gives
<code>easylocal::input_base<Input></code>	<code><easylocal/ utils/input_base.hpp></code>	<code>input_type</code> , a constructor from <code>const Input&</code> (the one from a temporary is deleted) and <code>input()</code>
<code>easylocal::parameters_base<Parameters></code>	<code><easylocal/ config/parameters_base.hpp></code>	<code>parameters_type</code> , which makes the class <code>configurable</code> , a constructor from the block and <code>parameters()</code>

```
class TourLength : public easylocal::input_base<Tsp>
{
public:
    using input_base::input_base;    // the constructor, which input() reads back

    double evaluate(const Tour& tour) const { /* ... input().distance ... */ }
};
```

- Both are optional: a class that declares the same members itself does just as well, and a cost component that needs neither the Input nor parameters derives from nothing.
- Inherit the constructor with `using input_base::input_base;` (or `using parameters_base::parameters_base;`). A class that derives data from the Input writes its own con-

structor instead and passes the Input on: `explicit C(const Input& input) : input_base{input}, table_{build(input)} {}`.

- Derive from each at most once: a cost component with a co-located delta is one class, with one Input.
- `solution_manager_base` and `neighborhood_explorer_base` derive from `input_base`, so the Input reaches a `SolutionManager`, an explorer and a cost component the same way.
- `parameters_base` does not validate the block: the recipe that holds it validates it before every construction.
- A class that takes both is built from the Input first, then the parameters, the order the recipes pass them in:

```
class WeightedLength
    : public easylocal::input_base<Tsp>,
      public easylocal::parameters_base<WeightParameters>
{
public:
    WeightedLength(const Tsp& input, WeightParameters parameters)
        : input_base{input}, parameters_base{std::move(parameters)} {}
};
```

Design choices

- **Plain values.** Entities carry no references to each other, so a Solution can be copied, stored and compared freely, and a Move can be applied to any compatible Solution. Behaviour lives in services (`SolutionManager`, `NeighborhoodExplorer`), not in the values.
- **Immutable, borrowed Input.** Services borrow the Input by `const&` for their whole lifetime; binding a temporary Input is rejected. Concurrent runs share only the Input.
- **Who receives the Input.** Three kinds of code, three rules:
 - values (`Input`, `Solution`, `Move`) hold no Input;
 - services (`SolutionManager`, `NeighborhoodExplorers`, cost components, delta cost components) receive it once, in their constructor, when a runner or an app is bound, and keep it; their members take only the solution and the move;
 - hooks, free functions that are not services (`read_solution`, `write_solution`, `read_cost`), receive it as a parameter.

A service is built once per Input, so its constructor is where data derived from the instance is computed, such as the conflicts of each exam or a neighbour list: once per bind, owned by the service that uses it, and kept out of the Input, which stays the instance as read. Passing the Input to every call instead would leave such data nowhere but in the Input itself or in a mutable cache.

- **Structural vs. feasible.** Validity (see [SolutionManager](#)) is about the representation; constraint violations are part of the cost.

SolutionManager

<easylocal/helpers/solution_manager.hpp>

The SolutionManager defines *solution semantics*: which solutions are valid and how to build them. It does not compute the cost; see [Cost](#).

Contract

Member	Required	Used by
input_type, solution_type	yes	everything
input() const -> const input_type&	yes	everything
is_valid(const solution_type&) const -> bool	yes	debug assertions, checks, Session and TextUI
initial_solution() const -> solution_type	no	initialization::initial, bound runner's initial_solution(), Session and TextUI
random_solution(RNG&) const -> solution_type	no	initialization::random, MultiStart, Session and TextUI
hash(const solution_type&) const -> std::uint64_t	no	solution identity, see below
equal(const solution_type&, const solution_type&) const -> bool	no	solution identity, see below

Concepts: `base_solution_manager<SM>` (the required members), `has_initial_solution<SM>`, `has_random_solution<SM, RNG>`, `has_solution_hash<SM>`, `has_solution_equality<SM>`.

Solution identity

Algorithms and tools that recognize a solution met before, such as a reactive tabu list or a trace of the search trajectory, use `easylocal::solution_hash(sm, solution)` and `easylocal::solutions_equal(sm, lhs, rhs)`. They come from the SolutionManager's `hash` and `equal` members when it has them, otherwise from the solution type's `std::hash` and `operator==`. The members take precedence, so a problem can leave out redundant data (caches, derived matrices) or identify symmetric representations, such as a tour and its reverse. Equal solutions must have equal hashes. Nothing requires them: only the features that recognize solutions do.

<easylocal/utils/hash.hpp> helps to write a hash:

```
std::uint64_t hash(const Tour& tour) const
{
    return easylocal::hash_range(tour.order);    // folds each element
}
// easylocal::hash_combine(seed, value) folds one value into a running hash
```

Both depend on the order of the values and on `std::hash`, so a hash may differ between standard libraries.

The `SolutionManager` is constructed from the `Input` when a runner is bound: `SM(const Input&, args...)`, where `args` come from the recipe `solution_manager<SM>(args...)` and convert as the constructor takes them.

A `SolutionManager` whose `parameters_type` is a parameter block is constructed from it, after the `Input`: `SM(const Input&, const parameters_type&, args...)`. `solution_manager<SM>(parameters, args...)` gives them (without a first argument of that type, they are the defaults); the recipe holds them, and a runner and an app expose them at their root, as `solution_manager.*`, beside `cost.*` (see [Configuration](#)).

Convenience base

```
template<class Input, class Solution>
class solution_manager_base; // input_type, solution_type, input()
```

Inherit its constructor with `using solution_manager_base::solution_manager_base;`. It is non-virtual and optional. The `Input` comes from `input_base`, which it derives from, as a cost component and a `NeighborhoodExplorer` do: it keeps the `Input` by reference, so the constructor from a temporary `Input` is deleted and the `Input` outlives the `SolutionManager`.

The services of a run

A `SolutionManager` may declare the state its components share while they search: a cache, the loads of each resource, the network of a declared model.

```
class MySolutionManager : public easylocal::solution_manager_base<Input, Solution>
{
public:
    using services_type = MyCache; // one per bind, from the Input
    ...
};
```

One `services_type` is built for each bind — `app.bind(input)`, `runner.bind(input)`, a `testing::fixture` — from the `Input` when it takes one and without arguments otherwise, and handed to whatever asks for it in its constructor:

```
MyComponent(const Input&, MyCache&, args...) // a cost component
MyDelta(const Input&, MyCache&, args...) // a delta cost component
MyExplorer(const MySolutionManager&, MyCache&, args...) // an explorer
```

A component that does not take them is built from the `Input` as before. The services are the mutable state of a run: a const `SolutionManager` hands them out, and what they hold changes while the search runs.

They are never copied and never shared between two binds, so two runs never share them — which is what makes a search on several threads safe: each thread binds its own app, as the `TextUI` does. Keep what the services hold, not the `SolutionManager`: the layers of a bound `SolutionManager` are moved as they are composed, while the services keep their address.

Recipes

```
solution_manager<SM>(args...) // the SolutionManager
| component<C>(args...) // its cost: one component, or a cost
// expression over several, see Cost
// equivalently: solution_manager<SM>(args...).with_cost(component<C>(args...))
```

A recipe without a cost is rejected: the cost always comes from cost components. A recipe has exactly one cost expression.

Design choices

- **Validity is structural.** `is_valid` checks that the representation is well formed. A solution violating problem constraints is valid and costly. A debug build asserts it at the start of a run and after each move made; after each move evaluated, which costs as much as the search, only with `EASYLOCAL_EXPENSIVE_CHECKS` defined (the CMake option of the same name defines it for EasyLocal's own tests and examples, the asan preset sets it).
- **Construction is optional and explicit.** A runner starts from a solution you provide; only solvers and tools need `initial_solution` or `random_solution`. There is no automatic fallback between the two.
- **Randomness is passed in.** `random_solution(rng)` receives the generator, so the caller (usually a solver) controls seeding and replay.
- **No cost in the SolutionManager.** A `SolutionManager` does not evaluate the cost: every problem has the same cost layer, so delta evaluation, weighting and per-component diagnostics are always available; a monolithic objective is a single cost component.

Cost

<easylocal/cost.hpp> (cost models and cost expressions, `easylocal::cost`) and <easylocal/helpers/recipes.hpp> (component, delta)

The cost of a solution is computed by **cost components**, combined by a **cost expression** into a value of a **cost type**. Costs are minimized: a lower cost is better; to maximize, return what is lost or the negated profit, or give a root cost::apply its own compare (Great Deluge, whose levels are fractions of the cost, needs positive costs). Moves are evaluated incrementally by **delta cost components**. Together with the SolutionManager they form the *cost layer*; the delta cost components bound to a NeighborhoodExplorer form the *delta cost layer*.

Cost components

Member	Required
<code>evaluate(const Solution&) const -> Value</code>	yes
<code>delta_evaluate(const Solution&, const Move&) const -> Delta</code>	no: a co-located delta cost component
<code>name() -> std::string_view (static or not)</code>	no: its name in reports, otherwise its position, #1; static, and required, for a component with parameters, which is configured under it
<code>parameters_type</code>	no: a parameter block, which makes the component configurable (<code>cost.<name>.*</code>); the component is constructed from it
<code>describe(const Solution& const -> std::string</code>	no: a text that explains its value on a solution, such as the violations it counts

- Value is deduced from `evaluate` and may be arithmetic or a domain type, but not an unsigned integer: the difference of two costs would wrap around, so a component that counts returns `int` or `long`.
- Construction: `Component(const Input&, args...)` is preferred, `Component(args...)` is accepted; `args` come from `component<C>(args...)` and convert as the constructor takes them. The same holds for a delta cost component and `delta<C, D>(args...)`.
- Deriving from `easylocal::input_base<Input>` gives the component that constructor and `input()`, and deriving from `easylocal::parameters_base<P>` gives it `parameters_type`, the constructor from the block and `parameters()`; both are optional, and a component that needs neither derives from nothing.
- With parameters: `Component(const Input&, const parameters_type&, args...)` (or without the `Input`), from `component<C>(parameters, args...)`; without a first argument of that type, the parameters are the defaults. The recipe holds them, and builds the component from them when a runner or an app is bound.
- A component type may appear only once in a cost expression: the type is the identity of the component. To use one parametric component twice (two thresholds), derive a named type for each, `struct LongEdges8 : LongEdges<8> {};`, or give each its own parameters as a class.
- `name()` and `describe(solution)` are for people: `Session::cost_report()`, `cli::run's --report` and the TextUI's solution window show each component's value with them. Without `describe`, only the value is shown.

Cost expressions

A SolutionManager recipe has exactly one cost expression: `solution_manager<SM>() | expression`, or `.with_cost(expression)`. Its leaves are cost components; its nodes combine the costs of their children.

Expression	Cost	Configuration
<code>component<C>([parameters,] args...)</code>	the value of the component	its <code>parameters_type</code> 's, under <code><name></code>
<code>cost::sum(t1, ..., tn)</code>	$\sum w_i \cdot \text{cost}_i$, in the common type of the costs and the weights	weights
<code>cost::weighted(child, w)</code> , or <code>child * w, w * child</code>	a term of <code>cost::sum</code> with weight <code>w</code> (default 1)	
<code>cost::in_order(c1, ..., cn)</code>	<code>cost::lexicographic</code> of the children's costs	children's
<code>cost::objectives(c1, ..., cn)</code>	<code>cost::pareto</code> of the children's costs ($n \geq 2$)	children's
<code>cost::hard_soft(hard, soft)</code>	<code>cost::hierarchical</code> of the two costs	hard, soft
<code>cost::apply(f, c1, ..., cn)</code>	<code>f(cost₁, ..., cost_n)</code>	children's
<code>cost::apply<F>(parameters, c1, ..., cn)</code>	<code>F{parameters}(cost₁, ..., cost_n)</code>	<code>F::parameters_type</code> 's, under <code><name></code> , and children's
<code>cost::approximately(c, tolerance)</code>	the cost of <code>c</code> , compared by the search within a <code>cost::tolerance</code> (at the root)	<code>tolerance.relative</code> , <code>tolerance.absolute</code> , and the child's as they are

```

solution_manager<SM>()
| cost::hard_soft(
|   cost::sum(component<A>(), component<B>() * 10),
|   cost::in_order(component<C>(), cost::apply(f, component<D>())))

```

- `cost::sum` adds arithmetic costs only; a domain value is turned into a number with `cost::apply`. A weighted term, `cost::weighted(child, w)` or its shorthand `child * w`, is allowed only directly inside a `cost::sum`.
- The components of the hard branch of a `cost::hard_soft` at the root come first: a pipeline stage `until_feasible()` (the first stage of `two_stage()`) evaluates only them.
- At the root, the function of a `cost::apply` may define the order of its costs, `compare(a, b)`, and `cost::approximately` compares them within a tolerance (see Cost semantics).
- Configuration: the expression is exposed under `cost`. A sum has its weights (one per term, finite: a negative weight is accepted, NaN and infinity are not), a `hard_soft` names its children `hard` and `soft`, `in_order`, `objectives` and `apply` name them by position (0, 1, ...); only configurable children appear. In the example above: `cost.hard.weights`.
- A component or a `cost::apply` function whose `parameters_type` is a parameter block is configured under its static `name()`, wherever it is in the expression: `cost.<name>.*`. Two of them with the same name make the configuration throw `std::invalid_argument`. A function with parameters is built from them, `cost::apply<F>(parameters, children...)`, and built again when they change; `cost::apply(F{...}, children...)` does not compile for it. A component or a function that has `parameters()` or `configuration()` and no such `parameters_type` does not compile.

Cost models

Type	Ordering	cost::delta
arithmetic (cost::arithmetic: signed integers and floating point)	<	candidate - current
cost::lexicographic<Ts...>	lexicographic over the values	none
cost::pareto<Ts...>	Pareto dominance (a partial or- der)	none
cost::hierarchical<Hard, Soft>	hard first, soft when hard is equivalent; each branch by its <=>, or else by < and ==	hard better: $-\infty$, hard worse: $+\infty$, else the soft delta

The structured types are constructed directly, with deduced types: `cost::hierarchical{hard, soft}`, `cost::lexicographic{a, b}`, `cost::pareto{a, b}`. The concepts `cost::lexicographic_cost`, `cost::hierarchical_cost` and `cost::pareto_cost` recognize them.

Pareto costs

A `cost::pareto` holds one value per objective, all minimized, and orders them by Pareto dominance, which leaves some costs unordered. With two objectives, makespan and tardiness of a schedule for example:

```
solution_manager<ScheduleManager>()
| cost::objectives(component<Makespan>(), component<Tardiness>())
```

each solution is a point of the plane:

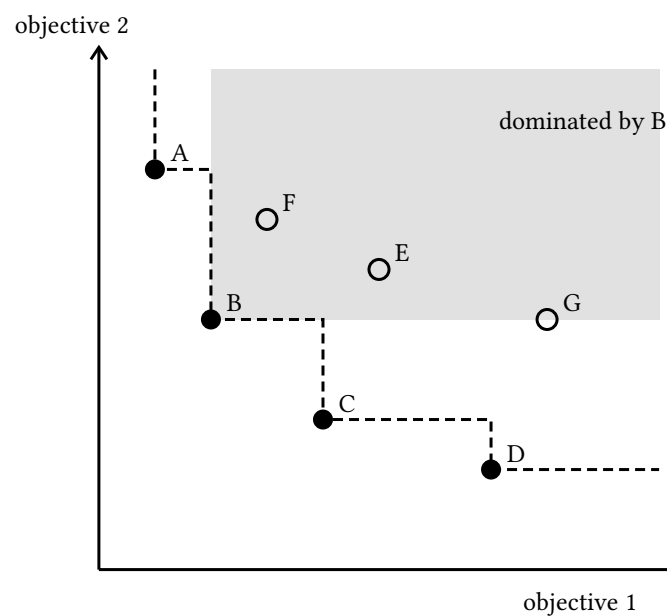


Figure 1: A, B, C and D are the Pareto front (the staircase); E, F and G are dominated. The shaded region is what B dominates.

The cost semantics read as follows, for costs a and b:

- `better(a, b)`, `a < b`: a dominates b, no worse in every objective and better in at least one. B is better than E and G, which lie in the region it dominates; C is better than E.
- `better_or_equivalent(a, b)`, `a <= b`: a weakly dominates b, no worse in every objective, equal costs included.
- `equivalent(a, b)`, `a == b`: equal in every objective.
- Neither `better(a, b)` nor `better(b, a)`: the costs are unordered, `a <=> b` is `std::partial_ordering::unordered`. A and B are, as are any two points of the front: each is better in one objective.

So a descent (First and Best Improvement, Hill Climbing) moves from E to B or C, never from B to A, and from a point of the front only to a point that dominates it. The Pareto front is the set of the points that no other dominates, and a search with a pareto cost keeps the non-dominated solutions it reaches in its archive and returns them as its front (see [Runners](#)). Pareto Late Acceptance Hill Climbing explores from several solutions at once to spread over the front. An app run by name carries the front in `named_run_result.front`, the Session keeps it (`last_run_front()`), `cli::run` prints it and REST returns it with the solution (see [Apps and tools](#)); the TextUI shows only the current solution.

The runners whose acceptance only compares costs accept a pareto cost: First and Best Improvement, Hill Climbing, Late Acceptance, Pareto Late Acceptance and the tabu searches, except Aspiration Plus and Elite Candidate, whose levels are factors of the cost, with any tabu list but `Foo` and `RandomFoo`, which need a delta. Late Acceptance and Tabu Search may move to a point that does not dominate the current one: Late Acceptance accepts a candidate no worse than an older cost, and Tabu Search applies the best admissible move even when it worsens, choosing among equivalent candidates only, so the first of two unordered ones stays. The runners that need a numeric delta (Simulated Annealing, Great Deluge) do not accept a pareto cost.

`cost::delta(candidate, current)` is the numeric difference used by delta-based acceptance; user cost types provide it as a free function found by ADL (call it as using `cost::delta; delta(a, b)`). The concept `cost::has_delta<Cost>` tells whether it exists.

The difference of two arithmetic costs is computed in a type that represents it, which the cost type itself need not: the delta of an integral cost narrower than 64 bits is an `std::int64_t`, exact for every pair of costs; that of a 64-bit one a double, which rounds differences above 2^{53} ; and a floating-point cost keeps its own type. A user cost type's own delta is responsible for the same: the difference of two costs that their type holds may not fit in it.

`cost::zero<Cost>()` is the cost of no violation and no penalty: `Cost{}` for value-initializable types (0 for arithmetic costs) and the zero of every level for lexicographic, hierarchical and pareto costs. Other cost types provide it by specializing `cost::zero_cost<Cost>` with a static `value()`; `cost::has_zero<Cost>` tells whether it exists. A pipeline stage `until_feasible()` stops at the zero of the hard cost, also when deltas leave a floating-point hard cost a rounding error above it: the run then evaluates the solution in full (see [Runners](#)).

Comparisons within a tolerance

`<easylocal/cost/tolerance.hpp>` compares costs forgiving the rounding errors of floating-point arithmetic. `cost::tolerance{.relative, .absolute}` (both $1e-9$ by default) is the tolerance: two floating-point values are equal within it when they differ by at most `max(absolute, relative * max(|a|, |b|))`.

Function	Result
<code>cost::approximately_equal(a, b, tolerance),</code> <code>or tolerance(a, b)</code>	numbers within the tolerance when either is floating point, integers exactly; lexicographic, hierarchical and pareto costs level by level; other values with ==
<code>cost::approximate_compare(a, b, tolerance)</code>	a <code>std::partial_ordering</code> : equivalent within the tolerance, otherwise by <=>; level by level, as the cost model orders, for structured costs

The checks of `easylocal::testing`, `check(app)` and the `Session` use it by default (`testing::approximately` is the same type). It is not transitive.

Costs as text

`cost::from_text<Cost>(text)` (<easylocal/cost/text.hpp>) reads a cost written by a user, such as a target: a number for an arithmetic cost, [hard, soft] for a `cost::hierarchical`, [v1, v2, ...] for a `cost::lexicographic` or a `cost::pareto`, nested as the types are ([0, [3, 1.5]]). It throws `std::invalid_argument` with the reason; `cost::to_text(cost)` writes a cost back in the same form. `cost::text_readable<Cost>` tells which costs it reads; a problem with another cost type, or its own notation, provides `read_cost(const Input&, std::string_view) -> Cost`, found by ADL through its `Input`, which the `Session` and the `TextUI` use instead.

Cost semantics

`cost::better(sm, a, b)`, `cost::equivalent(sm, a, b)` and `cost::better_or_equivalent(sm, a, b)` are the relations algorithms compare costs with, through `run.better(...)`. By default they are <, == and <= of the cost type, independent queries, so a cost type may answer <= in one pass.

The root of the expression may define them all with one hook: the function of a root `cost::apply` defines `compare(a, b)`, returning a `std::partial_ordering`. Then `better(a, b)` is `compare(a, b) < 0`, `equivalent(a, b)` is `compare(a, b) == 0` and `better_or_equivalent(a, b)` is `compare(a, b) <= 0`; an unordered result is none of them, as for two points of a Pareto front. A function that maximizes:

```
struct Profit
{
    int operator()(int value) const { return value; }
    std::partial_ordering compare(int lhs, int rhs) const { return rhs <=> lhs; }
};

solution_manager<SM>() | cost::apply(Profit{}, component<Gain>())
```

The rules, checked when the recipe is built:

- `compare` is the only hook: a function that defines `better`, `equivalent` or `better_or_equivalent` does not compile, since one relation of its own next to the defaults of the others would order the costs two ways.
- Only the root defines the semantics: a `cost::apply` with a `compare` below another node does not compile.
- A root `cost::apply` hides the structure below it: with a `cost::hard_soft` under it, a pipeline stage `until_feasible()` evaluates every component, and its hard stage compares the hard costs with their own operators.

`cost::approximately(child, {.relative, .absolute})` is the root that compares floating-point costs within a tolerance (both 1e-9 by default): costs equal within it are equivalent, and a cost is better only

by more than it, with `cost::approximate_compare` (see Comparisons within a tolerance). A descent then stops at a move that improves only by rounding errors, and a target is reached by a cost within the tolerance of it. It keeps the structure below it: over a `cost::hard_soft`, `until_feasible()` still evaluates only the hard components, and its hard stage compares within the same tolerance. The tolerance is configurable, `cost.tolerance.relative` and `cost.tolerance.absolute` in an app.

```
solution_manager<SM>()
| cost::approximately(
|   cost::hard_soft(component<Overlaps>(), component<Length>()),
|   {.relative = 1e-9, .absolute = 1e-6})
```

Without it the search compares exactly, and only the checks forgive rounding errors (see [Testing](#)).

- The algorithms that read `cost::delta` (Simulated Annealing, Great Deluge, the aspiration levels of Tabu Search) assume that a negative delta is an improvement: the sign of `cost::delta(a, b)` must agree with `better(a, b)`. A compare that finds a larger cost better breaks them; maximize with `cost::apply` returning the negated value instead. Simulated Annealing and Great Deluge reject such a compare when the compiler can evaluate it (a `constexpr` member of a default-constructible function); `check(app)` verifies the sign on the moves it tries.

Delta cost components

Member	Required
<code>delta_evaluate(const Solution&, const Move&) const -> Delta</code>	yes

- Law: `Value + Delta -> Value`, equal to the value of the component after the move. Check it with `testing::check_delta_cost_component`.
- Bound per component and per neighborhood: `delta<C, D>(args...)` for a separate delta cost component, `delta<C>()` for one co-located in the component.
- Components without a delta for a neighborhood are re-evaluated on a candidate solution; with full coverage no candidate is built.

Design choices

- **Cost comes only from components.** One evaluation path for every problem; the `SolutionManager` stays about solutions.
- **The structure of the cost is written in the recipe.** Which components are hard and which soft, their order and their weights are read where the components are listed, not deduced from a declaration order and a separate list of weights. `until_feasible()` reads the hard components from the expression.
- **Weights sit beside their term.** `component<C>() * w` cannot drift out of line with the components, as positional weights can. The `*` only marks a term of a `cost::sum`: it never multiplies component values, so domain types need no operator*.
- **cost::sum adds numbers.** Domain values are mapped explicitly with `cost::apply`, so domain types never carry arithmetic operators just to be summed.
- **Semantics belong to the cost layer.** The root of the expression may redefine ordering and equivalence (tolerances, a maximized objective) without touching the cost type.
- **One hook for the order.** `compare` defines the three relations together, so they cannot disagree, and a partial order (`unordered`) fits as Pareto dominance does.
- **Deltas never see weights or structure.** They are per component; the move cost is always recomputed by the expression, so a delta stays valid whatever the expression is.

- **delta is the contract, operator- is syntax.** Acceptance criteria rely on `cost::delta`; `cost::lexicographic` has none on purpose.

NeighborhoodExplorer

<easylocal/helpers/neighborhood_explorer.hpp>, <easylocal/helpers/neighborhood_union.hpp>

The NeighborhoodExplorer defines *move semantics*: which moves exist, whether a move is valid and how it changes a solution.

Contract

Member	Required	Used by
input_type, solution_type, move_type	yes (input_type and solution_type those of the SolutionManager)	everything
input() const -> const Input&	yes	the check that the explorer was built for the runner's Input, unions
is_valid(const Solution&, const Move&) const -> bool	yes	debug assertions, checks
make_move(Solution&, const Move&) const	yes; the Solution by reference, which it changes	every runner
input() const -> const Input&	no (neighborhood_explorer_base gives it)	the checks that the services share the Input
moves(const Solution&) const -> input range of moves	one of the two, for deterministic algorithms	First and Best Improvement, Tabu Search, check(app), Session and TextUI
first_move(const Solution&, Move&), next_move(const Solution&, Move&) -> bool		
random_move(const Solution&, RNG&) const -> std::optional<Move>	for stochastic algorithms	Simulated Annealing, Hill Climbing, Late Acceptance Hill Climbing, Great Deluge, Pareto Late Acceptance Hill Climbing, check(app), Session and TextUI
name() -> std::string_view	no	TextUI display
inverse(const Solution&, const Move& move, const Move& tabu_move) const -> bool	for tabu search	Tabu Search: whether move is forbidden by tabu_move, applied earlier
tabu_attribute(const Move&) const -> value with std::hash and ==	no (default: the move, if hashable)	frequency-based tabu memory

neighborhood_explorer_for<NHE, SM> checks the first four rows, so an explorer without them is rejected where it is composed, not deep in a debug assertion or a union. neighborhood_explorer_base<SM, Move> provides the types and input(), from the SolutionManager it is constructed with.

Concepts: neighborhood_explorer_for<NHE, SM>, cursor_neighborhood_for, native_moves_neighborhood_for, deterministic_neighborhood_for, random_neighborhood_for,

inverse_neighborhood_for, has_tabu_attribute. Customization points: easylocal::moves(nhe, solution), easylocal::random_move(nhe, solution, rng), easylocal::inverse(nhe, solution, move, tabu_move) and easylocal::tabu_attribute(nhe, move).

- make_move takes the Solution by non-const reference: a runner on an explorer whose make_move takes it by value or by const reference, which would change a copy and leave the search where it is, is rejected with a message when the neighborhood is added (runner | neighborhood<NHE>()), and check_neighborhood reports its null moves.
- moves may return any input range whose elements convert to move_type.
- When both moves and the cursor exist, the cursor is used.
- random_move may return std::optional<T> for any T convertible to move_type; no distribution is required.
- inverse has no default: which moves a tabu move forbids (the same pair of jobs, or any move of either job) is a modelling choice of the neighborhood, and can be one of its parameters.
- The recipe checks the contract member by member when it is written (when the explorer declares its solution_type, else when it is bound), so a missing move_type or is_valid, or a make_move that is not const, is named in the error.
- The explorer is constructed from the SolutionManager: NHE(const SM&, args...), args from neighborhood<NHE>(args...). The construction uses parentheses, so the arguments convert as the constructor takes them (neighborhood<NHE>(2) for a double parameter).

Parameters

An explorer may have parameters, such as the inverse definition of a tabu search, or a neighborhood size. It declares them as a parameter block (using parameters_type = P;, see [Configuration](#)) and is constructed from them after the SolutionManager:

```
struct SwapParameters { std::string inverse{"both_jobs"}; /* schema, validate */ };

class SwapExplorer
{
public:
    using parameters_type = SwapParameters;
    SwapExplorer(const SM&, const SwapParameters&, args...);
    // ...
};

neighborhood<SwapExplorer>() // default parameters
neighborhood<SwapExplorer>(SwapParameters{...}, args...) // given ones
```

The recipe holds the parameters (parameters(), configuration()), keeps them when deltas are attached, and gives them to the explorer each time a runner or an app is bound. A runner and an app expose them under neighborhood.* (a runner's own neighborhood in an app under runners.<name>.neighborhood.*); in a union, each child's under its position (neighborhood.0.*), next to neighborhood.random_biases. This is the rule of every configurable class: its parameters_type is a parameter block, and it is constructed from it (see [Configuration](#)).

Convenience base

```
template<class SolutionManager, class Move>
class neighborhood_explorer_base; // aliases, input(), protected solution_manager_
```

It keeps the `SolutionManager` by reference, so the constructor from a temporary `SolutionManager` is deleted: the `SolutionManager` outlives the explorer. Its `input()`, the `SolutionManager`'s Input, comes from `input_base`.

Recipes

```
neighborhood<NHE>(args...)
| delta<C, D>(args...)    // separate delta cost component for component C
| delta<C>()              // co-located: C::delta_evaluate
// equivalently: neighborhood<NHE>(args...).with_delta<C, D>(args...).with_delta<C>()
```

- A delta cost component, `D` or the co-located `C`, has `delta_evaluate(const Solution&, const Move&) const`, which returns the change of the value of `C`: the value plus the change is a value of `C`'s type. When the explorer declares its `solution_type` and `move_type`, as `neighborhood_explorer_base` does, `delta<...>()` checks it as it is written, with a message that names it; otherwise binding the recipe does.
- An explorer that takes a delta cannot be final: the delta layer derives from it, and `delta<...>()` says so.
- A recipe attaches at most one delta to each component.

Neighborhood unions

```
neighborhood_union(child_recipe_1, child_recipe_2, ...)
| random_biases(b1, b2, ...)
```

- The move type is a variant of the children's moves. A component is evaluated by the delta of the child the move comes from, each child using its own; the moves of a child that has no delta for it are evaluated in full, on a copy of the solution with the move made, and the moves of the other children keep their deltas. A copy is made only for a move that needs one, so a union where every child has a delta for every component never makes one.
- The bindings of those deltas are built once, when the union is constructed, and the union returns them by reference.
- Enumeration visits the children in order.
- Sampling draws a child with probability proportional to its bias, excluding zero-bias children and children that produced no move, until a move is found or no child is left. Biases are configurable (`NeighborhoodUnionParameters`).
- Unions nest; traces record each move's route through the nesting and the selection statistics.
- A neighborhood type may appear more than once in a union, with different parameters: `child<I>()` names a child by its position, `child<NHE>()` by its type when it occurs once.
- `inverse`: moves of different children never forbid each other; between moves of the same child, the child decides. The union has it when every child has it.
- `tabu_attribute`: the child's attribute, tagged with the child (a `std::variant` by index), so equal attributes of two children stay distinct.

Design choices

- **Two enumeration protocols.** The range protocol is idiomatic C++; the cursor keeps EasyLocal 3 explorers and lazy enumeration natural. Algorithms see a single customization point.
- **Validity is relative to a valid solution.** `is_valid(solution, move)` never re-checks the solution; the framework asserts both before and after every application in debug builds.
- **Sampling is not uniform by contract.** Explorers choose their own distribution; unions only control the choice of the child.

Runners

<easylocal/runners.hpp>; runners/runner.hpp, runners/search_run.hpp, runners/run_control.hpp, one header per algorithm

A **runner** couples a search algorithm with the recipes of the services. The algorithm describes the search; the framework owns its execution.

Composing a runner

```
auto runner = make_runner<Algorithm>({...})           // the algorithm's parameters
              | sm_recipe | nhe_recipe;              //
or .with_solution_manager(sm).with_neighborhood(nhe)

auto search = runner.bind(input);                    // materializes the services
auto result = search.run(solution, algorithm_args..., with(control, tracer));
```

When the algorithm's `parameters_type` is a parameter block (a schema and `validate()`), the runner holds the parameters and builds the algorithm when it is bound: `make_runner<Algorithm>(parameters)` creates it, and `Runner{Algorithm{...}}` is rejected at compile time. Any other algorithm is held as an object: `make_runner<Algorithm>(args...)` or `Runner{Algorithm{...}}`.

Invalid parameters throw `std::invalid_argument`, in Release builds as in Debug ones, wherever they enter: the constructors of the built-in algorithms and of their policies (temperature schedules, tabu lists), `make_runner`, and `bind` after a change through `parameters()`. The message names the field, `temperature.cooling_rate: expected a value in (0, 1), got 2`, or gives the requirement that does not hold. A real parameter with no upper bound (a temperature, a water level, a factor of the best cost) excludes infinity, `(0, unlimited)`, so its domain rejects it as any other value out of range. A custom algorithm gets the same check from `config::require_valid(parameters)`. Inside a run, the remaining contracts are asserts.

Member	Purpose
<code>parameters()</code>	the algorithm's parameters, to read or change (parameterized algorithms)
<code>bind(const Input&)</code>	build the services and the algorithm for an Input, a BoundRunner (temporaries are rejected)
<code>configuration()</code>	the parameters of the algorithm (search), the cost (cost), the SolutionManager (solution_manager) and the neighborhood (neighborhood), as a <code>config::parameter_set</code>
<code>with_hard_cost()</code>	the same runner on the hard branch of a hierarchical cost
BoundRunner: <code>run(solution, args..., [with(...)])</code>	run the algorithm
BoundRunner: <code>initial_solution(), random_solution(rng), input(), better(a, b)</code>	helpers

Built-in algorithms

Algorithm	Needs	Parameters	iterations counts
runners::FirstImprovement	moves or cursor, better	max_evaluations (unlimited: until a local optimum)	committed moves
runners::BestImprovement	moves or cursor, better	max_evaluations (unlimited: until a local optimum)	committed moves
runners::HillClimbing	random_move, better, better_or_equivalent	max_idle_iterations, max_evaluations	proposed moves
runners::LateAcceptanceHillClimbing	random_move, better, better_or_equivalent	history_length, max_idle_iterations, max_evaluations	proposed moves
runners::ParetoLateAcceptanceHillClimbing	random_move, better, a cost::pareto cost, random_solution	history_length, min_iterations, idle_ratio, second_chance, max_evaluations	proposed moves
runners::GreatDeluge	random_move, better, an arithmetic cost	initial_level, min_level, level_rate, neighbors_sampled, max_evaluations	proposed moves
runners::TabuSearch<List, Aspiration>	moves or cursor, better, inverse	max_idle_iterations, max_iterations, max_evaluations, tabu_list: the list's	committed moves
runners::FirstImprovementTabuSearch<List, Aspiration>	as TabuSearch	as TabuSearch, plus candidates: improve_on_best	committed moves
runners::AspirationPlusTabuSearch<List, Aspiration>	as TabuSearch, an arithmetic cost	as TabuSearch, plus candidates: min_moves, max_moves, plus, aspiration_level	committed moves
runners::EliteCandidateTabuSearch<List, Aspiration>	as TabuSearch, an arithmetic cost	as TabuSearch, plus candidates: elite_size, quality	committed moves
runners::SimulatedAnnealing<Temperature, Acceptance>	random_move, better, an acceptance-compatible cost	a temperature policy (temperature), an acceptance policy, max_evaluations	proposed moves

The parameters follow one naming rule. A max_<count> of the run (max_evaluations, max_iterations, max_idle_iterations, FixedTemperature's max_accepted) is a cap that ends the run when it is reached, of type easylocal::limit: a number, or easylocal::unlimited (a tag of type unlimited_t, which converts to the unlimited limit), written unlimited in a configuration file, on the command line and in the TextUI; 0 is a limit of zero, not "no limit". The size of a schedule, which shapes the search rather than stopping it, is

allowed_<x> (allowed_iterations of the annealing schedules, allowed_reheats, allowed_running_time), and a threshold the search must pass first is min_<x> (min_iterations of Pareto Late Acceptance). The bounds of a value, such as min_tenure and max_tenure, keep their names. The caps are unlimited by default, except the ones whose value belongs to the problem, which have no default and must be set: max_idle_iterations, allowed_iterations, allowed_running_time and min_iterations.

The budget is checked only before evaluating a move, so an empty neighborhood is a local optimum even when the budget is exhausted.

Best Improvement applies the first of the best moves in the order of moves(): its ties are broken deterministically, where EasyLocal 3's SteepestDescent drew among them at random. An explorer that wants random ties shuffles its moves.

Hill Climbing accepts a random move when its cost is better or equivalent to the current one, so it moves across plateaus where a descent stops. It ends with idle_limit_reached after max_idle_iterations consecutive proposals without a strict improvement, or with local_optimum when the neighborhood proposes no move. Its cost never worsens, so the final solution is the best. Every runner that draws random moves (Late Acceptance, Great Deluge, Simulated Annealing, Pareto Late Acceptance) also ends with local_optimum when the neighborhood proposes none.

Late Acceptance Hill Climbing (Burke and Bykov) also accepts a move whose cost is better than or equivalent to the cost the current solution had history_length proposals earlier; the history starts filled with the initial cost, and with history_length 1 it accepts the moves Hill Climbing accepts. Its idle count runs from the last improvement of the best cost, and it returns the best solution found. EasyLocal 3 recorded the best cost in the history; EasyLocal 4 records the current cost, as in the original algorithm.

Pareto Late Acceptance Hill Climbing (Da Ros and Di Gaspero) is the multi-objective variant, for a cost::pareto cost: its history holds solutions, the initial one and history_length - 1 drawn with the solution manager's random_solution, visited in a circle. A random move of the current solution replaces it in the history when the candidate dominates it, and the search moves on to the next solution of the history. Otherwise, with second_chance, a candidate that dominates the next solution replaces it, and the search goes on from the solution it replaced, two positions on; else it moves on to the next solution. Past min_iterations the search ends with idle_limit_reached as soon as more than idle_ratio of the iterations went without a replacement. Its result is the run's front, with the first solution by objectives as solution.

Tabu Search applies at each iteration the best admissible move of the whole neighborhood, even a worsening one, ties broken uniformly at random. A move is admissible unless the tabu list forbids it, through the neighborhood's inverse (see [NeighborhoodExplorer](#)), and the aspiration criterion does not lift the prohibition: aspiration::ByObjective (the default) admits a tabu move that would improve the best cost, aspiration::None never does and then skips evaluating tabu moves. Another criterion is a value with overrides(run, candidate, best) -> bool and static constexpr bool needs_cost, which aspiration_for<A, Run> checks. When every move is tabu, the least tabu one (the shortest remaining tenure) is applied. The search ends with idle_limit_reached after max_idle_iterations iterations without improving the best cost, and returns the best solution found. FirstImprovementTabuSearch stops the scan at the first admissible move that improves the current cost (with improve_on_best, the best cost); without one it applies the best admissible move. Both take the RNG as a run argument, for the ties.

The four tabu searches share one parameter block, TabuSearchParameters<ListParameters, CandidateParameters>: the limits of the run, the list's parameters as the group tabu_list, and those of the candidate strategy, the scan of each iteration, as the group candidates (search.candidates.* in a configuration): candidates::FirstImprovementParameters, AspirationPlusParameters and EliteListParameters. The full scan of TabuSearch, candidates::FullParameters, has none, and no group:

```
make_runner<AspirationPlusTabuSearch<tabu::RandomTenure>>(  
  {.max_idle_iterations = 500,  
   .tabu_list = {.min_tenure = 5, .max_tenure = 15},  
   .candidates = {.min_moves = 10, .max_moves = 40}})
```

Two runners implement Glover's candidate list strategies, whose levels are values of the cost (an arithmetic cost):

- `AspirationPlusTabuSearch` examines admissible moves until plus more after the first one under the aspiration level, but at least `min_moves` and at most `max_moves`, and applies the best examined.
- `EliteCandidateTabuSearch` keeps, from a full scan, the `elite_size` best admissible moves besides the one applied; the next iterations evaluate only the kept moves still valid and apply the best admissible one while its cost is at most the quality level, otherwise a full scan builds a new list.

Both levels are a factor f (`aspiration_level`, quality) of the best cost b , kept on the worse side of it whatever its sign: $b + (f - 1) * |b|$, which is $f * b$ for a positive best cost.

Tabu lists in runners: `tabu`; their parameters are the group `tabu_list`:

List	Parameters	A move stays tabu
FixedLength (TS1)	tenure	for tenure iterations after the move it would undo
RandomTenure (TS2)	min_tenure, max_tenure	for a tenure drawn uniformly in [min_tenure, max_tenure]
Cyclic (TS3)	period, tenures	for the current tenure, which takes the tenures in turn every period iterations
Reactive (TS4)	increase, decrease, repetitions, chaos, cycle_length, max_tenure, verify_equality	for a tenure that reacts to revisited solutions; needs the solution hash and random_move
Frequency (TS5)	threshold	while its attribute was applied in more than threshold of the iterations; needs tabu_attribute, not inverse
ObjectiveBased	tenure	while its cost is equivalent to one reached in the last tenure iterations, by the cost semantics (equivalent); needs neither inverse nor attribute
LimDynamic (limited dynamic)	min_tenure, max_tenure, idle_threshold	for a tenure that grows by one after idle_threshold idle iterations and falls back to min_tenure on an improvement or at max_tenure
Foo (FOO, Fluctuation Of the Objective)	window, increment, fluctuation	for a tenure that grows by increment when the costs of the last window iterations spread less than fluctuation, and shrinks by one otherwise; needs cost::delta, whose spread across two hard levels of a cost::hierarchical is infinite, so a window that changes the hard cost shrinks the tenure
RandomFoo (randomized FOO)	ranges of the three	as Foo, drawing them again at each window

Most lists forbid moves through the neighborhood's inverse, Frequency through its tabu_attribute, ObjectiveBased through the candidate's cost, evaluating each move before the tabu check (see [NeighborhoodExplorer](#)); each requires only what it uses. Reactive (Battiti and Tecchiolli) starts with tenure 1 and recognizes solutions by solution_hash (see [SolutionManager](#)); a hash collision counts as a revisit, unless verify_equality keeps a copy of each visited solution and compares those with the same hash, which needs solution equality (has_solution_equality, otherwise the run throws std::invalid_argument). A solution revisited within cycle_length iterations multiplies the tenure by

increase (up to `max_tenure`) and updates the average cycle length; without such cycles for longer than the average, the tenure is multiplied by decrease. A solution visited more than repetitions times counts as chaos; after more than chaos counts the memory is reset and the search escapes with $1 + (1 + r) * \text{average} / 2$ random moves (r uniform in $[0, 1)$), applied whatever their cost, counted as iterations and recorded in the list like the others.

The names of the lists are those of Da Ros, Di Gaspero and Schaerf's performance analysis of tabu list strategies: `LimDynamic` is a limited dynamic tenure, `Foo` the Fluctuation Of the Objective scheme of Blöchliger and Zufferey, and `RandomFoo` its variant that draws the parameters again at each window.

The lists with one tenure for all moves (`Reactive`, `LimDynamic`, `Foo`, `RandomFoo`) apply the current tenure to every move they hold: a move is tabu while it is younger than the tenure, so a growth also makes tabu again the older moves it covers. Each list keeps the moves that the next growth can reach: those of the last tenure + increment - 1 iterations before a `Foo` window ends, of the tenure that increase would bring for `Reactive`.

A tabu list is a value with its parameters that makes, for each run, a state with `make_state<Run>()`. `tabu_tenure(candidate)` gives the iterations left before a candidate move is admissible, nothing when it is (`tabu_candidate: move()`, `forbidden_by(tabu_move)` through the inverse, `attribute()`, and, when the state declares `static constexpr bool needs_cost = true`, `cost()` and `equivalent_cost(other)`, members of `tabu_candidate<Run, true>` only, so a list that reads the cost without declaring it does not compile); `update(step, rng)` records an applied move (`tabu_step: move()`, `solution()`, `cost()`, `iteration()`, `improved_best()`, `attribute()`, `solution_hash()`, the last two when the problem has them). A state with `escape_moves()` asks for that many random moves, and one with `current_tenure()`, a tenure for all its moves, has it traced as `tabu_tenure_changed` at the start of the run and whenever it changes. `tabu_list_for` checks a custom list.

Great Deluge accepts a move that improves the current cost or whose cost does not exceed the water level. The level starts at `initial_level` times the initial cost and is multiplied by `level_rate` every `neighbors_sampled` proposals; the search ends, completed, when the level falls below `min_level` times the best cost, and returns the best solution found. The level is a value of the cost, so Great Deluge requires an arithmetic cost (`cost::arithmetic`), and positive, since the levels are factors of it: a best cost of zero ends the search.

Simulated Annealing returns the best solution found. Temperature policies in `runners::temperature`, all configurable:

Policy	Parameters	Stops when
Classic	initial/final temperature, cooling rate, samples per temperature	the final temperature is reached
FixedLength	initial/final temperature, cooling rate, allowed_iterations	the allowed iterations are spent, spread over the levels
Cutoff	as FixedLength, plus accepted_ratio	the allowed iterations are spent; cools only after accepted_ratio of a level's share of them have been accepted, never on samples
Hybrid	as Cutoff	the allowed iterations are spent; cools on samples or acceptances (samples_per_temperature(), accepted_limit())
FixedTemperature	temperature, allowed_iterations, max_accepted (unlimited by default)	the allowed iterations are spent, or max_accepted moves are accepted; never cools
TimeBased	initial/final temperature, cooling rate, running time, accepted per temperature (unlimited: cools only on time)	the running time is over or the final temperature is reached; levels share the time
Reheating<Descent>	descent: the schedule's, plus allowed_reheats, reheat ratio, first-descent share	the last descent ends; each reheat restarts from reheat_ratio times T0

The levels of a schedule are counted once, from its temperatures: the smallest k with $\text{initial_temperature} * \text{cooling_rate}^k \leq \text{final_temperature}$, a count within rounding of an integer being that integer, so that 1 to 0.001 by 0.1 is three levels. Classic and TimeBased end after that many levels, and the schedules with a budget spread it over them.

TimeBased reads the clock (`std::chrono::steady_clock`; `BasicTimeBased<Clock>` takes another one) as a run checks its time limit: at an interval of proposals that adapts so that readings come about a millisecond apart, and at each early cooling. Its trajectory depends on the speed of the machine, so equal seeds do not give equal runs.

Reheating<Descent> reheats any schedule whose parameters have an `initial_temperature` (all but `FixedTemperature`): a first descent, then up to `allowed_reheats` descents that restart from `reheat_ratio` times the initial temperature. When the schedule has a budget, `allowed_iterations` or `allowed_running_time`, the first descent spends `first_descent_share` of it and the reheats divide the rest evenly, at least one iteration each, so `allowed_reheats` may not exceed the iterations the first descent leaves (a requirement of the block); Classic, which has none, runs whole at every descent. The schedule's parameters are the group `descent: { .temperature = { .descent = { ... }, .allowed_reheats = 2 } }`, `search.temperature.descent.*` in a configuration. It calibrates when the schedule does, keeping the reheat temperature above the final one. `Reheating<Hybrid>` is `EasyLocal 3`'s annealing with reheating.

For a complete extension example, see [Customizing Simulated Annealing](#). It walks through the library's run implementation, configures this wrapper and writes a temperature policy that reheats after consecutive rejections.

Every built-in policy can estimate its initial temperature (the constant one for `FixedTemperature`) with `calibration_samples > 0`: before the run, Simulated Annealing evaluates that many random moves at the initial solution, without applying them, and the policy starts from the temperature at which a worsening move of average size is accepted with probability `initial_acceptance` (0.5 by default; 0.2 suits a good initial solution), as in Johnson et al. (1989). Improving moves and infinite deltas (a hierarchical hard level) are ignored; without worsening moves, or when the deltas ask for a temperature that is not a representable number, `initial_temperature` stays, and the estimate is kept above the final temperature. The sampled moves count as evaluations, not as iterations. A custom policy opts in by modelling `calibrating_temperature_policy: calibration_samples()` and `calibrate(std::span<const double> deltas)`.

`runners::MetropolisAcceptance` (the default) requires `cost::delta` (see [Cost](#)). On a `cost::hard_soft` cost the delta of a move that worsens the hard cost is infinite, so Simulated Annealing never accepts it, at any temperature, and cannot cross an infeasible region as `EasyLocal 3's HARD_WEIGHT` let it do. To cross one, write the cost as a single weighted sum, `cost::sum(component<H>() * 1000, component<S>())`, and check that the result is feasible. Another acceptance criterion is a value with `accept(candidate, current, temperature, rng) -> bool`, which `acceptance_policy_for<Acceptance, Cost, RNG>` checks. The parameters of `SimulatedAnnealing<Policy>` are `SimulatedAnnealingParameters<Policy::parameters_type>`, the policy's under the group temperature and the evaluation budget `max_evaluations: make_runner<SimulatedAnnealing<Classic>>({.temperature = {...}}), search.temperature.*` in a configuration. The temperatures have no default, since they live in the scale of the cost, and neither has the budget of a schedule that has one; the rest of each block does. A calibrating schedule with `calibration_samples` above 0 needs a cost with `cost::delta`; with any other cost the run throws `std::invalid_argument`.

Results

Built-in algorithms return `search_result<Solution, Cost>`: solution, cost, evaluations, iterations, termination (`termination_reason::completed`, `local_optimum`, `evaluation_budget_exhausted`, `cancelled`, `target_reached`, `idle_limit_reached`, `time_limit_reached`). With a `cost::pareto` cost they return `pareto_search_result<Solution, Cost>`, a `search_result` that adds `front`: the non-dominated solutions the run reached, as `pareto_point{solution, cost}`, ordered by their objectives. The `search_run` keeps them in a `pareto_archive` as the run starts, evaluates solutions and commits moves, so every algorithm has a front without doing anything. The archive compares costs with the run's relations (better, equivalent, so a root compare of the cost expression applies): a solution enters unless an archived one is better or has an equivalent cost, and removes those it is better than. So the front holds one point per non-dominated cost, the first reached.

The run options choose otherwise, for a run or a solve: `options.keep_front({.keep_equivalent = true, .max_front_size = 100})` (a `pareto_archive_parameters`, the field `front` of `run_options`) keeps every distinct solution of an equivalent cost (by the problem's solution equality), and bounds the front: once it holds `max_front_size` points (at least 1, unlimited by default), a solution enters only by removing points it dominates. On a plateau, keeping every solution makes the front grow with the plateau and each offer scan it, so bound it. A reached target is the termination reason also when it coincides with a local optimum or the end of the algorithm; a cancellation takes precedence. Solvers and tools require only `search_result_for<Result, Solution, Cost>`: solution and cost.

Writing an algorithm

An algorithm is a class with a `run` member taking the `search_run` first:

```

class MySearch
{
public:
    using parameters_type = MyParameters;    // with a schema: configurable
    explicit MySearch(const MyParameters&);

    template<class Run>
    auto run(Run& run, typename Run::solution_type solution /*, extra args */) const;
};

```

With a schema and `validate()` on `MyParameters`, the runner holds them and builds `MySearch` at bind; they are configurable as `search.*` with no other member. Without one, `parameters_type` serves only app registration.

Extra run arguments (an RNG, for example) are passed through `search.run(solution, extra...)`.

An algorithm may also declare what it asks of the services it searches with, so that a registration on a neighborhood it cannot use names the capability that is missing instead of the concepts that failed:

```

using search_requirements = easylocal::runners::requirements<
    easylocal::runners::needs::random_moves,
    easylocal::runners::needs::non_worsening>;

```

Capability	What the services must provide
<code>needs::enumerated_moves</code>	the explorer's <code>moves()</code> , or <code>first_move()</code> and <code>next_move()</code>
<code>needs::random_moves</code>	the explorer's <code>random_move(solution, rng)</code>
<code>needs::random_solutions</code>	the <code>SolutionManager</code> 's <code>random_solution(rng)</code>
<code>needs::strict_improvement</code>	a cost with <code>better()</code>
<code>needs::non_worsening</code>	a cost with <code>better_or_equivalent()</code>

The capabilities are reported in the order they are declared, one message for the first the services do not provide; the neighborhood's before the cost's reads best. They only choose the message: an algorithm that runs is never rejected because of what it declares, and one that declares nothing keeps the generic message. The built-in algorithms declare theirs.

search_run

Member	Effect
<code>limit_evaluations(n)</code>	evaluation budget, including the initial evaluation: a count, or <code>easylocal::unlimited</code>
<code>start(solution) -> evaluation</code>	first evaluation, <code>run_started</code> , progress
<code>evaluate_solution(solution) -> evaluation</code>	another solution (a population, a history): counts, traced as visited, offered to the archive
<code>front()</code>	with a <code>cost::pareto</code> cost, the archive of the non-dominated solutions reached
<code>should_stop() -> bool</code>	cancellation, a reached target cost, an exhausted evaluation budget or a passed time limit; the reason is recorded
<code>moves(solution), random_move(solution, rng)</code>	neighborhood access; unions emit selection events
<code>evaluate_move(solution, current, move) -> candidate</code>	counts, <code>move_evaluated</code> , progress
<code>commit(solution, current, candidate, move)</code>	applies, <code>move_accepted</code>
<code>next_iteration()</code>	advances the iteration counter
<code>incumbent_updated(previous, cost)</code>	<code>incumbent_updated</code> event; the cost checked against the target
<code>commit_improvement(solution, current, candidate, move)</code>	<code>commit</code> , then <code>incumbent_updated</code> , for an algorithm whose current solution is its best (Hill Climbing, First and Best Improvement)
<code>finish(solution, cost[, reason]) -> search_result</code>	<code>local_optimum</code> (if that is the reason), <code>run_finished</code>
<code>better, equivalent, better_or_equivalent</code>	cost semantics
<code>evaluations(), iterations(), target()</code>	counters, the caller's target cost
<code>context(), neighborhood_explorer(), input(), solution_manager()</code>	the search context
<code>evaluation(), emit(event), tracer(), control()</code>	escape hatches (bypass counters and events)
<code>with_evaluation(wrap)</code>	a run that evaluates through <code>wrap(evaluation())</code> , a decorated facility (a delay, a cache), and shares the context, the cost relations, control, tracer, deadline, target and the budget left

`evaluation` / `candidate` expose `cost()`; finish without a reason uses the one recorded by `should_stop()`, or completed.

A decorated facility, the one `wrap` returns to `with_evaluation`, has the interface of the facility it wraps: the types `evaluation_type` and `candidate_type`, `evaluate(solution) -> evaluation_type`, `evaluate_move(solution, current, move) -> candidate_type` and `commit(solution, current, candidate)`, the costs of an evaluation and of a candidate being `cost()` of the run's cost type. It forwards

them, adding its own work (a delay, a count, a cache); the run keeps the counters, the events and the limits around it.

An algorithm that returns the best solution it visited, rather than the last, keeps it in a `best_so_far{solution, cost}`: `best.update(run, solution, current)` copies the solution when its evaluation is better than the best, emits `incumbent_updated`, and returns whether it did; at the end `run.finish(std::move(best.solution), std::move(best.cost))`. Late Acceptance, Great Deluge, Simulated Annealing and the tabu searches use it. An algorithm whose current solution is its best one applies an improving move with `run.commit_improvement(...)`, which emits `incumbent_updated` too: so every built-in runner reports each new best cost.

Run options

Spelling	Effect
<code>with(control)</code>	cancellation (<code>std::stop_token</code>) and progress observer
<code>with(tracer)</code>	semantic trace events
<code>with(control, tracer)</code>	both
<code>stop_at(target), with(...).stop_at(target)</code>	stop as soon as the best cost is at least as good as target
<code>options.without_target()</code>	the same options without their target (a pipeline gives them to a stage that is not the last)
<code>max_evaluations(n), with(...).max_evaluations(n)</code>	stop once the run has made <code>n</code> evaluations, the initial one included; a runner's own <code>max_evaluations</code> , if smaller, still applies; termination <code>evaluation_budget_exhausted</code>
<code>timeout(5s), timeout(2.5), with(...).timeout(...)</code>	stop once the time limit has passed since the run started: a <code>std::chrono</code> duration or a number of seconds; termination <code>time_limit_reached</code>

`run_control{stop_token, observer}` calls `observer(const run_progress&)` with evaluations, iterations and `evaluation_limit`. A frontend that shows the progress from another thread stores it in a `shared_run_progress` from the observer and loads a copy when it draws, as the TextUI and the REST server do. `control.stop_token()` gives the token back, for a control of its own that shares it, and `control.observing_progress(observer)` a copy with another observer of the progress.

`control.observe_best_cost<Cost>(best)` adds an observer of the best costs: the run calls `best(const Cost&)` with its first cost, then with each cost better, by its cost semantics, than those it reported, the last one being the best the run found. It is called on a commit that improves, not on each evaluation; costs of another type than `Cost` are not reported. A `shared_best_cost<Cost>` keeps the last one for another thread, behind a mutex: the TextUI's progress line (`best=`) and REST's `progress.best` show it.

The options combine in any order: `with(control).timeout(30s).max_evaluations(100000).stop_at(0)`. They are the fields `control`, `tracer`, `target`, `timeout` (none: no time limit) and `max_evaluations` (an `easylocal::limit`, unlimited by default) of `run_options`. A negative time limit, a NaN one (a number of seconds or a floating-point duration) and an infinite number of seconds throw `std::invalid_argument`; a duration beyond what the clock counts is the longest it can. `search_run` checks the deadline with the other stopping conditions, reading the clock at an interval of checks that adapts so that readings come about a millisecond apart: a loop that checks at every move reads it rarely, one that checks once per long iteration reads it every time. There is no timer thread.

The target converts to the runner's cost type and is compared with its cost semantics (better_or_equivalent). `search_run` checks the costs reached in `start`, `commit` and `incumbent_updated`, so every algorithm that checks `should_stop()` honours a target without further code. A run that ends with `target_reached` returns a solution that meets the target: with a partial order, such as a `cost::pareto` cost, the reaching cost need not dominate the best one, so `best_so_far` keeps the first cost that meets the target, and a run with a front returns a point of the front that meets it. A floating-point cost updated by deltas drifts from its full evaluation by rounding errors: when a committed cost misses the target only within `cost::tolerance{}`, the run evaluates the solution in full (not counted in the budget) before deciding, so a zero target, such as the one of `until_feasible()`, is reached by a cost that the deltas left at $1e-17$. The same options are accepted by every solver's `solve(input, options)`.

Design choices

- **One run member.** Algorithms describe their logic; counters, budget, cancellation, progress and events belong to `search_run`, so they behave the same in every algorithm. What the algorithm alone knows stays its own: counting an iteration, checking `should_stop()` and reporting a new incumbent (`commit_improvement()`, `best_so_far`).
- **Every runner is cancellable.** A run always carries a `run_control`; checking `should_stop()` is the whole contract. The cost of an inactive control is a null check.
- **Tracing is compile-time.** Without a tracer, events are not even constructed, nor the routes of the moves of a neighborhood union.
- **Per-run state is local.** `BoundRunner::run` is `const`, and each run starts from a new algorithm, built from the parameters or copied: a custom algorithm may keep state during a run, never from one run to the next.
- **The algorithm class is its own key.** No tag types: the class names the algorithm in `make_runner` and in app registrations.

Solvers

<easylocal/solvers.hpp>; solvers/local_search.hpp, solvers/multi_start.hpp, solvers/pipeline.hpp, solvers/two_stage.hpp, solvers/initialization.hpp

A solver goes from an Input to a final solution. It owns the RNG, builds the initial solutions and orchestrates its runners.

Construction

```
auto solver = make_solver<solvers::MultiStart>(runner,
solvers::MultiStartParameters{.starts = 5})
    .initialization(initialization::random)
    .seed(7);
auto result = solver.solve(input);
auto controlled = solver.solve(input, easylocal::with(control,
tracer).stop_at(target));
```

Every solver is built from its runner (and, for MultiStart, its parameters) and configured with the same two builders, which return the solver: itself on an lvalue, by value on a temporary.

Builder	Default	Effect
.seed(n)	0	seeds the solver's RNG
.initialization(tag)	initialization::automatic	how the initial solutions are built (Initialization)

The optional trailing [run options](#) — cancellation, tracer, target cost — reach every run of the solver. Before each run, the solver sends the tracer a run_context event with the pipeline stage and the attempt (or the start), which tells the runs of a solve apart in a trace ([Tracing](#)).

make_solver takes the solver class template as its key and deduces the runner type. A custom RNG type is the last constructor argument, from which the solver deduces it: solvers::LocalSearch{runner, RNG{seed}}, solvers::MultiStart{runner, parameters, RNG{seed}}, and for a pipeline solvers::pipeline<RNG>(stages...).

Built-in solvers

Solver	Built with	Behaviour
solvers::LocalSearch	LocalSearch{runner}	one initial solution, one run
solvers::MultiStart	MultiStart{runner, {.starts = n}} (MultiStartParameters)	up to starts runs from fresh solutions, keeps the best by cost semantics
solvers::Pipeline	stages with \ or pipeline(...) (below)	runners in sequence, each stage from the solution of the previous one

Results carry the effort of the whole solve: evaluations and iterations add up over MultiStart's starts and a pipeline's stages and attempts. With a cost::pareto cost, the front of MultiStart's result merges the fronts of all its starts, and so does the front of a stage over its attempts.

MultiStart's starts and a pipeline stage's attempts run in the same loop. It ends early when a run is cancelled, reaches the target or its time limit, or when the solve's time or evaluations are spent; the

termination is then cancelled, `target_reached`, `time_limit_reached` or `evaluation_budget_exhausted`, otherwise the last run's.

A solve's time limit and evaluation budget (`solve(input, easylocal::timeout(30s))`, `easylocal::max_evaluations(1'000'000)`, or both, `with(control).timeout(30s).max_evaluations(1'000'000)`) bound all its runs together: `LocalSearch` gives them to its one run, `MultiStart` starts no run once they are spent and gives each start what is left, and a pipeline gives each stage what is left.

The observer of the caller's `run_control` sees the progress of the whole solve: `MultiStart` and a pipeline give each run the caller's stop token with an observer that adds the evaluations and iterations of the runs before it, and the evaluation limit it reports is the solve's, so a progress bar never goes back from one start, attempt or stage to the next. Its observer of the best costs gets only the costs better than those of the runs before, by the cost semantics of the runner of each run: a new start does not report a worse first cost. A stage on another cost than the observed one reports none.

Pipeline

A pipeline runs its stages in order over the same Input and Solution: the first from an initial solution, each other from the solution of the previous one. A stage is any runner, with its own recipes (cost, neighborhood), named:

```
using namespace easylocal::solvers;
auto solver = (stage("feasible", descent) & until_feasible() & attempts(10))
  | stage("descent", descent)
  | stage("anneal", annealing);
auto result = solver.seed(7).solve(input);
```

& gives a stage its options and | chains the stages. & binds tighter than |, but GCC warns about a & b | c (-Wparentheses): put a stage and its options in parentheses. The same pipeline spelled out, with methods:

```
auto solver = pipeline(stage("feasible", descent).until_feasible().with_attempts(10))
  .then(stage("descent", descent))
  .then(stage("anneal", annealing));
```

`pipeline(a, b, c)` is `a | b | c`, and `pipeline(a)` a pipeline of one stage. The stages must have the same Input and Solution (checked at compile time); their costs may differ.

Stage option	Method	Effect
<code>& target(cost)</code>	<code>.with_target(cost)</code>	the stage stops as soon as its best cost reaches cost, in the stage's own cost
<code>& until_feasible()</code>	<code>.until_feasible()</code>	the stage runs <code>runner.with_hard_cost()</code> until the hard cost is zero; requires a <code>cost::hierarchical</code> cost
<code>& attempts(n)</code>	<code>.with_attempts(n)</code>	up to <code>n</code> runs while the target is not reached, keeping the best; the first stage starts each from a new initial solution, the others from the solution they received
<code>& restart(tag)</code>	<code>.with_restart(tag)</code>	the attempts after the first start from a new solution built as the <code>initialization</code> tag says, rather than as the first; checked at compile time against the stage's <code>SolutionManager</code>
<code>& max_evaluations(n)</code>	<code>.with_max_evaluations(n)</code>	the stage stops once it has made <code>n</code> evaluations, its attempts together; the next stage runs with what is left of the solve's budget
<code>& timeout(d)</code>	<code>.with_timeout(d)</code>	the stage stops once <code>d</code> (a <code>std::chrono</code> duration or seconds) has passed since it started, its attempts together; the next stage runs with what is left of the solve's time

With a `cost::hard_soft` cost expression, `until_feasible()` evaluates only the components of the hard branch, and deltas attached to soft components are ignored; with another expression producing a hierarchical cost it evaluates them all and keeps the hard part.

The pipeline's `.initialization(...)` and `.seed(...)` are those of every solver, and refer to its first stage. A caller's target applies to the last stage, unless that stage has its own: the solve then ends `target_reached` at the stage's own target, whatever the caller's. Stage names are checked when the pipeline is built, and a stage rejects run options that carry a control (`stage & with(control)`), which it would drop: the control goes to `solve()`, and a floating-point target for an integer cost, which would be truncated. After a cancellation, or once the solve's time or evaluations are spent, the stages between the first and the last are skipped without binding their runner (their report has 0 attempts and the reason as termination), and the last stage runs once, which only evaluates the solution, so the result still has the last stage's cost.

`pipeline.run(input, solution, rng, options...)` runs the stages from a given solution with the caller's RNG: the first stage's attempts all start from that solution, unless the stage restarts them (`& restart(initialization::random)`, a multi-start). It is how an app runs a pipeline registered by name.

An algorithm that declares itself deterministic (static constexpr bool deterministic = true, as First and Best Improvement do) repeats the same run from the same solution: a stage of one with more than one attempt that would all start from the same solution, without restart(...), is rejected with std::invalid_argument, which suggests restart(initialization::random). That is a stage after the first, at the pipeline's construction, or the first stage when it starts from the initial solution or, in an app, from the current solution, at the run; a custom algorithm without the trait is not checked.

The result is the last stage's, with the effort of every stage and attempt, and result.stages: per stage its name, attempts, evaluations, iterations, termination and cost (as cost::to_text writes it). A stage's termination is why it stopped, not why its best attempt did: a run cancelled, out of time or at the target, the stage's or the solve's budget spent, or else the end of its last attempt; the result's termination is the last stage's.

The parameters (configuration()) are each stage's under its name: its runner's (<name>.search.*, <name>.cost.*, ...), <name>.attempts and <name>.timeout (seconds; inf, the default, for no limit of its own) and <name>.max_evaluations (unlimited by default). Stage names must be distinct and non-empty: the pipeline's construction throws std::invalid_argument otherwise.

Algorithm stages

A pipeline [registered in an app](#) may name its stages by algorithm rather than by runner: stage<Algorithm>(name, parameters), or stage<Algorithm>(name, parameters, neighborhood) with a neighborhood recipe of its own. Such a stage has no recipes: when the app runs the pipeline, it becomes the stage of the runner of Algorithm on the app's SolutionManager and cost, and on the app's neighborhood or its own, with the recipes and the parameters the app had when it was bound.

```
using namespace easylocal::solvers;
auto application = app("tsp") | sm | two_opt
    | pipeline("cascade",
        stage<FirstImprovement>("feasible") & until_feasible() & attempts(5)
          & restart(initialization::random),
        stage<SimulatedAnnealing<Classic>>("anneal", {...}),
        stage("polish", polishing)); // a stage of a runner, with its own recipes
```

	Stage of a runner, stage(name, runner)	Stage of an algorithm, stage<A>(name, ...)
Where	any pipeline	a pipeline registered in an app
Cost and SolutionManager	the runner's own	the app's
Neighborhood	the runner's own	the app's, or the third argument
Options	& and the with_* methods	&: attempts, timeout, max_evaluations, target, until_feasible, restart
Parameters, under runners.<pipeline>.<stage>	attempts, timeout, max_evaluations, search.*, cost.*, neighborhood.*	attempts, timeout, max_evaluations, search.* and, with its own neighborhood, neighborhood.*

The two kinds mix in one pipeline. The app's cost.* parameters apply to every stage of an algorithm, where each stage of a runner has a copy of its own. until_feasible() derives the hard-cost stage from the app's cost, which must be hierarchical; a target is written in the stage's cost, as for a stage of a runner; restart is checked against the app's SolutionManager. A stage of an algorithm with its

own neighborhood is checked when it is registered, as a **runner with its own neighborhood** is: the neighborhood explores the app's Solution and the algorithm runs on it.

two_stage()

`solvers::two_stage(first, second)` is the pipeline of the hierarchical hard/soft model, (`stage("first", first) & until_feasible()`) | `stage("second", second)`: the second stage works on the whole cost. `two_stage(runner)` uses the same runner for both. Its parameters are `first.*` and `second.*`.

All solvers expose `supports_initial`, `supports_random`, `seed(n)`, `initialization(tag)` and `rng()`; for a pipeline they refer to its first stage.

Exceptions

An exception thrown during a run, by a hook of the problem (a cost component, a move of the neighborhood explorer, the SolutionManager) or by the tracer, is not caught: it ends the run and leaves `runner.run()` and `solver.solve()` as it is. A solver loses the runs before it: MultiStart's best start so far, a pipeline's earlier stages and their reports. The run emits no `run_finished`, so a trace ends with an unfinished run (**Tracing**). An app passes it on as well: `Session::run` leaves the current solution as it was and `last_run_effort()` empty, and `cli::run` writes error: `<what>` (unknown exception for what is not a `std::exception`) and returns 1.

Initialization

Tag	Starts from	Requires
<code>initialization::automatic</code> (the default)	a random solution when the SolutionManager builds one, else its initial solution	either
<code>initialization::initial</code>	<code>initial_solution()</code>	<code>initial_solution()</code>
<code>initialization::random</code>	<code>random_solution(rng)</code> , with the solver's RNG	<code>random_solution(rng)</code>

A tag is checked at compile time against the SolutionManager: a solver given one it does not support does not compile. Besides automatic, there is never a fallback from one to the other.

Design choices

- **The solver owns randomness.** One RNG feeds random initialization and random-aware runners, so a seed reproduces the whole solve. The RNG is seeded once, at construction or with `.seed(...)`: each `solve()` continues its stream, so a seed reproduces the sequence of solves, and a second `solve()` differs from the first. Seed again to repeat one.
- **Runners stay solution-to-solution.** Construction belongs to solvers, which keeps runners composable.
- **One way to chain runners.** The hierarchical hard/soft model is a pipeline of two stages, so `two_stage()` and a longer pipeline share their options, their result and their parameters.

Configuration

<easylocal/config/*.hpp> (easylocal::config); TOML in <easylocal/adapters/toml.hpp>

Parameters are plain structs that describe themselves; configurable objects expose them as a **parameter set**: paths, descriptions and textual values, which frontends (command line, files, TOML, the TextUI) read and change.

Parameter blocks

```
struct MyParameters
{
    easylocal::parameter<std::size_t, "size"> size{10}; // 10 is its default
    easylocal::parameter<double, "bound"> bound{};      // required: no default
    Schedule schedule{};                               // a parameter block

    static consteval auto parameter_schema()
    {
        return config::fields(
            config::field<&MyParameters::size>(
                "Description", config::range(1, easylocal::unlimited)),
            config::field<&MyParameters::bound>(
                "Description", config::range(0.0, easylocal::unlimited)),
            config::group<"schedule", &MyParameters::schedule>("Description"));
    }

    config::validation_result validate() const; // success() or failure(reason)
};
```

Every field is an `easylocal::parameter<Value, "name">`, which holds the value and whether anything has set it; `field<&MyParameters::size>` reads the name from the member instead of repeating it. **The default of a field is its member initializer, and a field declared `bound{}` has none**: nothing sets it, so `validate()` reports `bound: not set`, and it has no default, and reading it before a value arrives throws `parameter_not_set`. A plain member does not compile: its default would be implicit.

A parameter converts to its value, so it reads like the value it holds (temperature * `parameters().cooling_rate`); where a template deduces its argument, as `(std::max)(a, b)` does, or where the value is a class with its own templated operators, as `std::string` compared with a literal is, `value()` spells the value out.

The value may be an integral or floating-point type, `bool`, `easylocal::limit`, `std::string`, `std::filesystem::path`, and `std::arrays` and `std::vectors` of those (written `[a, b, c]`; `[]` is an empty vector; see [Values as text](#)). A group nests another block: its fields are under `schedule.`, and `config::check_schema` of the enclosing block runs its `validate()` once the enclosing block's own fields pass, so that the enclosing block's `validate()` checks it too.

Domains

A field may declare the values it takes as the second argument of `field`:

```
config::field<&P::cooling_rate>(
    "Multiplicative cooling factor", config::range(0.0, 1.0).open()),
config::field<&P::samples>("Proposals", config::range(1, 1000).log()),
config::field<&P::policy>("Tabu list", config::one_of("fixed", "random")),
```

Domain	Values
<code>range(low, high)</code>	the numbers from <code>low</code> to <code>high</code> , both included, of one type; for a number or a limit
<code>range(low, easylocal::unlimited)</code>	the numbers from <code>low</code> up, with no upper bound, its upper end included: infinity for a floating-point field, unlimited for a limit ([1, unlimited]); <code>.open_high()</code> leaves the end out, a finite number or a count ([1, unlimited])
<code>.open()</code> , <code>.open_low()</code> , <code>.open_high()</code>	the same range without both bounds, the lower or the upper one
<code>.log()</code>	the same range, which a configurator samples on a logarithmic scale; <code>low</code> must be positive
<code>one_of(a, b, ...)</code>	the values given: text for a <code>std::string</code> , numbers for a number (numbers of different types are held as their common type: <code>one_of(1, 1.5, 2)</code> holds 1.0, 1.5 and 2.0)
<code>easylocal::unlimited</code>	any value of the field's type: a seed, free text, a path, a number that may be negative

The name of a field or group is a segment of its paths: a letter or `_`, then letters, digits and `_`. The schema is computed once at compile time, so a mistake in it does not compile, and the error names `parameter_schema()`.

For an array or a vector the domain applies to each element. The library handles NaN and infinity in parameters and costs (a domain rejects NaN, a range with no upper bound includes infinity): `-ffast-math`, which assumes neither occurs, breaks these checks. A domain that does not fit the field's type, a range whose bounds are not in order or a logarithmic range from zero do not compile, and neither do a range of two types, `range(0.0, 1)`, or a count where a domain goes, `field<...>("...", 5)`: `easylocal::unlimited` is a tag of its own type, `unlimited_t`, which converts to the unlimited limit and which no number converts to.

Enumerations

A field may be of an enum type, which a configurator reads and writes by name. C++ cannot read the names of an enum, so they are declared once, beside it:

```
enum class Start { random, initial, greedy };

template<>
struct easylocal::enumeration<Start>
{
    static constexpr std::array values{
        easylocal::enumerator{"random", Start::random},
        easylocal::enumerator{"initial", Start::initial},
        easylocal::enumerator{"greedy", Start::greedy}};
};
```

The field is then declared and used like any other, and **the values declared are its domain**:

```
easylocal::parameter<Start, "start"> start{Start::random};
// in the schema:
config::field<&Parameters::start>("Where the search starts"),
```

--start greedy on the command line, start = "greedy" in a configuration file or in TOML, a list in the TextUI, {random, initial, greedy} in --help, and a categorical parameter for irace, c ("random", "initial", "greedy"). A name that is not declared is rejected with expected one of random, initial, greedy, and a value cast from a number that no enumerator has fails validation like any value outside its domain.

config::one_of(Start::random, Start::initial) as the domain of the field keeps only the values it gives, when a field takes fewer than the enum declares. A field of an enum with no easylocal::enumeration does not compile: its values would have no names.

Conditions and requirements

```
config::field<&P::initial_acceptance>("...", config::range(0.0, 1.0).open())
  .only_if(config::value<"calibration_samples"> > 0),
config::require(
  config::value<"final_temperature"> < config::value<"initial_temperature">,
  "final_temperature must be smaller than initial_temperature"),
```

Declaration	Meaning
field(...).only_if(expression)	the field matters only when the expression holds; otherwise its domain is not checked and a configurator does not tune it
require(expression, message)	an element of fields(...): the expression must hold, else the block is invalid with message (which must outlive it)
value<"name">, value<"group.name">	a field of the block, or of a nested group, by its path relative to the block; a path that names no field does not compile

Expressions combine values and constants (numbers, bool, text) with <, <=, >, >=, ==, !=, &&, ||, !, +, -, * and /. Numbers are computed in double, as R computes them, so that an expression means the same in its irace export: 7 / 2 is 3.5, a limit is its count and unlimited is +infinity, and -count is negative for an unsigned count. config::evaluate(expression, block) computes one, config::is_active(field, block) tells whether a field's condition holds, and describe_expression(expression, prefix) gives an expression_info, its text with the full paths of its references (text_with, references): in R syntax, booleans are TRUE and FALSE, infinity and NaN Inf and NaN, and a text escapes its quotes and backslashes.

Checking a schema

The validation of a parameter set checks each block against its schema before its validate(): the fields that matter outside their domains, with the field's path (expected a value in (0, 1), got 1.5), and the requirements that do not hold, with the block's path and their message; then each nested group, under its own path; the block's validate() runs only when they pass. A validate() checks the same with config::check_schema, so that a block made in the code is checked too; its reason, the first that fails, has the same text, with the field's name first. A validation_result owns its message, so failure() takes one built at run time as well as a literal:

```

config::validation_result validate() const
{
    if (const auto schema = config::check_schema(*this); !schema)
        return schema; // "cooling_rate: expected a value in (0, 1), got 2", or a
        requirement's message
    // ... the checks the schema does not declare
}

```

`config::require_valid(block)` returns the block when it passes these checks, its groups included, and throws `std::invalid_argument` otherwise, the path of the field first (temperature.cooling_rate: expected a value in (0, 1), got 1.5); `config::require_valid(set)` does the same for a parameter set, with each invalid block as `<path>: <message>`. The runners, their policies and the neighborhood recipes call it on the parameters they are made from, and an app on its whole configuration when it is bound.

Every field declares a domain: `check(app, ...)` fails for a parameter of the app without one, and the library's tests for a built-in one; a boolean's domain is true and false. A field that takes any value says so with `easylocal::unlimited`. `config::undeclared_domains(set)` lists the paths of the parameters of a set that declare none. Automatic configurators read the domains: a domain is where the values are valid, not necessarily where they are worth trying, and a program can narrow it for tuning; a range with no upper bound, or any value, is not tuned until it is given a finite range.

Parameter sets

```

config::parameter_set parameters;
parameters.add("application", app_parameters); // a block
parameters.add("solver", runner.configuration()); // another set, under a prefix

```

Member	Purpose
<code>add([prefix,] block)</code>	the fields of a parameter block; read-only when the block is const
<code>add([prefix,] object)</code>	an object with <code>parameters()</code> and <code>configure(block) -> validation_result</code> , for objects that rebuild something from their parameters
<code>add([prefix,] set)</code>	the parameters of another set
<code>parameters()</code>	every parameter: path, description, value (as text), <code>read_only</code> , kind (boolean, integer, real, limit, text, path, list), domain (a <code>domain_info</code> , empty when none is declared), <code>active</code> (whether its condition holds) and <code>condition</code> (an <code>expression_info</code> with full paths, empty when it has none)
<code>requirements()</code>	the requirements of every block: path of the block, <code>message</code> , <code>expression</code> (full paths) and <code>satisfied</code>
<code>validate()</code>	the fields outside their domains, the requirements that do not hold and the diagnostics of every block's <code>validate()</code> , by path: a <code>parameter_validation_result</code> , true when there are none, with a <code>parameter_diagnostic{path, message}</code> for each
<code>apply(text_overrides)</code>	apply path = value overrides (views, or overrides that own their text, as a file gives them), all or none; its result must be read

A set refers to the objects it was built from: they must outlive it and stay in place. Adding a path that is already in the set throws `std::invalid_argument`.

`apply` parses each override into a copy of its block and validates the copy, and validates every other block as it is; only when every override names a parameter and every block is valid are the copies committed, so a batch may repair invalid defaults, a requirement of an enclosing block included. `load_and_apply` applies the file and the command line in one such batch. Errors: `unknown_parameter`, `duplicate_path`, `parse_error`, `validation_error` (with the path of the invalid block), `read_only_parameter`.

Configurable objects

One rule makes a class configurable: its `parameters_type` is a parameter block, and it is constructed from it. The framework holds the parameters in the class's recipe or registration, builds the class from them when a runner or an app is bound (again after they change), and exposes them at the path of the class's role:

Class	Constructed as	Its parameters given by	Path in a runner	Path in an app
an algorithm	<code>A(parameters)</code>	<code>make_runner<A>(parameters),</code> <code>runner<A>("name",</code> <code>parameters),</code> <code>stage<A>("name",</code> <code>parameters)</code>	<code>search.*</code> <code>runners.<pipeline>.<stage>.search.*</code>	<code>runners.<name>.*;</code> <code>runners.<pipeline>.<stage>.search.*</code> for a stage
a neighborhood explorer	<code>NHE(sm, neighborhood<NHE>(parameters,</code> <code>parameters,</code> <code>args...))</code>	<code>neighborhood.*;</code> <code>args...)</code>	<code>runners.<name>.neighborhood.*</code>	for a runner's own
a SolutionManager	<code>SM(input,</code> <code>parameters,</code> <code>args...)</code>	<code>solution_manager<SM>(parameters,</code> <code>args...)</code>	<code>solution_manager.*</code>	<code>solution_manager.*</code>
a cost component	<code>C(input,</code> <code>parameters,</code> <code>args...)</code> or <code>C(parameters,</code> <code>args...)</code>	<code>component<C>(parameters, cost.<name>.*</code> <code>args...)</code>	<code>cost.<name>.*</code>	<code>cost.<name>.*</code>
a <code>cost::apply</code> function	<code>F(parameters)</code>	<code>cost::apply<F>(parameters,</code> <code>children...)</code>	<code>cost.<name>.*</code>	<code>cost.<name>.*</code>

Deriving from `easylocal::parameters_base<P>` declares `parameters_type`, takes the block in the constructor and gives it back through `parameters()`; it is optional, and a class that declares them itself is configured just the same.

Without a first argument of the `parameters_type`, the parameters are the defaults. A component or a function is configured under its `name()`, which must be static, wherever it is in the cost expression. Only `parameters_type` is read: a class with a `parameters()` that returns a parameter block, or a `configuration()`, and no such `parameters_type` does not compile, and the message says what to write. An algorithm registered in an app whose `parameters_type` is not a block still runs, unconfigured, and `check(app, ...)` reports it as runner parameters.

The structure of the framework's own objects is configurable too: the weights and tolerance of a cost expression, the biases of a neighborhood union, the budgets of pipeline stages. An object gives its parameters when `configuration()` returns a `parameter_set`, with paths relative to it; whoever composes it chooses the prefix.

The set refers to the object, so configure the object that will run, after its last copy: `configuration()` does not compile on a temporary `((make_runner<A>() | sm | nhe).configuration())`, whose set would refer to an object already gone.

Provider	Paths
a parameterized algorithm (FirstImprovement, BestImprovement, HillClimbing, GreatDeluge, LateAcceptanceHillClimbing, ParetoLateAcceptanceHillClimbing, SimulatedAnnealing, the Tabu Search family), through its runner	the fields of its parameters_type; temperature.* for Simulated Annealing, tabu_list.* for Tabu Search
the cost expression of a SolutionManager recipe	weights of a cost::sum; children by position (0.*, 1.*), hard.* and soft.* of a cost::hard_soft; tolerance.* of a cost::approximately; each component and cost::apply function with a parameters_type under its name() (<name>.*)
neighborhood_union with random_biases	random_biases, and each child's parameters under its position (0.*, 1.*)
neighborhood<NHE>(parameters, args...) for an explorer with parameters_type	the explorer's parameters
solution_manager<SM>(parameters, args...) for a SolutionManager with parameters_type	the SolutionManager's parameters, as solution_manager.* in a runner or an app
runner.configuration()	search.*, cost.*, solution_manager.*, neighborhood.*
an app, app.configuration() (also a Session's)	cost.*, solution_manager.*, neighborhood.*, runners.<name>.*
MultiStart, LocalSearch, Pipeline solvers	starts and the runner's (MultiStart), the runner's (LocalSearch), each stage's runner and its own attempts, timeout and max_evaluations under its name (Pipeline; first.* and second.* for two_stage()); an algorithm stage of an app's pipeline has its algorithm's search.* and its own neighborhood's, the cost and the SolutionManager being the app's

Values as text

Every frontend reads a value from the same text, and config::format_value writes it back in that syntax:

Type	Text
<code>bool</code>	true or false
an integer	a decimal number within the type's range (no sign for an unsigned type)
a floating-point number	a decimal or scientific number, <code>inf</code> , <code>-inf</code>
<code>easylocal::limit</code>	a count, or unlimited
<code>std::string</code> , <code>std::filesystem::path</code>	the text as it is; a path in UTF-8, on Windows too
an enum with an <code>easylocal::enumeration</code>	the name of the value (Enumerations)
<code>std::array</code> , <code>std::vector</code>	[a, b, c], the brackets optional; [] is an empty vector, and an array needs exactly its size

Spaces around a value or an element are ignored, except in a text. A list is split at the commas outside nested brackets, `[[1, 2], [3, 4]]` included, and there is no quoting: a text element of a list cannot contain a comma or a bracket, so `format_value` and the parser are each other's inverse except for such lists. An error names what was expected, the element of a list by its position (element 2: expected a number), and the size of an array (expected 2 elements, got 3).

Frontends

Function	Purpose
<code>config::parse_cli(argc, argv)</code>	the overrides <code>--path=value</code> and <code>--path value</code> , <code>--config <file></code> and <code>--help</code> of a command line, with its errors; the arguments are UTF-8 text, on Windows too
<code>config::parse_config_text(text)</code>	the overrides of the text of a configuration file, as <code>load_config_file</code> reads them
<code>config::overlay_overrides(base, top)</code>	one batch of two: the overrides of <code>top</code> replace those of <code>base</code> with the same path
<code>config::require_valid(set)</code>	throws <code>std::invalid_argument</code> unless every block is valid
<code>config::undeclared_domains(set)</code>	the paths of the parameters that declare no domain
<code>config::load_and_apply(argc, parameters[, read_config])</code> <code>argv,</code>	apply <code>--config <file></code> and <code>--path.to.field=value</code> ; the file is read by <code>read_config</code> , a <code>config_file_reader</code> , <code>load_config_file</code> by default, <code>load_toml_file</code> for a TOML file
<code>config::load_config_file(path)</code>	the overrides of a file of <code>path = value</code> lines: <code>#</code> starts a whole-line comment (a <code>#</code> after a value is part of it), a UTF-8 byte order mark is skipped, a directory is an error
<code>config::cli_help(program, parameters)</code>	help text: each parameter that can be changed, with its description, its values (the domain; <code>true</code> or <code>false</code> for a boolean), when it matters (only if ...) and its current value
<code>config::print_diagnostics(out, result)</code>	report the errors of <code>load_and_apply</code> , or of a <code>config_file_parse_result</code> , one per line, those of a file with their line and column
<code>config::format_value(value)</code>	a value as text, in the syntax overrides use
<code>config::load_toml_file(path)</code> (TOML adapter)	a <code>config_file_reader</code> : the overrides of a TOML file, each value by its TOML type (a boolean, an integer, a float that sets only a floating-point field, a string, or an array of numbers, booleans and such arrays; not of strings), as a <code>config_file_parse_result</code> ; a <code>parse_error</code> gives its line and column and the file as its text, an <code>unsupported_value</code> its path

Design choices

- **Configuration is independent of construction.** A set refers to live objects; nothing is rebuilt from a parameter box, and binding a runner later sees the applied values.

- **Typed fields, no reflection macros.** Schemas are consteval descriptions of member pointers, checked at compile time; only the paths are matched at run time, as the command line and files already require.
- **Relative paths.** Components do not choose their own top-level names, so one program can combine several of them under prefixes of its choice.
- **Validation lives with the parameters**, so every frontend reports the same errors.

Apps and tools

<easylocal/app/app.hpp>, app/check.hpp, app/session.hpp, app/run_parameters.hpp; adapters
<easylocal/adapters/tui.hpp>, <easylocal/adapters/rest.hpp>

An **app** names a problem graph (SolutionManager and neighborhood recipes) and the runners available on it. Tools consume apps.

Building an app

```
auto application = app("name")
    .with_solution_manager(sm)
    .with_neighborhood(nhe)
    .with_runner<Algorithm>("runner-name", parameters); // parameters optional

// equivalently
auto application = app("name") | sm | dhe | runner<Algorithm>("runner-name",
parameters);
```

The components are added in this order: a neighborhood needs the SolutionManager, a runner needs both. The app may be used before the last step, so that a model is tried a component at a time: one with a SolutionManager and a neighborhood binds and opens in the tools, with no runner to pick (the tester's Run page lists none, and cli::run reports that none is registered). A neighborhood is still needed to bind; a tester on an app without one is on the [roadmap](#).

A registered algorithm must expose a default-constructible parameters_type and be constructible from it. Its parameters are configurable under runners.<name>.* when parameters_type is a parameter block; otherwise the app still runs it, and check(app, ...) reports it as runner parameters (unless the type is empty). The same algorithm may be registered under several names.

A runner runs on the app's neighborhood, unless it brings its own, the third argument of runner (or of with_runner). A registration is checked when it is added: the app's neighborhood explores the Solution of its SolutionManager, and each algorithm runs on its neighborhood (FirstImprovement needs moves(), Simulated Annealing random_move()), so a mismatch fails to compile at the registration, not where a tool first runs the runner:

```
auto application = app("tsp") | sm | two_opt
    | runner<FirstImprovement>("fi")
    | runner<FirstImprovement>("fi-swap", {}, neighborhood<Swap>() | delta<Length,
SwapDelta>());
```

Its neighborhood is built over the app's SolutionManager when the app is bound, next to the app's own, and its parameters are under runners.<name>.neighborhood.*. The registration is checked at compile time: the neighborhood explores the app's Solution and the algorithm runs on it. check(app, ...) checks every neighborhood, the runners' own included; the Session, the TextUI and REST show and apply the moves of the app's neighborhood. make_runner<A>("name") requires the runners of A to run on neighborhoods of the same recipe type.

The names of the runners and pipelines of an app must be non-empty, distinct, and made of letters, digits, _ and -: each is the key of a registration and a segment of its parameter paths (runners.<name>.*). configuration() and bind() throw std::invalid_argument otherwise, and check(app, ...) reports it as registration names.

The parameters of the app are checked when it is bound, and when a Session or a REST blueprint is made on it: an invalid value throws std::invalid_argument with its path,

runners.sa.temperature.cooling_rate: expected a value in (0, 1), got 2, and check(app, ...) reports it as app configuration.

Pipelines

An app also registers [pipelines](#), beside its runners and under names of the same list:

```
using namespace easylocal::solvers;
auto application = app("tsp") | sm | nhe
    | runner<runners::FirstImprovement>("fi")
    | pipeline("cascade",
        stage("feasible", descent) & until_feasible() & attempts(5),
        stage("climb", climbing));
// or pipeline("cascade", (stage(...) & ...) | stage(...)), or .with_pipeline(...)
```

A stage may also be an algorithm, stage<FirstImprovement>("descent", {...}) (or with its own neighborhood, stage<A>("a", {...}, neighborhood<Swap>())), which runs on the app's recipes: the app's SolutionManager and cost, so cost.* apply to it, and the app's neighborhood or its own. Its parameters are runners.<name>.<stage>.search.*, attempts, timeout, max_evaluations and, with its own neighborhood, .neighborhood.*; a stage whose own neighborhood does not fit is rejected at compile time.

A stage of a runner keeps its own recipes; the stages must have the app's Input and Solution, and the last one the app's cost (checked at compile time). pipeline(name, ...) throws std::invalid_argument when two stages have the same name, or one has none. For every tool a pipeline is one more runner: it is listed among the runners, run by name from the current solution (pipeline.run(input, solution, rng), so its first stage's attempts all start from that solution), and configured under runners.<name>.<stage>.* (--runners.cascade.climb.search.*, runners.cascade.feasible.attempts).

app("name") returns an App, and each | a new App type, which spells out all its recipes: let auto deduce it, or a function that returns it. The name of a registration is its key: the members below find a runner by it, never by its algorithm type or its position, so an algorithm may be registered twice ("sa-fast", "sa-slow").

Member of App	Purpose
<code>name()</code>	the app name
<code>runner_names()</code>	the names of the runners and pipelines, in the order of registration
<code>runner_parameters<A>("name")</code>	the stored parameters of the runner of algorithm A registered as name; <code>std::invalid_argument</code> for an unknown name
<code>configuration()</code>	the app's parameters as a <code>config::parameter_set::cost.*</code> , <code>solution_manager.*</code> , <code>neighborhood.*</code> and <code>runners.<name>.*</code>
<code>bind(input)</code>	the <code>BoundApp</code> : services built once for input, which it borrows (a temporary <code>Input</code> is rejected)
<code>run("name", input, solution, rng, options...)</code>	run the runner or pipeline registered under a name; <code>std::optional<named_run_result></code> , empty for an unknown name
<code>make_runner<A>("name")</code>	a standalone <code>Runner</code> with that runner's parameters and the app's recipes, for a solver: <code>make_solver<Solver>(application.make_runner<A>("name"))</code>
<code>check_registration_names()</code>	throws <code>std::invalid_argument</code> unless the names are valid
<code>check_configuration()</code>	throws <code>std::invalid_argument</code> unless the names and every parameter are valid, each invalid block as <code><path>: <message>; bind()</code> calls it
<code>with_solution_manager(spec), with_neighborhood(spec), with_runner<A>("name"[, parameters[, neighborhood]]), with_pipeline(registration)</code>	the method spellings of the pipe
<code>registration_count</code>	the number of runners and pipelines registered

`run` binds the app to the `Input` for that run only, with the current runner parameters: the app and the `Input` may be shared by concurrent runs. A `BoundApp` offers `input()`, `solution_manager()`, `neighborhood_explorer()` and `run("name", solution, rng, options...)` on services built once; each run builds its algorithm from the parameters the registration had at `bind`, so runs do not share an algorithm's state. `Runner::bind` returns a `BoundRunner`, with `input()`, `solution_manager()`, `initial_solution()`, `random_solution(rng)`, `better(a, b)` and `run(solution, args...)`.

`named_run_result{solution, cost, evaluations, iterations, termination, front}` keeps what every runner result provides (`search_result_for`): a runner chosen by name may be any algorithm, built-in or your own, each with its own result type. `evaluations`, `iterations` and `termination` are the result's, as in `search_result`, and 0, 0 and completed for a result without those members. `front` holds the non-dominated solutions of a run with a `cost::pareto` cost, as `pareto_search_result` does (see [Runners](#)), and is empty otherwise. The `Session` keeps the front of its last run (`last_run_front()`), `cli::run` prints it and REST returns it with the solution; the `TextUI` does not show it yet.

Tools

Tool	Purpose
<code>check(app, input[, solution][, options]) -> app_check_report</code>	contract checks of the composed problem, with <code>testing::check_options</code> (samples, random solutions, seed, cost tolerance); <code>print_report</code> , <code>composition()</code>
<code>cli::run(app, argc, argv[, options]) -> int</code>	the app as a command-line program (<code><easylocal/app/cli.hpp></code>): see below
<code>tui::run(app, options), tui::run_launcher(options, apps...)</code>	interactive terminal tester (TUI component, FTXUI); <code>tui::options</code>
<code>rest::blueprint(prefix, app, codec, options)</code>	Crow blueprint: asynchronous runs, status, cancellation, solutions (REST component); see REST

`tui::run_launcher(options, apps...)` opens a list of apps over the same problem. The launcher owns the Input, read from `options.testster.input_path` when it is set, and the current solution: its first entry, *Input and solution*, is a tester with the Input/Output page only, which loads and saves them; each app opens on them in the complete tester, and what a tester leaves, loaded or computed, becomes the shared state. Each app keeps its own session while the launcher runs: its runner and problem parameters and its seed stay when it is left and opened again. The apps must have the same `SolutionManager` recipe, cost included, which a `static_assert` checks; they differ in the neighborhood and the runners.

Tools give stochastic runners an RNG they own, from a configurable seed: the TextUI seed option (also editable on its Run page) and REST's per-run seed (default `blueprint_options::seed + run id`). Each run of a Session, and so of every tool, gets a generator of its own, seeded with one draw of the session's RNG: the same seed and the same commands give the same runs in every frontend (the conditions are in [Stability](#)). `run("name", ...)` passes the RNG to the algorithm only if it takes one, so deterministic runners ignore it.

The TextUI edits the app's parameters in modal windows: G on the Run page opens the selected runner's (`runners.<name>.*`) before running it, P the problem's (`cost.*`, `solution_manager.*`, `neighborhood.*`). Values are checked before anything changes and stay for the rest of the session. Its *Target cost* field, when filled, stops each run at the first solution that reaches it; below it, the current cost is shown in the same syntax. Its *Stop after* row has two fields: *seconds*, when filled, stops each run after that many seconds (the progress shows the time elapsed, after the evaluations, the iterations and the best cost found so far); *evaluations* does the same with a number of evaluations. The result of the last run is shown below the controls and in the status line: the costs before and after, why the run ended and its evaluations, such as "(time limit reached, 812 evaluations)", when the runner reports them. In a terminal too small for a page, the page scrolls to the focused control (Tab, arrows).

`cli::run` parses the command line and a `--config` file with `config::load_and_apply`: its own block `cli::CommandLineParameters` at the root (`instance`, `seed`, `runner`, `start`, `solution`, `output`, `target`, `timeout`, `max_evaluations`, `report`, `trace`), the app's `configuration()`, and `options.program_parameters`, the program's own set; `options.defaults`, a `cli::CommandLineParameters`, gives the values of its switches before the command line, and `options.read_config` reads the `--config` file: `config::load_config_file` by default, `config::load_toml_file` for a TOML file (TOML adapter). It then builds a Session with the seed, loads the Input, takes the starting solution (`--solution`, else `--start`: random by default when the problem has `random_solution`, initial otherwise), runs the runner by name, with `stop_at` when `--target` is set (a pipeline gives it to its last stage, unless that stage has a target of its own, which then decides when the run ends `target_reached`), `timeout` when `--timeout` is (seconds) and `max_evaluations` when `--max_evaluations` is, and writes cost, time, the session's `last_run_effort()`

(iterations, evaluations, termination), with `--report` the session's `cost_report()` (a line component `<name> <value>` for each component, followed by its description, indented), and the solution (or saves it to `--output`) to `options.out`; errors go to `options.err`. After a run with a front (a `cost::pareto` cost) it writes a line front `<n>`, then for each point, in the order of the front, a line point `<i> cost <cost>`, from 1, followed by its solution. Its numbers read back whatever the locale of `options.out`: no decimal comma, no grouped digits. With `--output` the solutions of the front go instead to numbered files next to it, one solution per file as `save_solution` writes it, which `--solution` reads back: `--output best.txt` gives `best.1.txt`, `best.2.txt`, ... With `--trace <file>` the run is recorded, with timestamps and with the program and its parameters as metadata, by a `jsonl_recorder` for a `.jsonl` name and a `binary_recorder` otherwise ([Tracing](#)); a file that cannot be written is an error (1), a cost the format cannot encode an invalid command line (2). It returns 0, 2 for an invalid command line, an unknown runner, an app with no runner registered or a `--solution` that is not valid for the Input (error: solution: ...), 1 when the run throws (error: unknown exception for what is not a `std::exception`). It requires the `read_input` hook, and the solution hooks only when the corresponding switches are used.

Tuning with irace

`<easylocal/app/tuning.hpp>`; tutorial: [chapter 13](#).

`cli::run` also reads the block `easylocal::TuningParameters` under `tuning`:

Switch	
<code>--tuning.irace=DIR</code>	write an irace scenario to DIR, which it creates, and exit without loading the Input
<code>--tuning.print=cost, cost_time</code>	print only the cost as one number (<code>scalar_cost</code>), and the running time in seconds after it with <code>cost_time</code>
<code>--tuning.hard_weight=W</code>	the weight of a hard cost over a soft one in that number; 10^9 by default

`options.tuning`, a vector of `tuning_range{path, domain}`, gives the values to try for parameters instead of the domains of their schemas; a range must lie within the declared domain and name a parameter of the app or of `options.program_parameters`, or the export fails with status 2.

`write_irace_stub(irace_stub) -> irace_stub_result` writes the scenario:

- `parameters.txt`: the parameters with a finite domain, as name `--path=" type (values)`, where name, the irace identifier, is the path with every character but letters, digits, `.` and `_` turned into `_` (a numeric suffix tells apart two paths that would get the same identifier, and runner is reserved); conditions, [forbidden] and `configurations.txt` use the identifiers, and `configurations.txt` maps them back through the switch; with `r`, `i` (`,` log on a logarithmic range), `o` for a set of numbers and `c` for text and the runner; real values have irace's 4 digits after the point, or as many as a bound needs (at most 15, written as digits in a [global] section), and an open real bound moves inward by one such step, an open integer bound to the nearest integer inside the range. With several runners a categorical runner is added and each runners `<name>.*` gets the condition `| runner == "<name>"`; a runner chosen with `--runner` is the only one written. A field's `only_if` condition becomes an irace condition when it names tuned parameters, those that are not tuned replaced by their values; when it names none, it is decided at once, and a parameter whose condition is false is commented out as inactive. Each require that names a tuned parameter becomes a [forbidden] expression, `!(...)`, after its message; with values in place of some names, the expression with every name follows as a comment, and so does the condition of a commented-out line. The built-in runners and policies state every rule between their parameters (a minimum not above its maximum) as a `require`, so irace samples no configuration they reject. The other parameters, whose domain has no upper bound or is any value, are commented out,

with a range of a factor of ten around a positive value, within the lower bound, to start from, or (LOW, HIGH) with the lower bound when it has one; lists, paths, text and unlimited limits are noted, not written. `cost.*` is never written: it defines the cost that irace compares.

- `fixed.conf`: the parameters the command line changed when the stub was written (but instance, seed, solution, output, report, tuning.irace and tuning.print; tuning.hard_weight stays, so that every run weighs the hard cost the same), with paths made absolute, read by each run with `--config`; the tuned values override them.
- `target-runner`: a shell script that runs the program, by its absolute path, with `--config fixed.conf --tuning.print=cost --instance=... --seed=...` and the candidate's switches; a bound passed by irace is ignored. It does not run on Windows.
- `instances.txt`, `scenario.txt`: stubs, with the program's instance and a budget of 1000 runs.
- `configurations.txt`: the current values as a first configuration, NA for the parameters of the runners that are not the first and for those whose condition is false.

The files that exist are kept, so that the user edits them; only `configurations.txt` is rewritten, each time from `parameters.txt` as it is on disk (up to a [global] or [forbidden] section), with each value moved into its range when it lies outside (moved lists them; unlimited moves to the upper bound). A parameter of `parameters.txt` that the program does not have is an error.

`scalar_cost(input, cost, hard_weight) -> double` is the number irace compares: the problem's `scalar_cost(const Input&, const Cost&)` when one is found by argument-dependent lookup, else `cost::scalar(cost, hard_weight)` (<easylocal/cost/scalar.hpp>): a number as it is, a hierarchical cost as `hard * weight + soft`, a lexicographic one as the sum of each value times weight to the number of values after it. `scalar_cost_available<Input, Cost>` tells whether either applies; without it, `--tuning.print` and `--tuning.irace` fail with status 2.

A program that reads its configuration from the command line itself adds `easylocal::RunParameters`, the limits of a run (target, timeout in seconds, max_evaluations, the switches `cli::run` has at its root), under a prefix of its choice:

```
easylocal::RunParameters run;
configuration.add("run", run);    // --run.target=0 --run.timeout=10
// ... once the Input is loaded:
session.run("sa", run.options<Cost>(session.input()));
```

`run.options<Cost>(input[, base])` gives the run options of the limits that are set, added to base (such as `with(control)`); `run.target_cost<Cost>(input)` gives the target alone as a `std::optional<Cost>`, empty when none is set, for programs that run a runner or a solver directly. Their errors name the field. `cli::CommandLineParameters::run_parameters()` gives the block of `cli::run`'s switches.

The target stays text until the Input is known, because a problem may read its costs with its own `read_cost`. A program that runs a runner without a Session reads it with `easylocal::read_cost<Cost>(input, text)`, which the Session also uses: the problem's `read_cost` when it has one, else `cost::from_text`; `easylocal::readable_cost<Input, Cost>` tells whether either applies.

Sessions

A Session (<easylocal/app/session.hpp>) is an app at work on one Input: an owned Input, the app bound to it, a current solution, a selected move and an RNG, with the commands that change them. It is how a program runs an app headless, and the model of an interactive frontend: the TextUI is a view on it, and a GUI or a web frontend would be another.



Syntax error in text

mermaid version 11.17.2

Constructor

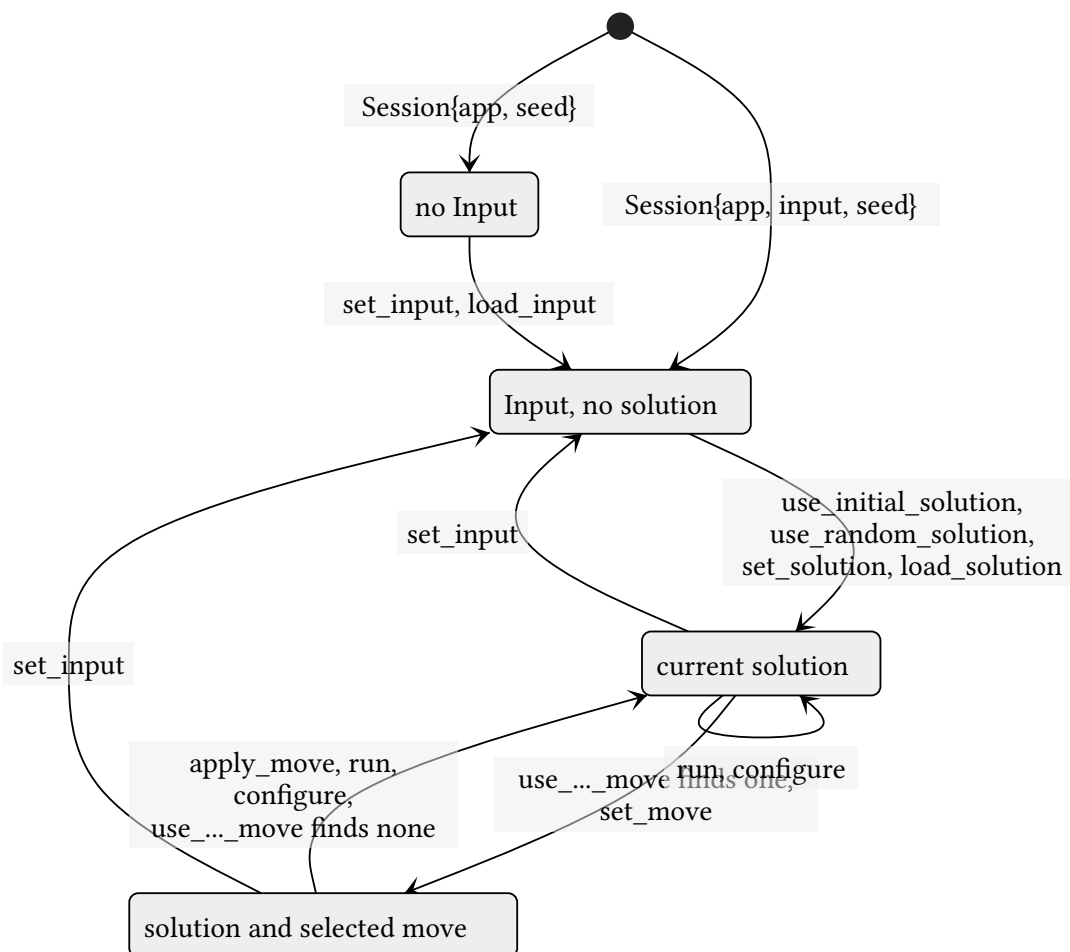
<code>Session{app, input[, seed]}</code>	a session on input, which it owns; another Input is another session
<code>Session{app, std::shared_ptr<const Input>[, seed]}</code>	a session on an Input it shares with other owners, without copying it
<code>Session{app[, seed]}</code>	a session without an Input yet, for frontends that load one (set_input, read_input, load_input)

Commands

Input	set_input, read_input(in) from a stream, load_input(path) from a file (the I/O hooks of Problem model), input, has_input, input_handle (the shared Input, null without one)
Solution	use_initial_solution, use_random_solution(), set_solution, read_solution(in) and write_solution(out) on streams, load_solution(path) and save_solution(path) on files, solution, has_solution, is_valid, evaluate, check()
Move	select with use_first_move, use_next_move, use_first_improving_move, use_best_move, use_random_move() or set_move; then has_move, move, move_is_valid, evaluate_move, evaluate_move_fully, move_evaluation_matches_full([tolerance]), apply_move
RNG	rng(), the session's generator, and set_seed(seed); the seed of every constructor is 0 by default; random solutions and moves are drawn with it, and each run draws its own generator from it
Neighborhood	neighborhood_preview, neighborhood_statistics, check_neighborhood_costs([tolerance]) (the delta and the full costs agree by equivalent or within a cost::tolerance, 1e-9 by default), check_move_independence (needs solutions_equal: the SolutionManager's equal or Solution::operator==), check_random_move_distribution([rounds_per_move]) (draws the session's RNG, 20 rounds per move by default; needs Move::operator==); with a totally ordered cost these two compare a value only with those of the same cost, about linear in the moves, otherwise with all of them, quadratic. Each takes a std::stop_token as its last argument (none by default), stops at the next move once it is requested, and says so in its result's stopped
Runners	runner_names (the runners and pipelines), run("name", options...) (replaces the current solution; the options are run options: with(control, tracer), stop_at, timeout, max_evaluations), last_run_effort(): the evaluations, iterations and termination of the last run, as named_run_result has them (empty before the first run, after a new Input or a run that did not complete), last_run_front(): read_position(), a cost::value as text, apply with a cost::Percent cost the front of the last run, a vector of pareto_point(solution, cost) and a random_state_report(search_cost_component_type) empty in the same cases; the solution may change if a move is cleared; if result is a type, error component must be a value, description path is one #<position>, its own value without weights and its
Costs	Cost, a cost::value as text, apply with a cost::Percent cost the front of the last run, a vector of pareto_point(solution, cost) and a random_state_report(search_cost_component_type) empty in the same cases; the solution may change if a move is cleared; if result is a type, error component must be a value, description path is one #<position>, its own value without weights and its

A command the problem cannot serve does not compile; a frontend tests it first with the static constexpr bool capability named after it: supports_initial_solution, supports_random_solution, supports_read_input (also load_input), supports_read_solution (also load_solution), supports_write_solution (also save_solution), supports_check_neighborhood_costs, supports_check_move_independence and supports_check_random_move_distribution. Three name a group: supports_deterministic_moves (use_first_move, use_next_move, neighborhood_preview), supports_random_moves (use_random_move) and supports_improving_moves (use_first_improving_move, use_best_move, neighborhood_statistics). The result of a check is named after it too: check_neighborhood_costs_result, check_move_independence_result, check_random_move_distribution_result.

The commands move the session through these states; a new Input drops the solution and the move, a new solution or a run drops the move:



The selections and run return false when there is nothing to select or no runner with that name, and are [[nodiscard]]. set_solution and load_solution accept a solution that is not valid for the Input, to inspect it; run does not start from one: it throws std::invalid_argument and changes nothing. run runs on the session's bound app, with the current parameters: when they have changed through app() since it was bound, it is bound again first; it executes on the calling thread. The REST service gives every run

a Session of its own and calls `run` on a worker thread, with `with(control)` for progress and cancellation. The TextUI, which keeps reading its Session while a runner works, runs it with `app.run("name", ...)` on copies of the app, Input and solution instead. Its Move-page scans run likewise on a worker, on a Session of their own with the current solution, and stop through their `std::stop_token` when the page is left, a move applied or another scan started.

Design choices

- **Fresh services per run.** `app.run` binds new services, so concurrent runs share only the immutable Input and no locking is needed.
- **Registration by class and name.** The class is the key, the name tells registrations of the same class apart and is what tools show.
- **One way to run by name.** `app.run("name", ...)` is the single place where a name selects a runner; the TextUI, the REST service and Session use it.
- **Adapters are optional components** outside Core, meant to become separate projects once the API is stable.

Testing

<easylocal/testing.hpp> (easylocal::testing). The Core umbrella brings only what check(app) uses (check_options, the reports, run_checks), not the component checks.

Contract checks for user components. Each check takes a fixture and returns a check_report; run_checks(reports...) prints them, the report of check(app) among them, and returns a process exit code.

```
const easylocal::testing::fixture<TourManager> f{input, solution};
return easylocal::testing::run_checks(
    easylocal::testing::check_solution_manager(f),
    easylocal::testing::check_neighborhood<TwoOptExplorer>(f),
    easylocal::testing::check_delta_cost_component<TwoOptExplorer, TourLength,
    TwoOptDelta>(f));
```

Fixture

fixture<SM, Equivalent = approximately> owns an Input, a Solution and the SolutionManager SM{input}; it is neither copyable nor movable.

Constructor	Meaning
fixture{input, solution[, options]}	the given Solution (checked for validity)
fixture{input[, options]}	the SolutionManager's initial_solution()

check_options holds random_samples (default 32), max_enumerated_moves (default 1024), random_solutions (default 4), walk_length (default 8), seed and tolerance: the randomized checks draw from a std::mt19937_64 seeded with seed, so the same seed repeats the same draws. The move checks (check_neighborhood, check_delta_cost_component) start from the fixture's Solution and from random_solutions others, each a random_solution() (the fixture's when the SolutionManager draws none) walked by walk_length random moves: a delta right on one solution only, such as one that mixes up positions and cities and is right on the identity tour, fails on them. From each, they visit max_enumerated_moves enumerated moves (all of them when there are no more, otherwise a uniform sample) and random_samples random ones. deterministic_rng, which cycles through a few fixed values, is for unit tests that script the draws, not for sampling.

Equivalent compares values (component values, value + delta against the full evaluation). The default, testing::approximately (the type cost::tolerance), uses the options' tolerance: two floating-point values are equal when they differ by at most max(absolute, relative * max(|a|, |b|)), both 1e-9 by default, integers compare exactly, lexicographic, hierarchical and pareto costs level by level, other values with ==. A cost updated by deltas accumulates rounding errors that a full evaluation does not have (on a TSP with decimal distances, value + delta differs from the new length in the last bits): with an exact comparison the delta check would report them. {.tolerance = {.relative = 0, .absolute = 0}} compares exactly, as does fixture<SM, std::equal_to<>>; another Equivalent replaces the comparison.

Checks

Check	Verifies
<code>check_solution_manager(f)</code>	input binding; the fixture, initial and random solutions are valid
<code>check_cost_component<C>(f)</code>	evaluating the same solution twice gives equivalent values (values the comparison cannot compare do not compile)
<code>check_neighborhood<N>(f)</code>	enumerated and sampled moves are valid; applying them keeps the solution valid; with solutions that compare, not every move leaves the solution unchanged (null moves, a <code>make_move</code> taking the Solution by value); with moves that compare and a neighborhood that enumerates and draws, random moves are enumerated ones, found whenever a valid move exists, and drawn again by a generator with the same seed
<code>check_delta_cost_component<N, C, D>(f)</code>	for each valid enumerated (or, failing that, sampled) move, <code>value + delta</code> equals the component value after the move
<code>check_delta_cost_component<N, C>(f)</code>	the same, with the co-located <code>C::delta_evaluate</code>

A failure names the move (with its describe hook or operator<<), the solution it starts from and the values that disagree. A hook that throws (`evaluate`, `make_move`, `random_move`...) fails the check, with what the exception says, instead of leaving it. `print_report` groups the failures of a check: their number and the first three.

The type form builds the object under test as an app does: components and delta cost components from the Input (or default-constructed), NeighborhoodExplorers from the fixture's SolutionManager. For other constructors pass the objects instead: `check_cost_component(f, c)`, `check_neighborhood(f, n)`, `check_delta_cost_component(f, n, c[, d])`.

For a composed problem, `easylocal::check(app, input[, solution][, options])` (<easylocal/app/check.hpp>) runs the same kinds of checks on an app, with the options; its report, an `app_check_report`, goes to `run_checks` with the others.

Design choices

- **Checks are contracts, not unit tests.** They verify the laws the framework relies on (validity preservation, the delta law), so a component that passes them can be used by every algorithm.
- **Deterministic.** Randomized checks draw from a generator seeded by the options: the same seed repeats the same draws.
- **A tolerance in the checks, exact relations in the search.** Rounding errors are not bugs, so the checks forgive them; the search compares costs exactly unless the cost expression says otherwise (`cost::approximately`, see [Cost](#)).
- **One fixture, many checks.** The data is written once; each check names only the types under test.