

EasyLocal – Tutorial

A C++23 framework for local search and metaheuristics

Contents

Tutorial	5
The running example	5
The workflow	6
Conventions of the code	6
Chapters	6
1. Modelling the problem	8
Representing tours	8
Input, Solution and Move	9
The SolutionManager	9
See also	10
Next steps	10
2. The cost	11
Cost components	11
Cost expressions	11
See also	12
Next steps	12
3. Structured costs	13
Choosing an expression	13
Costs compared by priority	13
Domain values	14
Cost semantics	16
See also	16
Next steps	16
4. Moves	17
Only what your algorithms use	17
Swap moves	17
2-opt moves	18
See also	21
Next steps	21
5. Delta evaluation	22
The delta cost component	22
Choice: separate or co-located	23
Checking the deltas during a run	24
See also	24
Next steps	24
6. Running a search	25
A runner	25
Built-in algorithms	25
Results	27
Reading the instance, printing the solution	27
See also	29
Next steps	29
7. Combining neighborhoods	30
See also	30
Next steps	30
8. Writing your own runner	31
See also	32
Next steps	33
9. Solvers	34

Hard and soft costs	34
Several stages	35
See also	36
Next steps	36
10. Configuration	37
Configuration files	41
TOML files	41
See also	42
Next steps	42
11. Testing your components	43
See also	44
Next steps	44
12. Applications	45
The app	45
The Session	45
A command-line program	47
See also	49
Next steps	49
13. Tuning with irace	50
What is tuned	50
The scenario	50
Running irace	51
The cost as one number	52
See also	52
Next steps	52
14. The interactive tester	53
A tour of the pages	53
Loading, saving and displaying	56
Several apps on the same problem	56
Options	57
See also	58
Next steps	58
15. Checking a composed problem	59
check: the app's contracts	59
Session checks: the neighborhood of a solution	60
See also	62
Next steps	62
16. A REST service	63
Without a codec	63
The codec	63
The service	65
See also	66
Next steps	66
17. Observing and controlling a run	67
See also	68
Next steps	68
18. Building without CMake	69
Before you start	69
One command	69
A makefile	69
The optional components	71

What CMake does that a makefile does not	72
19. Advanced: customizing Simulated Annealing	73
Inside the search loop	73
First, use the built-in reheating	75
A different rule: reheat after rejected moves	75
Run and configure the variant	78
When to change the loop itself	79
See also	79

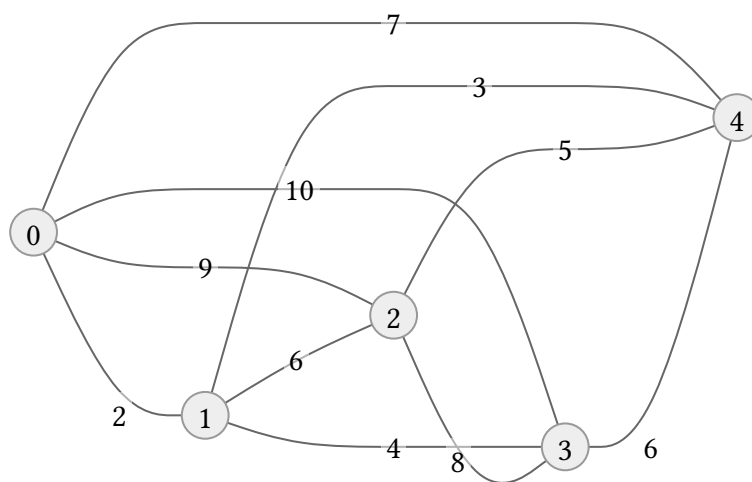
Tutorial

Build a local search solver for the symmetric Travelling Salesperson Problem (TSP), then give it configuration, tests and user interfaces. Each chapter adds one capability to the same program, reusing the components you have already written. Links to the [reference](#) provide the full API details when you need them.

If you have not done so yet, start with the [quick start](#): it is the complete program of the first chapters, in one file.

The running example

The TSP asks for the shortest round trip that visits every city exactly once. In the symmetric version, distances are the same in both directions. We use five cities with these distances:



In the code the instance is a distance matrix, one row per city:

```
inline Tsp five_cities()
{
    return Tsp{
        .distance =
        {
            {0, 2, 9, 10, 7},
            {2, 0, 6, 4, 3},
            {9, 6, 0, 8, 5},
            {10, 4, 8, 0, 6},
            {7, 3, 5, 6, 0},
        },
    };
}
```

The shortest round trip has length 26, reached by two tours (each also in the opposite direction): $0 \rightarrow 1 \rightarrow 3 \rightarrow 2 \rightarrow 4 \rightarrow 0$ ($2 + 4 + 8 + 5 + 7$) and $0 \rightarrow 1 \rightarrow 3 \rightarrow 4 \rightarrow 2 \rightarrow 0$ ($2 + 4 + 6 + 5 + 9$).

The chapters build the solver in this order:

- chapters 1 and 2 model a tour as a sequence of cities and give it a cost;
- chapter 4 defines swaps and 2-opt moves, which reverse a segment of the tour, and shows how to enumerate them with a generator or a cursor;
- chapter 5 computes the effect of a 2-opt move on the length from four distances only;

- chapter 6 runs both First Improvement and Simulated Annealing, and chapter 7 lets a search use both moves;
- the remaining chapters build on these pieces: new algorithms, solvers, configuration, testing and the interactive and HTTP tools.

The complete code lives in `examples/tutorial/`: `tsp.hpp` defines the problem, `main.cpp` runs the searches and `checks.cpp` tests the components. The examples are built and tested with the library, and the snippets on these pages are checked against their sources. You can follow along in the files or use them as a starting point for your own problem.

The workflow

Every EasyLocal program follows the same three steps:

model	values	Input (immutable), Solution, Move
	services	SolutionManager valid solutions, construction
	cost components	one term of the objective each
	cost expression	the cost from the component values
	NeighborhoodExplorer	moves: enumeration, sampling, application
	delta cost components	the change of a component under a move
compose	a runner	an algorithm plus the recipes of the services (and, optionally, a solver or an app around it)
run		bind the runner to an Input, run it from a solution, read the result

The SolutionManager, cost components and cost expression form the *cost layer*. The NeighborhoodExplorer and its deltas form the *delta cost layer*. Recipes describe how to assemble these services; binding a runner to an Input constructs them. The services borrow the Input by `const&` and leave it unchanged.

Start with the members your search needs. Later chapters add features for solvers, checks and interactive tools as they become useful. Missing capabilities are checked at compile time, so you can develop the problem incrementally without implementing an entire interface up front.

Conventions of the code

The library uses namespace `easylocal`. The examples declare two aliases at the start of each program to keep the snippets short:

```
using namespace tutorial;           // the tutorial's types: Tsp, Tour, ...
namespace el = easylocal;           // el::app, el::component, ...
namespace runners = easylocal::runners; // runners::FirstImprovement, ...
```

Thus `el::app` means `easylocal::app`. Put aliases in a function or source file to avoid introducing them into files that include your headers. Unlike `using namespace easylocal;`, an alias keeps the library's names identifiable.

The first line brings in the tutorial's own types, from `tsp.hpp`, so the snippets write `Tour` for `tutorial::Tour`.

Chapters

1. [Modelling the problem](#): Input, Solution, Move and the SolutionManager.
2. [The cost](#): cost components and cost expressions.
3. [Structured costs](#): priorities, objectives, domain values and cost semantics.
4. [Moves](#): the NeighborhoodExplorer.
5. [Delta evaluation](#): evaluating moves incrementally.

6. [Running a search](#): runners, built-in algorithms, results, reading the instance and printing the solution.
7. [Combining neighborhoods](#): neighborhood_union.
8. [Writing your own runner](#): search_run.
9. [Solvers](#): from an Input to a final solution.
10. [Configuration](#): parameters from the command line and files.
11. [Testing your components](#): contract checks.
12. [Applications](#): the app, a problem and its runners, the Session that runs it on an Input, and a command-line program.
13. [Tuning with irace](#): the parameters of a program, tuned on its instances.
14. [The interactive tester](#): the TextUI.
15. [Checking a composed problem](#): check, and the neighborhood checks of a Session.
16. [A REST service](#): searches over HTTP.
17. [Observing and controlling a run](#): progress, cancellation, tracing.
18. [Building without CMake](#): a makefile, and the headers and libraries of every component.
19. [Advanced: customizing Simulated Annealing](#): the library's search loop, built-in reheating and a custom temperature policy.

Porting an EasyLocal 3 program? [Coming from EasyLocal 3](#) walks through the migration step by step.

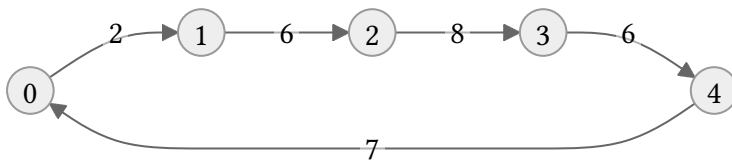
1. Modelling the problem

Start by describing the data the search will use: an Input, a Solution and a Move. A SolutionManager will create solutions and check their validity.

Representing tours

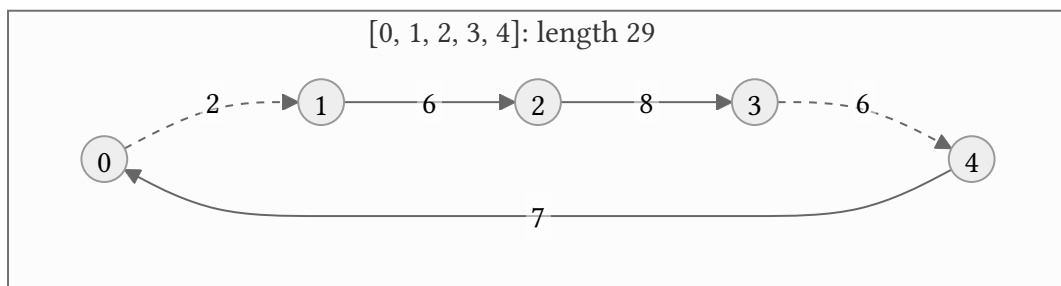
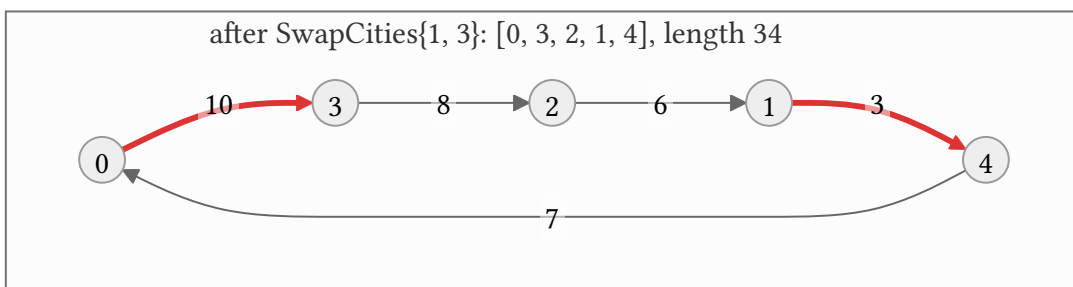
Before writing code, decide what a solution looks like and how the search changes it.

A solution is a tour, stored as the sequence of the cities in the order they are visited. $[0, 1, 2, 3, 4]$ is the tour $0 \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 0$: the return to the first city is implicit. Its length is $2 + 6 + 8 + 6 + 7 = 29$:



Not every sequence is a tour. A tour of n cities is a *permutation* of $0, 1, \dots, n - 1$: n positions, every city exactly once. $[0, 1, 1, 3, 4]$ visits city 1 twice and city 2 never; $[0, 1, 2]$ forgets two cities.

A move changes a tour. We start with `SwapCities{i, j}`, which exchanges the cities at positions i and j ; chapter 4 adds a second kind of move. On $[0, 1, 2, 3, 4]$, `SwapCities{1, 3}` gives $[0, 3, 2, 1, 4]$: the edges $0-1$ and $3-4$ (dashed) leave the tour, the edges $0-3$ and $1-4$ (red) enter it, and the length goes from 29 to 34.



The edges $1-2$ and $2-3$ stay, traversed in the other direction. In general a swap replaces the two edges around each of the two positions, four edges in all; when the positions are next to each other, only two. A swap always turns a permutation into a permutation, so it never breaks a tour.

Input, Solution and Move

In code, the three are plain types:

```
// Input: the instance, immutable during the search.
struct Tsp
{
    // distance[a][b] is the distance between cities a and b (symmetric).
    std::vector<std::vector<double>> distance;

    std::size_t cities() const
    {
        return distance.size();
    }
};

// Solution: order[k] is the k-th city visited; after the last city the tour
// returns to order[0].
struct Tour
{
    std::vector<std::size_t> order;
};

// Move: exchange the cities visited at positions i and j, with i < j.
struct SwapCities
{
    std::size_t i;
    std::size_t j;
};
```

- The **Input** is built once, for example by your parser, and treated as immutable while services are bound to it. The distance matrix is a vector of rows: the distance between a and b is distance[a][b].
- A **Solution** does not own or reference the Input.
- A **Move** does not own or reference a Solution: the NeighborhoodExplorer applies it to one (chapter 4).

None of them derives from a framework class or needs any operator: plain structs are enough.

The SolutionManager

The SolutionManager defines which solutions are valid and how to build them:

```
class TourManager : public easylocal::solution_manager_base<Tsp, Tour>
{
public:
    // The constructor of the base: from the Input, which input() gives back.
    using solution_manager_base::solution_manager_base;

    // The cities in index order: 0, 1, ..., n - 1.
    Tour initial_solution() const
    {
        Tour tour{std::vector<std::size_t>(input().cities())};
        std::ranges::iota(tour.order, std::size_t{0});
        return tour;
    }

    // A tour is valid when it is a permutation of 0, 1, ..., n - 1: it has
    // n positions and visits every city exactly once.
    bool is_valid(const Tour& tour) const
    {
        return std::ranges::is_permutation(
```

```

        tour.order,
        std::views::iota(std::size_t{0}, input().cities()));
    }
};

```

- `solution_manager_base<Input, Solution>` supplies the type aliases, constructor and `input()` accessor. This optional base has no virtual functions; a class providing the same members works too.
- `is_valid` checks the representation. Here, `std::ranges::is_permutation` verifies that `tour.order` contains exactly the cities 0 through `n - 1`. `std::views::iota` supplies that sequence without storing it. The check can take quadratic time; for large tours, marking visited cities in a `std::vector<bool>` gives a linear check.
- `initial_solution()` creates a starting tour. It is optional when you supply your own solution, but lets runners, solvers and the tester create one for you. Chapter 9 adds random construction.

Validity here means that the representation is well formed. A tour may still violate problem constraints; those violations become part of the cost in chapter 2.

Add capabilities as you need them. This is enough to start a search. Later we add I/O hooks for files (chapter 6), `random_solution` for random starts (chapter 9), and equality for neighborhood checks (chapter 15). Missing capabilities produce a compile error when required; the interactive tester simply omits commands that the problem does not support.

The `SolutionManager` never computes the cost: the cost always comes from cost components, the subject of the next chapter.

See also

- [Problem model](#): optional I/O and display hooks for `Input`, `Solution` and `Move`.
- [SolutionManager](#): the full contract.

Next steps

[Chapter 2](#) gives the tour a cost.

2. The cost

The search needs a way to compare tours. We start with tour length, then combine it with other measures without changing the solution or move code.

Cost components

Costs are minimized by default. To maximize a profit, use `cost::apply` to return its negation or the lost profit. [Cost semantics](#) in the next chapter also explains how to define a custom comparison.

A **cost component** computes one term of the objective:

```
class TourLength : public easylocal::input_base<Tsp>
{
public:
    using input_base::input_base; // constructed from the Input, read by input()

    double evaluate(const Tour& tour) const
    {
        const auto n = tour.order.size();
        double length = 0.0;
        for (std::size_t k = 0; k < n; ++k)
        {
            const auto from = tour.order[k];
            const auto to =
                tour.order[(k + 1) % n]; // the last city goes back to the first
            length += input().distance[from][to];
        }
        return length;
    }
};
```

- The only requirement is `evaluate(const Solution&) const -> Value`.
- Value is usually a number, as here. It may also be a struct of your own, a *domain value*: an advanced use, described in [Domain values](#).
- `input_base<Tsp>` supplies `input()` and the constructor inherited by `using input_base::input_base;`. The `SolutionManager` in [chapter 1](#) uses the same base. You can also provide `input()` yourself or omit it if the component needs no `Input` (see the [convenience bases](#)).
- Binding the runner constructs the component as `Component{const Input&, args...}`, or `Component{args...}` for a component that needs no `Input`. Precompute instance data in this constructor so that `evaluate` only needs the solution. For example: `C(const Tsp& input) : input_base{input}, table_{build(input)} {}`.
- `(k + 1) % n` makes the last city connect back to the first one.

Components are attached to the `SolutionManager` with a **recipe** (`el` is the alias of the namespace `easylocal`, see the [conventions](#)):

```
auto sm = el::solution_manager<TourManager>() | el::component<TourLength>();
```

This recipe describes a `TourManager` with `TourLength` as its cost. Binding it to an `Input` will construct the services. The named spelling is `solution_manager<TourManager>().with_cost(component<TourLength>())`.

Cost expressions

With several components, a **cost expression** combines their values. For example, this weighted sum balances tour length against the longest edge:

```
auto weighted_sm = el::solution_manager<TourManager>()
| el::cost::sum(el::component<TourLength>(), el::component<MaxEdge>() * 10.0);
```

MaxEdge is a second component, the longest edge of the tour:

```
class MaxEdge : public easylocal::input_base<Tsp>
{
public:
    using input_base::input_base;

    double evaluate(const Tour& tour) const
    {
        const auto n = tour.order.size();
        double longest = 0.0;
        for (std::size_t k = 0; k < n; ++k)
        {
            const auto from = tour.order[k];
            const auto to = tour.order[(k + 1) % n];
            longest = std::max(longest, input().distance[from][to]);
        }
        return longest;
    }
};
```

- `child * w` is shorthand for `cost::weighted(child, w)`; either form marks a term of a `cost::sum`, and only there. The weights are parameters, here `cost.weights` (see [Configuration](#)).
- `cost::sum` adds numbers, and `cost::apply(f, c1, ..., cn)` computes anything else from the values of its children: `cost::apply([](double longest) { return std::max(0.0, longest - 8.0); }, component<MaxEdge>())` is the excess of the longest edge over 8.
- Expressions can contain components or other expressions. This keeps the cost structure and weights together, so you can change the objective while reusing the components.

A single number does not always say which of two solutions is better: a constraint may have to come before the objective, or two objectives may have no common unit. [Chapter 3](#) builds those costs, and the rest of the tutorial uses the sum of this chapter.

See also

- [Cost](#): cost components, cost expressions, cost models and semantics.

Next steps

[Chapter 3](#) compares solutions by priority, by objective, or by a rule of your own.

3. Structured costs

The sum of [chapter 2](#) weighs every term against the others: a longer tour is acceptable when some other term improves enough. Some problems do not work that way. A constraint must be satisfied before the objective counts; two objectives have no exchange rate; a component measures several numbers at once. This chapter builds those costs, from the same components.

Choosing an expression

Choose an expression that matches how you want to compare solutions:

Expression	Cost
<code>cost::sum(t1, ..., tn)</code>	$\sum w_i \cdot \text{cost}_i$ over arithmetic costs; child * w gives a term its weight (default 1)
<code>cost::in_order(c1, ..., cn)</code>	a <code>cost::lexicographic</code> cost: compared child by child
<code>cost::hard_soft(hard, soft)</code>	a <code>cost::hierarchical</code> cost: hard has strict priority over soft
<code>cost::objectives(c1, ..., cn)</code>	a <code>cost::pareto</code> cost: one value per objective, compared by Pareto dominance
<code>cost::apply(f, c1, ..., cn)</code>	$f(\text{cost}_1, \dots, \text{cost}_n)$, for anything else

An expression nests: a branch of one is a component or another expression, so the structure and the weights stay together. A hard/soft model is:

```
auto sm = el::solution_manager<TimetableManager>()
| el::cost::hard_soft(
    el::cost::sum(el::component<Conflicts>(),
                  el::component<Unavailability>()),
    el::cost::sum(el::component<Compactness>() * 2,
                  el::component<Preferences>() * 5));
```

- Each branch has its own weights, named after its path: here `cost.hard.weights` and `cost.soft.weights` (see [Configuration](#)).
- `cost::sum` adds numbers. A component with a domain value is turned into one with `cost::apply`, for example `cost::apply([], (const Load& l) { return l.overload; }, component<Capacity>());` [Domain values](#) below is about those components.
- `solvers::two_stage(first, second)` (see [Solvers](#)) evaluates only the components of the hard branch in its first stage.

Costs compared by priority

`cost::in_order` and `cost::hard_soft` build the two types of `<easylocal/cost.hpp>` that compare values by priority:

- `cost::lexicographic<Ts...>` compares its values in order: the second one matters only between costs whose first one is equal, and so on;
- `cost::hierarchical<Hard, Soft>` gives the hard branch strict priority over the soft one: no improvement of the soft cost can make up for a worse hard cost. Each branch is compared with its operator `<=`, or else with `<` and `=` alone; the hierarchical cost then has all the comparisons, `<=` included.

For separate objectives, `cost::objectives` builds a third type, `cost::pareto`, compared by Pareto dominance. This supports searches that return a front of non-dominated solutions (see [Cost](#)).

You rarely need to spell the types: `cost::in_order` and `cost::hard_soft` deduce them from the components. Here are two ways to combine `MaxEdge` and `TourLength`:

```
// Lexicographic: the longest edge first; between tours with the same
// longest edge, the shorter one.
auto bottleneck_sm = el::solution_manager<TourManager>()
    | el::cost::in_order(el::component<MaxEdge>(), el::component<TourLength>());

// Hierarchical: edges longer than 8 are a violation, measured by how much
// the longest edge exceeds 8 (hard); the tour length is the objective (soft).
auto bounded_sm = el::solution_manager<TourManager>()
    | el::cost::hard_soft(
        el::cost::apply(
            [](double longest) { return std::max(0.0, longest - 8.0); },
            el::component<MaxEdge>()),
        el::component<TourLength>());
```

- `bottleneck_sm` has a `cost::lexicographic<double, double>` cost: a tour is better if its longest edge is shorter, and between two tours with the same longest edge, the shorter tour is better. The order of the children is the order of comparison.
- `bounded_sm` treats edges longer than 8 as a constraint. Its hard cost measures the violation: `cost::apply` turns the value of `MaxEdge` into the excess over 8, zero when every edge is short enough. Its soft cost is the length. The result is a `cost::hierarchical<double, double>`: a tour with a smaller excess is always better, whatever its length. The bound is a literal here; [chapter 10](#) makes it a parameter.

Both recipes work with the runner from chapter 6 and the 2-opt neighborhood from chapters 4 and 5. Read a lexicographic result by position and a hierarchical result by branch:

```
auto bottleneck =
    (el::make_runner<runners::FirstImprovement>(runners::FirstImprovementParameters{}))
    | bottleneck_sm | nhe)
    .bind(tsp);
const auto by_edge = bottleneck.run(bottleneck.initial_solution());
// A lexicographic cost is read by position, a hierarchical one by branch.
const double longest_edge = by_edge.cost.get<0>();
const double length_after_edge = by_edge.cost.get<1>();

auto bounded =
    (el::make_runner<runners::FirstImprovement>(runners::FirstImprovementParameters{}))
    | bounded_sm | nhe)
    .bind(tsp);
const auto within_bound = bounded.run(bounded.initial_solution());
const double excess = within_bound.cost.hard();
const double length_within_bound = within_bound.cost.soft();
```

On the five-city instance, both find a tour of length 26 with a longest edge of 8. Here the shortest tour also minimizes the longest edge, so the constraint does not increase the length. For a nested example, see `examples/assignment/cost.hpp`: its `cost::hard_soft` has a lexicographic hard branch computed with `cost::apply`.

Domain values

Sometimes a component measures more than one number at once. `LongEdges` counts the edges longer than 6 and adds up how much they exceed 6, in a single pass over the tour:

```

// A domain value: the edges longer than 6, as a total excess and a count.
struct LongEdges
{
    double excess{}; // the total length beyond 6 of the long edges
    std::size_t count{}; // how many edges are longer than 6

    // Needed only when LongEdges is itself the cost: tours are then compared
    // by excess first and by count between equal excesses.
    auto operator<=>(const LongEdges&) const = default;
};

class LongEdgesComponent : public easylocal::input_base<Tsp>
{
public:
    using input_base::input_base;

    LongEdges evaluate(const Tour& tour) const
    {
        const auto n = tour.order.size();
        LongEdges value;
        for (std::size_t k = 0; k < n; ++k)
        {
            const auto edge = input().distance[tour.order[k]][tour.order[(k + 1) % n]];
            if (edge > 6.0)
            {
                value.excess += edge - 6.0;
                ++value.count;
            }
        }
        return value;
    }
};

```

The component returns the struct, and the framework stores it like any other value. What the struct needs depends on how the cost expression uses it:

```

// The struct is the cost: it is compared with its operator<=>.
auto long_edges_sm =
    el::solution_manager<TourManager>() | el::component<LongEdgesComponent>();

// A function turns it into a number: here the excess is a hard cost.
auto excess_sm = el::solution_manager<TourManager>()
    | el::cost::hard_soft(
        el::cost::apply(
            [](const LongEdges& value) { return value.excess; },
            el::component<LongEdgesComponent>()),
        el::component<TourLength>());

```

- **The struct is the cost** (`long_edges_sm`). Algorithms compare costs, so the struct must be ordered: better, equivalent and better_or_equivalent default to `<`, `==` and `<=` (see [Cost semantics](#) below). The defaulted operator<=> provides them all, comparing the members in declaration order: first excess, then count. Without it, the runner does not compile.
- **A function turns it into a number** (`excess_sm`). `cost::apply` maps the struct to its excess, a number that is then the hard cost. Here the struct needs no operators: the algorithms compare the numbers `cost::apply` returns. `cost::apply` is also how a domain value enters a `cost::sum`, which adds numbers only.

- **With a delta cost component** (chapter 5), the delta is added to the value, so the struct also needs `operator+(Value, Delta)`.

With First Improvement and the 2-opt moves, `long_edges_sm` stops at an excess of 3 over two edges. That is a local optimum: no 2-opt move improves it, although a tour with the same excess on a single edge exists. `excess_sm` finds the minimum excess, 3, and then the shortest tour with it, of length 26.

Cost semantics

Algorithms compare costs through `better`, `equivalent` and `better_or_equivalent`. These use the cost type's `<`, `==` and `<=` by default. To customize all three, give the function at the root `cost::apply` a `compare(a, b)` member returning `std::partial_ordering`; less means that `a` is better.

Simulated Annealing also reads `cost::delta` and treats a negative delta as an improvement. Negating a profit is therefore the simplest way to maximize it. For floating-point costs, wrap the expression in `cost::approximately(...)` to treat differences within a tolerance as equivalent.

See also

- [Cost](#): cost components, cost expressions, cost models and semantics.

Next steps

[Chapter 4](#) defines the moves.

4. Moves

The **NeighborhoodExplorer** defines the moves available from a solution, checks their validity and applies them. Keeping this logic in one component lets different algorithms search the same neighborhood.

Only what your algorithms use

Every explorer needs `is_valid(solution, move)` and `make_move(solution, move)`. The optional `neighborhood_explorer_base` supplies its type aliases and Input access from the `SolutionManager`. Add other members as your algorithms and tools need them:

Member	Needed by
<code>input_type, solution_type, move_type, input()</code>	every algorithm (from <code>neighborhood_explorer_base</code>)
<code>is_valid, make_move</code>	every algorithm
<code>moves(solution)</code> , or the cursor <code>first_move / next_move</code>	algorithms that scan the neighborhood: First Improvement, Best Improvement, Tabu Search
<code>random_move(solution, rng)</code>	algorithms that sample it: Simulated Annealing, Hill Climbing, Late Acceptance Hill Climbing, Great Deluge, Pareto Late Acceptance Hill Climbing
<code>inverse(solution, move, tabu_move)</code>	Tabu Search: whether a move is forbidden by one applied earlier
<code>name()</code>	the interactive tester (chapter 14)

For example, Simulated Annealing needs random sampling but no enumeration. The quick start uses only First Improvement, so its explorer has no `random_move`. If an algorithm needs a missing member, the compiler names it.

Unlike EasyLocal 3's virtual interface, this lets you implement just the operations your program uses.

The explorers of this tutorial implement both enumeration and sampling, because chapter 6 runs them with First Improvement and with Simulated Annealing.

Swap moves

The *neighborhood* of a tour is the set of moves that can be applied to it. For `SwapCities` it has one move for every pair of positions $i < j$: $n(n - 1)/2$ moves, 10 for five cities.

```
class SwapExplorer : public easylocal::neighborhood_explorer_base<TourManager,
SwapCities>
{
public:
    // The constructor of the base: from the SolutionManager, whose Input the
    // explorer reaches with input().
    using neighborhood_explorer_base::neighborhood_explorer_base;

    // Every pair of positions i < j, one move at a time.
    easylocal::generator<SwapCities> moves(const Tour& tour) const
    {
        const auto n = tour.order.size();
        for (std::size_t i = 0; i < n; ++i)
```

```

        for (std::size_t j = i + 1; j < n; ++j)
            co_yield SwapCities{i, j};
    }

    // Uniform: two distinct positions, each pair equally likely, put in order.
    template<std::uniform_random_bit_generator RNG>
    std::optional<SwapCities> random_move(const Tour& tour, RNG& rng) const
    {
        const auto n = tour.order.size();
        if (n < 2)
            return std::nullopt;
        std::uniform_int_distribution<std::size_t> pick{0, n - 1};
        const auto i = pick(rng);
        auto j = pick(rng);
        while (j == i)
            j = pick(rng);
        return SwapCities{std::min(i, j), std::max(i, j)};
    }

    bool is_valid(const Tour& tour, const SwapCities& move) const
    {
        return move.i < move.j && move.j < tour.order.size();
    }

    void make_move(Tour& tour, const SwapCities& move) const
    {
        std::swap(tour.order[move.i], tour.order[move.j]);
    }
};

```

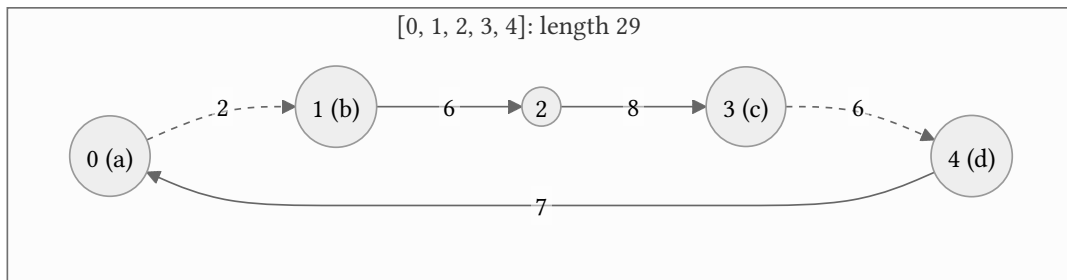
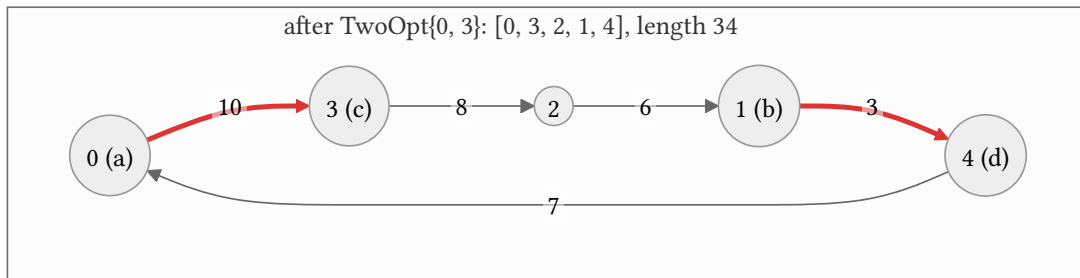
- `is_valid` checks that the positions are distinct, ordered and within an already valid tour. `make_move` applies the move in place with `std::swap`.
- `moves(solution)` enumerates moves for algorithms such as First Improvement. Each `co_yield` returns one move and pauses until the runner requests another. This avoids building a whole container when the search may stop after the first improvement. Other input ranges also work if their elements convert to the move type.
- `random_move(solution, rng)` samples a move for algorithms such as Simulated Annealing. Here it draws two distinct positions and orders them, giving every pair equal probability. Uniform sampling is optional, but sampling should be cheap: avoid enumerating the neighborhood for each proposal.
- A tour with fewer than two cities has no swap moves. The return type, `std::optional<SwapCities>`, can hold either a move or no value. Return `SwapCities{...}` on success and `std::nullopt` for an empty neighborhood. The caller checks `if (move)` and reads the value with `*move`.

`easylocal::neighborhood_explorer_base<SolutionManager, Move>` provides the aliases and the Solution-Manager reference; like the `SolutionManager` base, it is optional and non-virtual.

Attach the explorer with a recipe, `neighborhood<SwapExplorer>()`. In debug builds the framework asserts that the solution and the move are valid before every application, and that the solution is valid afterwards.

2-opt moves

`TwoOpt{i, j}` removes two edges of the tour: the one leaving position `i`, from city `a` to city `b`, and the one leaving position `j`, from `c` to `d`. It reconnects the tour with the edges `a → c` and `b → d`, which means traversing the part from `b` to `c` backwards: the cities at positions `i + 1` to `j` are reversed. On `[0, 1, 2, 3, 4]`, `TwoOpt{0, 3}` reverses positions 1 to 3:



The edges $a \rightarrow b$ and $c \rightarrow d$ (dashed) leave the tour, $a \rightarrow c$ and $b \rightarrow d$ (red) enter it. The edges in between stay; in a symmetric TSP, traversing them backwards does not change their length.

```
// Move: reverse the part of the tour between positions i + 1 and j.
struct TwoOpt
{
    std::size_t i;
    std::size_t j;
};

class TwoOptExplorer : public easylocal::neighborhood_explorer_base<TourManager,
TwoOpt>
{
public:
    // The constructor of the base: from the SolutionManager, whose Input the
    // explorer reaches with input().
    using neighborhood_explorer_base::neighborhood_explorer_base;

    // The moves are the pairs i + 2 <= j < n, by i and then by j: with
    // j = i + 1 the segment would be one city, and with i = 0, j = n - 1 the
    // two removed edges would be the same one, so that pair is skipped.
    // A cursor enumerates them in place: first_move writes the first move into
    // `move`, next_move turns `move` into the following one, and both return
    // false when there is none.
    bool first_move(const Tour& tour, TwoOpt& move) const
    {
        move = TwoOpt{0, 1}; // just before the first move, TwoOpt{0, 2}
        return next_move(tour, move);
    }

    bool next_move(const Tour& tour, TwoOpt& move) const
    {
        const auto n = tour.order.size();
        do
        {
            if (++move.j == n) // the last j for this i: on to the next i

```

```

        {
            ++move.i;
            move.j = move.i + 2;
        }
        if (move.j >= n) // no i left
            return false;
    }
    while (move.i == 0 && move.j + 1 == n);
    return true;
}

// Uniform by rejection: two positions drawn independently, ordered, and
// drawn again while they are not a 2-opt move.
template<std::uniform_random_bit_generator RNG>
std::optional<TwoOpt> random_move(const Tour& tour, RNG& rng) const
{
    const auto n = tour.order.size();
    if (n < 4)
        return std::nullopt;
    std::uniform_int_distribution<std::size_t> pick{0, n - 1};
    while (true)
    {
        auto i = pick(rng);
        auto j = pick(rng);
        if (j < i)
            std::swap(i, j);
        if (i + 2 <= j && !(i == 0 && j + 1 == n))
            return TwoOpt{i, j};
    }
}

// The moves the cursor lists: (0, n - 1) would only reverse the tour,
// preserving the same edges in the symmetric TSP.
bool is_valid(const Tour& tour, const TwoOpt& move) const
{
    const auto n = tour.order.size();
    return move.i + 2 <= move.j && move.j < n && !(move.i == 0 && move.j + 1 == n);
}

// The segment is the j - i cities from position i + 1: a span views it in
// place, and reversing the view reverses those cities in the tour.
void make_move(Tour& tour, const TwoOpt& move) const
{
    std::ranges::reverse(std::span{tour.order}.subspan(move.i + 1, move.j -
move.i));
}
};

```

- The moves are the pairs with $i + 2 \leq j$: with $j = i + 1$ the reversed segment would be a single city. The pair $i = 0, j = n - 1$ is skipped too: the two edges meet at the first city, and reversing the remaining cities would only traverse the same symmetric tour in the opposite direction.
- This explorer uses a **cursor**. `first_move(tour, move)` fills in the first move; `next_move(tour, move)` advances to the next. Both return false when no move is available. Here `first_move` starts from `TwoOpt{0, 1}`, just before the first valid move, and delegates the advance to `next_move`.
- `make_move` reverses the segment in place. `std::span{tour.order}` is a view of the vector, and `subspan(offset, count)` narrows it to the $j - i$ cities from position $i + 1$; `std::ranges::reverse` then reverses them in the vector itself.

- `random_move` draws two positions and draws again until they form a 2-opt move: uniform over the moves, and constant expected time, since most draws are valid.

Generator or cursor

The two explorers enumerate their moves in the two ways EasyLocal accepts:

	SwapExplorer: generator	TwoOptExplorer: cursor
members	<code>moves(solution)</code>	<code>first_move(solution, move),</code> <code>next_move(solution, move)</code>
where the position is kept	in the paused generator	in the move itself
how it reads	nested loops, as if listing every move	a step from one move to the next

Both produce moves only as needed, so First Improvement stops enumeration when it finds an improvement. A generator is usually easier to write; a cursor needs no coroutine and matches the structure of EasyLocal 3 explorers. If an explorer provides both, the library uses the cursor.

The choice does not affect how you compose the search. Runners accept either form, and chapter 7 combines these explorers in one neighborhood.

See also

- [NeighborhoodExplorer](#): the full contract, cursors and unions.

Next steps

Evaluating a move currently means applying it to a copy and re-evaluating the whole tour. [Chapter 5](#) computes the effect of a 2-opt move on the length directly.

5. Delta evaluation

So far, evaluating a move means copying the tour, applying the move and recomputing its length. A **delta cost component** computes the change directly from the affected edges. For 2-opt, this reduces evaluation from a pass over the tour to four distance lookups.

We add a delta for 2-opt because it always replaces two edges. Swap deltas need more care: adjacent or nearby positions can share affected edges, including across the end of the tour. We leave swaps on full evaluation.

The delta cost component

The change of the tour length under a 2-opt move needs only the four cities a, b, c and d:

```
class TwoOptLengthDelta : public easylocal::input_base<Tsp>
{
public:
    using input_base::input_base;

    // The tour goes a -> b ... c -> d; after the move it goes a -> c ... b -> d,
    // the segment b ... c reversed, at the same cost with symmetric distances.
    double delta_evaluate(const Tour& tour, const TwoOpt& move) const
    {
        const auto n = tour.order.size();
        const auto a = tour.order[move.i];
        const auto b = tour.order[move.i + 1];
        const auto c = tour.order[move.j];
        const auto d = tour.order[(move.j + 1) % n];
        const auto& distance = input().distance;
        return distance[a][c] + distance[b][d] - distance[a][b] - distance[c][d];
    }
};
```

It is bound to the component in the neighborhood recipe:

```
auto nhe =
    el::neighborhood<TwoOptExplorer>() | el::delta<TourLength, TwoOptLengthDelta>();
```

- The contract is algebraic: Value + Delta -> Value. With an arithmetic value the delta can have the same type; with a domain value, define operator+(Value, Delta).
- With floating-point values the law holds up to rounding: the length of a tour with decimal distances, updated by deltas, drifts from its full evaluation in the last bits. The checks of chapters 11 and 15 forgive it, within a tolerance; the search compares the costs exactly, unless the cost expression is `cost::approximately(...)` (chapter 2).
- Deltas are **per component and per neighborhood**. The cost of a move is always recomputed by the cost expression from the updated component values, so a delta never deals with weights or hierarchical structure.
- Coverage can be partial: components without a delta for a neighborhood are re-evaluated on a candidate solution. When every component has one, no candidate solution is built.

Write a delta when it saves work. If computing the change requires scanning the whole solution, full evaluation is often just as cheap. Leave that component without a delta and let EasyLocal evaluate the candidate. The Assignment and Exam Timetabling examples use this fallback to avoid code that adds no speed benefit.

Choice: separate or co-located

A delta cost component can be a separate class, as `TwoOptLengthDelta` above, or a `delta_evaluate(solution, move)` member of the component itself. The co-located version of the tour length puts the two computations in one class:

```
// The tour length with its 2-opt delta in the same class: a co-located delta.
// The component is attached as usual, with component<TourLengthWithDelta>(),
// and its delta with delta<TourLengthWithDelta>().
class TourLengthWithDelta : public easylocal::input_base<Tsp>
{
public:
    using input_base::input_base;

    double evaluate(const Tour& tour) const
    {
        const auto n = tour.order.size();
        double length = 0.0;
        for (std::size_t k = 0; k < n; ++k)
            length += input().distance[tour.order[k]][tour.order[(k + 1) % n]];
        return length;
    }

    double delta_evaluate(const Tour& tour, const TwoOpt& move) const
    {
        const auto n = tour.order.size();
        const auto a = tour.order[move.i];
        const auto b = tour.order[move.i + 1];
        const auto c = tour.order[move.j];
        const auto d = tour.order[(move.j + 1) % n];
        const auto& distance = input().distance;
        return distance[a][c] + distance[b][d] - distance[a][b] - distance[c][d];
    }
};
```

The component is attached as before, and `delta` names only the component, which is also its delta cost component:

```
auto colocated_sm =
    el::solution_manager<TourManager>() | el::component<TourLengthWithDelta>();
auto colocated_nhe =
    el::neighborhood<TwoOptExplorer>() | el::delta<TourLengthWithDelta>();
```

The two choices give the same search; they differ in how the code is organised:

	Separate	Co-located
classes	the component, and one delta cost component per neighborhood	one
recipe	<code>delta<TourLength, TwoOptLengthDelta>()</code>	<code>delta<TourLengthWithDelta>()</code>
fits	a component reused by several neighborhoods, each with its own delta, or one without a delta	a component that serves a single neighborhood

A separate delta keeps `TourLength` independent of `TwoOpt`. You can reuse the component with any neighborhood and add deltas as needed.

The co-located version uses one class, but ties it to the move type. Its `evaluate` and `delta_evaluate` members share an object, so they can also share precomputed data. The rest of this tutorial uses the separate version.

Chapter 11 shows how to check that a delta agrees with the full evaluation.

Checking the deltas during a run

A delta bug may appear only after many moves. To check the solutions a search actually visits, compile with `-DEASYLOCAL_VERIFY_DELTAS` or set the definition through `target_compile_definitions`. After every committed move, the search compares each component's incremental value with its full evaluation. The first disagreement stops the program and identifies the component:

```
EASYLOCAL_VERIFY_DELTAS: the delta of cost component #1 disagrees with its full
evaluation after a move
```

Floating-point values are compared with a relative tolerance of $1e-9$. This adds a full evaluation per accepted move, so enable it for debugging and disable it for performance measurements.

See also

- [Cost](#): the delta contract.
- [NeighborhoodExplorer](#): how deltas are bound.

Next steps

[Chapter 6](#) runs the search.

6. Running a search

A runner

A **runner** combines an algorithm with the problem recipes. We now have both: `sm` describes the `SolutionManager` and cost (chapter 2), and `nhe` describes 2-opt moves and their delta (chapter 5). Attach them to First Improvement and run it on the five-city instance:

```
auto fi =
  el::make_runner<runners::FirstImprovement>(runners::FirstImprovementParameters{})
  | sm | dhe;

auto search = fi.bind(tsp);
const auto result = search.run(search.initial_solution());
```

There are three steps:

1. **Compose.** `make_runner` selects the algorithm and stores its parameters; `| sm | dhe` attaches the services in that order. The resulting `fi` still needs no Input and has constructed no services.
2. **Bind.** `fi.bind(tsp)` constructs the algorithm, `TourManager`, `TourLength`, `TwoOptExplorer` and `TwoOptLengthDelta`. The returned *bound runner*, `search`, owns these objects and borrows `tsp`, which must outlive it. For this reason, `bind` rejects a temporary Input.
3. **Run.** `search.initial_solution()` creates a starting tour. `search.run(tour)` repeatedly scans the moves and takes the first improvement, stopping when none remains. It returns the final tour, cost and search counters (see [Results](#)).

`FirstImprovementParameters{}` leaves `max_evaluations` at `easylocal::unlimited` (unlimited in text), so this run reaches a local optimum. To cap its work, pass `{.max_evaluations = 1000}` to `make_runner`. You can also change the parameters through `fi.parameters()` before binding.

The same `fi` can be bound to other Inputs. Reusing the recipes also makes it easy to try another algorithm on exactly the same problem.

Composition can also be written with explicit `with_*` calls, which spell out each step:

```
auto same_runner =
  el::make_runner<runners::FirstImprovement>(runners::FirstImprovementParameters{})
  .with_solution_manager(
    el::solution_manager<TourManager>().with_cost(
      el::component<TourLength>()))
  .with_neighborhood(
    el::neighborhood<TwoOptExplorer>()
    .with_delta<TourLength, TwoOptLengthDelta>());
```

Built-in algorithms

Algorithms live in `easylocal::runners`, with headers under `runners/`. Choose one that supports your neighborhood and cost. The [Runners reference](#) lists their parameters and requirements:

Algorithm	Header	Needs
FirstImprovement, BestImprovement	first_improvement.hpp, best_improvement.hpp	moves or cursor
HillClimbing LateAcceptanceHillClimbing	hill_climbing.hpp, late_acceptance_hill_climbing.hpp	random_move
GreatDeluge	great_deluge.hpp	random_move, an arithmetic cost
SimulatedAnnealing<Temperature, Acceptance>	simulated_annealing.hpp	random_move, cost::delta
TabuSearch<List, Aspiration> and its three variants	tabu_search.hpp	moves or cursor, inverse
ParetoLateAcceptanceHillClimbing	pareto_late_acceptance_hill_climbing.hpp	random_move, a cost::pareto cost, random_solution

Every one bounds its run with `max_evaluations` (unlimited by default: a descent stops at a local optimum).

Simulated Annealing lets you choose a temperature policy from `runners::temperature: Classic, FixedLength, Cutoff, Hybrid, FixedTemperature` or `TimeBased`. `Reheating<...>` wraps any of these except `FixedTemperature`. Acceptance defaults to `runners::MetropolisAcceptance`.

For `Classic`, the parameter block is `SimulatedAnnealingParameters<ClassicParameters>`. It groups schedule settings under `temperature` and also holds the run's evaluation budget: `{.temperature = {...}, .max_evaluations = 10000}`. Configuration paths use `search.temperature.*` (chapter 10). Setting `calibration_samples` above zero estimates the initial temperature from sampled moves.

Pass an RNG to run for a stochastic algorithm:

```
using Classic = runners::temperature::Classic;

auto sa =
    el::make_runner<runners::SimulatedAnnealing<Classic>>({
        .temperature =
            {
                .initial_temperature = 10.0,
                .final_temperature = 0.1,
                .cooling_rate = 0.95,
                .samples_per_temperature = 50,
            },
    })
    | sm | nhe;

std::mt19937_64 rng{42};
auto sa_search = sa.bind(tsp);
const auto annealed = sa_search.run(sa_search.initial_solution(), rng);
```

Metropolis acceptance needs the numeric difference of two costs, `cost::delta(candidate, current)`: arithmetic costs and `cost::hierarchical` provide it, `cost::lexicographic` deliberately does not. The delta of a `cost::hard_soft` cost is infinite when the hard cost worsens, so Simulated Annealing never accepts such a move: the hard cost of its current solution never increases. To let it cross infeasible regions, as `EasyLocal 3's HARD_WEIGHT` did, write the cost as one weighted sum, `cost::sum(component<Hard>() * 1000, component<Soft>())`.

To change the schedule, follow [Advanced: customizing Simulated Annealing](#). It shows the actual search loop, the built-in reheating wrapper and a policy that reheats after consecutive rejected moves.

Results

Built-in algorithms return an `easylocal::search_result` (Pareto Late Acceptance a `pareto_search_result`, which adds the front):

Member	Meaning
<code>solution</code>	the final solution (for Late Acceptance, Great Deluge, Simulated Annealing and Tabu Search, the best found)
<code>cost</code>	its cost
<code>evaluations</code>	evaluations performed, including the initial one
<code>iterations</code>	committed moves (First/Best Improvement, Tabu Search), proposed moves (Hill Climbing, Late Acceptance, Great Deluge, Simulated Annealing)
<code>termination</code>	<code>local_optimum</code> , <code>evaluation_budget_exhausted</code> , <code>time_limit_reached</code> , <code>cancelled</code> , <code>target_reached</code> , <code>idle_limit_reached</code> or <code>completed</code>

Solvers and tools only rely on `solution` and `cost`, the `easylocal::search_result_for` concept.

Reading the instance, printing the solution

A program usually reads its Input from a file and prints the solution it finds. Both go through optional hooks on your types. The tutorial writes them as free functions next to the types, found by argument-dependent lookup:

```
// Optional hooks, found by ADL, that read, write and describe the values
// (chapter 6).
inline Tsp read_input(std::type_identity<Tsp>, std::istream& in)
{
    std::size_t cities = 0; // "n", then the n rows of the distance matrix
    if (!(in >> cities))
        throw std::runtime_error{"invalid TSP header"};
    Tsp tsp{.distance = std::vector(cities, std::vector<double>(cities))};
    for (auto& row : tsp.distance)
        for (auto& value : row)
            if (!(in >> value))
                throw std::runtime_error{"invalid TSP distances"};
    // The 2-opt delta reverses a segment without its cost: symmetric only.
    for (std::size_t i = 0; i < cities; ++i)
        for (std::size_t j = 0; j < i; ++j)
            if (tsp.distance[i][j] != tsp.distance[j][i])
                throw std::runtime_error{"the TSP distances are not symmetric"};
    return tsp;
}

inline Tour read_solution(const Tsp& tsp, std::istream& in)
{
    Tour tour{std::vector<std::size_t>(tsp.cities())};
    for (auto& city : tour.order)
```

```

        if (!(in >> city))
            throw std::runtime_error{"invalid tour"};
        return tour;
    }

    inline void write_solution(const Tsp&, const Tour& tour, std::ostream& out)
    {
        for (const auto city : tour.order)
            out << city << ' ';
        out << '\n';
    }

    inline std::string describe(const Tour& tour)
    {
        std::string text;
        for (const auto city : tour.order)
            text += std::to_string(city) + ' ';
        return text;
    }

    inline std::string describe(const TwoOpt& move)
    {
        return "2-opt(" + std::to_string(move.i) + ", " + std::to_string(move.j) + ")";
    }

```

Hook	Enables
<code>read_input(std::type_identity<Input>, std::istream&) (or static <code>Input::read</code>, or <code>operator>></code>)</code>	reading an Input
<code>read_solution(const Input&, std::istream&) (or static <code>Solution::read</code>, or <code>operator>></code>)</code>	reading a Solution
<code>write_solution(const Input&, const Solution&, std::ostream&) (or <code>Solution::write</code>, or <code>operator<<</code>)</code>	writing a Solution
<code>describe(value) (or a <code>describe()</code> member, or <code>operator<<</code>)</code>	the text of an Input, a Solution or a Move, for people

`<easylocal/app/io.hpp>` reads and writes values through them. Here the five cities come from `five.tsp`, next to the tutorial's sources, and the tour found is printed:

```

// The same five cities, read from a file with the read_input hook.
const auto from_file = el::load_input<Tsp>(EASYLOCAL_TUTORIAL_INSTANCE);
auto file_search = fi.bind(from_file);
const auto file_result = file_search.run(file_search.initial_solution());
std::cout << "from file " << el::describe(file_result.solution) << '\n';

```

Function	
<code>read_input<Input>(in),</code> <code>load_input<Input>(path)</code>	an Input, from a stream or a file
<code>read_solution<Solution>(input, in),</code> <code>load_solution<Solution>(input, path)</code>	a Solution of input
<code>write_solution(input, solution, out),</code> <code>save_solution(input, solution, path)</code>	writes a Solution
<code>describe(value)</code>	the text of a value

These helpers throw `std::runtime_error` on stream failures; file errors name the affected path. Implement only the hooks you use: a program that constructs its Input in code needs no `read_input`.

The Session (chapter 12) and interactive tester (chapter 14) reuse these hooks, so adding a frontend does not require another file parser or solution writer.

See also

- [Runners](#): Runner, the built-in algorithms and their parameters.

Next steps

[Chapter 7](#) adds a second neighborhood.

7. Combining neighborhoods

We now have two ways to change a tour: swaps and 2-opt. Combining them lets a search try improvements that either neighborhood alone might miss.

`neighborhood_union` combines the explorers without changing either one. Swaps keep their generator, 2-opt keeps its cursor, and each keeps its own move type:

```
auto both =
  el::neighborhood_union(
    el::neighborhood<TwoOptExplorer>()
    | el::delta<TourLength, TwoOptLengthDelta>(),
    el::neighborhood<SwapExplorer>()
    | el::random_biases(3.0, 1.0);

auto union_sa =
  el::make_runner<runners::SimulatedAnnealing<Classic>>({
    .temperature =
      {
        .initial_temperature = 10.0,
        .final_temperature = 0.1,
        .cooling_rate = 0.95,
        .samples_per_temperature = 50,
      },
  })
  | sm | both;
```

- Deterministic algorithms enumerate the children in order, each in its own way: the swap moves from the generator, the 2-opt moves from the cursor.
- Each child is evaluated by its own delta. Swaps have none for `TourLength`, so this union evaluates a swap on a candidate tour and keeps `TwoOptLengthDelta` for the 2-opt moves; adding a swap delta removes the candidate tour from the union altogether.
- Random sampling selects a child according to its bias, then asks for a move. Here 2-opt is chosen with probability 3/4. A zero bias excludes a child from random selection; enumeration still includes it. If a child has no move, the union excludes it and tries another. Configure the biases through `NeighborhoodUnionParameters`.
- Unions nest, and traces report the route of every move through the nesting (chapter 17).

Algorithms need no change to use a union: it is just another neighborhood.

See also

- [NeighborhoodExplorer](#).

Next steps

[Chapter 8](#) writes a new search algorithm.

8. Writing your own runner

To try a search strategy of your own, write a class with a run member. The framework supplies evaluation, counters and run control, leaving you to decide which moves to accept. This example samples random moves and keeps improvements:

```
struct RandomDescentParameters
{
    easylocal::parameter<easylocal::limit, "max_evaluations"> max_evaluations{
        1000}; // a count, or easylocal::unlimited

    // The schema makes the parameters configurable (chapter 10).
    static consteval auto parameter_schema()
    {
        return easylocal::config::fields(
            easylocal::config::field<&RandomDescentParameters::max_evaluations>(
                "Evaluation budget",
                easylocal::config::range(1, easylocal::unlimited)));
    }

    easylocal::config::validation_result validate() const
    {
        return easylocal::config::check_schema(*this);
    }
};

class RandomDescent : public easylocal::parameters_base<RandomDescentParameters>
{
public:
    using parameters_base::parameters_base; // configurable, built from the block

    // The result type is the one run.finish() returns: auto deduces it.
    template<class Run, std::uniform_random_bit_generator RNG>
    auto run(Run& run, Run::solution_type solution, RNG& rng) const
    {
        run.limit_evaluations(parameters().max_evaluations);
        auto current = run.start(solution); // evaluates, emits run_started

        // should_stop: cancelled, the budget spent, the target reached or the
        // time up.
        while (!run.should_stop())
        {
            auto move = run.random_move(solution, rng);
            if (!move) // no move at all: a local optimum
                return run.finish(
                    std::move(solution),
                    current.cost(),
                    easylocal::termination_reason::local_optimum);

            run.next_iteration();
            auto candidate = run.evaluate_move(solution, current, *move);
            // A descent only ever improves: every commit is a new incumbent,
            // which commit_improvement() reports to the trace.
            if (run.better(candidate.cost(), current.cost()))
                run.commit_improvement(solution, current, std::move(candidate), *move);
        }
        return run.finish(std::move(solution), current.cost()); // run_finished
    }
};
```

The bound runner passes an `easylocal::search_run` to `run`. It provides comparison and neighborhood access, and handles the work shared by algorithms:

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)</code>	first evaluation; emits <code>run_started</code> , reports progress
<code>should_stop()</code>	true on cancellation, a reached target cost, an exhausted evaluation budget or a passed time limit; records the reason
<code>moves(solution), random_move(solution, rng)</code>	neighborhood access
<code>evaluate_move(solution, current, move)</code>	counts, emits <code>move_evaluated</code> , reports progress
<code>commit(solution, current, candidate, move)</code>	applies the move, emits <code>move_accepted</code>
<code>commit_improvement(solution, current, candidate, move)</code>	applies the move, emits <code>move_accepted</code> and <code>incumbent_updated</code> , for an algorithm whose current solution is its best
<code>next_iteration()</code>	advances the iteration counter
<code>incumbent_updated(previous, cost)</code>	for algorithms that keep a best-so-far solution
<code>finish(solution, cost[, reason])</code>	emits <code>run_finished</code> , returns a <code>search_result</code>

Check `should_stop()` in the loop to honour cancellation and run limits. Keep per-run state in local variables so run can be const and the algorithm can be reused.

In a descent, the current solution is also the best found. Use `commit_improvement()` to apply a move and report the new incumbent. Algorithms that accept worsening moves, such as Simulated Annealing, use `commit()` and track the best solution separately with `best_so_far`.

The runner is used like a built-in one:

```
auto descent =
  el::make_runner<RandomDescent>(RandomDescentParameters{.max_evaluations = 200})
  | sm | nhe;
```

`parameters_base<RandomDescentParameters>` declares `parameters_type`, stores the parameters and exposes them through `parameters()`. The runner keeps this block and constructs the algorithm from it when bound.

Adding `parameter_schema()` and `validate()` to the block (chapter 10) also exposes it to configuration files and frontends. Standalone runners use paths under `search.*`; apps use `runners.<name>.*` (chapter 12). Your custom algorithm then gets the same configuration support as the built-in ones.

An algorithm whose `parameters_type` is not a parameter block can still run, but frontends cannot edit its settings. `check(app, ...)` reports this gap (chapter 15).

See also

- [Runners](#): the complete `search_run` interface and its escape hatches.

Next steps

[Chapter 9](#) lets the framework build the initial solution.

For a worked extension of an existing algorithm, the advanced [Simulated Annealing chapter](#) reads the library's search loop and implements a custom reheating policy.

9. Solvers

A runner starts from a solution you supply. A **solver** handles the whole search from an Input: it creates a starting solution, owns the RNG and coordinates one or more runs. This makes random restarts and staged searches available without writing the orchestration yourself.

```
el::solvers::MultiStart solver{fi, {.starts = 5}};
solver.initialization(el::initialization::random).seed(1);
const auto best = solver.solve(tsp);
```

Construct the solver from a runner, then choose its initialization and seed. MultiStart also takes parameters: here it makes five starts. `initialization::random` needs a random tour for each start, so add `random_solution` to `TourManager`:

```
// The same cities in a random order.
template<std::uniform_random_bit_generator RNG>
Tour random_solution(RNG& rng) const
{
    auto tour = initial_solution();
    std::shuffle(tour.order.begin(), tour.order.end(), rng);
    return tour;
}
```

Use the RNG passed by the solver to keep random starts reproducible. The solver seeds it once; a second `solve()` continues the stream rather than repeating the first solve's starts.

The built-in solvers live in `easylocal::solvers`:

Solver	Does	Needs
LocalSearch	builds an initial solution, runs once	<code>initial_solution</code> or <code>random_solution</code>
MultiStart	starts independent runs, keeps the best	<code>initial_solution</code> or <code>random_solution</code>
Pipeline	runners in sequence, each stage from the solution of the previous one	stages with the same Input and Solution

- The initialization is a tag, `initialization::initial` or `initialization::random`, checked at compile time against the `SolutionManager`. Without one, a solver starts from a random solution when the `SolutionManager` builds one, else from its initial solution (`initialization::automatic`), with seed 0.
- `make_solver<Solver>(runner[, parameters])` deduces the solver type from the runner, as `solvers::MultiStart{runner, {...}}` does.
- `solver.solve(input, el::with(control, tracer))` cancels and traces a solve like a run ([chapter 17](#)); the result counts the evaluations and iterations of all the runs.

Hard and soft costs

For a `cost::hard_soft` objective, `solvers::two_stage()` separates the search for feasibility from optimization of the full cost. It derives the hard-only stage from the existing cost expression, so you can reuse the same components ([chapter 3](#)):

```

auto sm = el::solution_manager<TimetableManager>()
    | el::cost::hard_soft(
        el::cost::sum(el::component<Conflicts>(),
            el::component<Unavailability>()),
        el::component<Compactness>() * 2);

auto solver = el::solvers::two_stage(
    el::make_runner<runners::FirstImprovement>(
        runners::FirstImprovementParameters{})
    | sm | nhe)
    .initialization(el::initialization::initial);

```

- The first stage evaluates only the components of the hard branch, read from the expression, and stops as soon as the hard cost is zero.
- The second stage starts from that solution with the full hierarchical cost, where a hard degradation is never accepted.
- The tutorial's examples/tutorial/staged_main.cpp is a complete program with two_stage.

Several stages

two_stage() is a pipeline of two stages. A pipeline chains any number of runners, each with its own name, cost and neighborhood, over the same solution:

```

// | chains the stages, & gives a stage its options, in parentheses: the
// first one works on the hard cost until it is zero, in up to five
// descents from new random tours; the others continue from its tour on the
// whole cost.
using namespace el::solvers;
auto solver = (stage("feasible", descent) & until_feasible() & attempts(5))
    | stage("descent", descent) | stage("climb", climbing);
const auto result = solver.seed(7).solve(tsp);

```

- | chains stages and & adds stage options. Parenthesize combinations of the two to avoid GCC warnings. The equivalent named members are .until_feasible(), .with_attempts(10), .with_target(0) and .then(stage).
- until_feasible() runs the stage on the hard cost until it is zero; target(cost) stops a stage at another target, in its own cost.
- attempts(5) tries the stage up to five times, stopping at its target and keeping the best result. Here each attempt starts from a new random tour. Later stages reuse the solution they received unless you add restart(initialization::random). First and Best Improvement would repeat the same deterministic search from that solution, so repeated attempts require random restarts.
- result.stages reports each stage (attempts, effort, termination, cost), and the parameters of a stage are under its name (climb.search.*, feasible.attempts). examples/tutorial/pipeline_main.cpp is the complete program.

An app can register a pipeline alongside its runners. Define a stage with stage<Algorithm>(name, parameters) to reuse the app's problem recipes:

```

// The same stages registered in an app, as algorithms on the app's
// recipes: its cost, so its cost.* parameters apply to every stage, and
// its neighborhood, or the stage's own. The app runs the pipeline from the
// current solution; restart() starts the first stage's other attempts from
// new random tours.
auto application = el::app("tsp") | sm | two_opt
    | el::pipeline(

```

```

    "cascade",
    stage<runners::FirstImprovement>("feasible") & until_feasible() & attempts(5)
        & restart(el::initialization::random),
    stage<runners::FirstImprovement>("descent"),
    stage<runners::HillClimbing>("climb", {.max_idle_iterations = 200}, both));
const auto initial = application.bind(tsp).solution_manager().initial_solution();
std::mt19937_64 rng{7};
// run() returns no result for a name that is not registered.
if (const auto cascade = application.run("cascade", tsp, initial, rng))
{
    std::cout << "cascade: violations " << cascade->cost.hard() << ", length "
        << cascade->cost.soft() << '\n';
}

```

- An algorithm stage shares the app's SolutionManager, cost and cost.* parameters. It also uses the app's neighborhood unless you supply one as the third argument, such as both for the climb.
- until_feasible() derives the hard-cost stage from the app's cost, and the stage options are those of a stage of a runner.
- In an app, the pipeline starts from the current solution. restart(el::initialization::random) gives later attempts of the first stage new random tours.
- stage("climb", climbing), a stage of a runner, may still be registered: it keeps its own recipes and its own cost.* parameters.

The command line (`--runner cascade`), the TextUI and the REST service then run it by name from the current solution, as they run a runner, and configure its stages under `runners.cascade.*` ([Apps and tools](#)).

See also

- [Solvers](#).

Next steps

[Chapter 10](#) exposes the parameters to the command line.

10. Configuration

Expose the search parameters to try new settings without recompiling. A **parameter set** collects them under paths such as `search.temperature.cooling_rate`, with a description and current value.

A runner's `configuration()` collects settings from its algorithm (`search`), cost (`cost`), `SolutionManager` (`solution_manager`, when configurable) and neighborhood (`neighborhood`). Add this configuration to your program's set under a prefix, then load values from the command line or a file:

```
el::config::parameter_set configuration;
configuration.add("solver", sa.configuration()); // --solver.search.*
configuration.add("limited", limited.configuration()); // --limited.cost.*

const auto configured = el::config::load_and_apply(argc, argv, configuration);
if (configured.help_requested)
{
    std::cout << el::config::cli_help(argv[0], configuration);
    return 0;
}
if (!configured)
{
    el::config::print_diagnostics(std::cerr, configured);
    return 2;
}
```

```
$ ./easylocal_tutorial --help
--solver.search.temperature.initial_temperature <value>
    Initial annealing temperature
    values: (0, unlimited)
    current: 10
...
$ ./easylocal_tutorial --solver.search.temperature.cooling_rate=2
error: solver.search.temperature.cooling_rate: expected a value in (0, 1), got 2
$ ./easylocal_tutorial --solver.search.temperature.final_temperature=20
error: solver.search.temperature: final_temperature must be smaller than
initial_temperature
```

`load_and_apply` validates the values and updates the runner in place. The example then binds `sa` again and prints the new result as configured annealing. Updates are all-or-nothing: one invalid value leaves every parameter unchanged.

The set refers to the runner's parameters. Keep the runner alive and in the same location while using the set.

The weights of the cost expression are parameters too, under `cost`. For the weighted sum of [chapter 2](#) in a runner configured as solver:

```
$ ./program '--solver.cost.weights=[1, 20]'
```

A `cost::hard_soft` names its branches, so its sums are `solver.cost.hard.weights` and `solver.cost.soft.weights`; children of `cost::in_order` and `cost::apply` are named by position (0, 1, ...).

Parameters of your own classes

In [chapter 3](#), we treated edges longer than 8 as violations. To make that bound configurable, replace the lambda with a class that takes a parameter block:

```

// The bound of chapter 3 as a parameter: how much the longest edge exceeds it.
struct ExcessParameters
{
    easylocal::parameter<double, "bound"> bound{8.0}; // the longest edge allowed

    static constexpr auto parameter_schema()
    {
        return easylocal::config::fields(
            easylocal::config::field<&ExcessParameters::bound>(
                "Longest edge allowed",
                easylocal::config::range(0.0, easylocal::unlimited()));
    }

    easylocal::config::validation_result validate() const
    {
        return easylocal::config::check_schema(*this);
    }
};

class Excess : public easylocal::parameters_base<ExcessParameters>
{
public:
    // The base declares parameters_type, which makes the class configurable,
    // and keeps the block, which parameters() gives back.
    using parameters_base::parameters_base;

    // Its parameters are configured under its name: cost.excess.*
    static std::string_view name()
    {
        return "excess";
    }

    double operator()(double longest) const
    {
        return std::max(0.0, longest - parameters().bound);
    }
};

```

Each field is an `easylocal::parameter<Value, "name">`, and **its default is its member initializer**: `bound{8.0}` here. A field written `bound{}` has no default, and nothing but a value given to it will do: `validate()` reports `bound: not set`, and it has no default, `--help` writes `required: it has no default` and must be set in place of a current value, and a run that reads it anyway throws `parameter_not_set`. This is how the library declares the parameters whose value belongs to the problem — the temperatures of Simulated Annealing, the budget of a search — rather than inventing one. A plain member does not compile: its default would be implicit.

A parameter reads like the value it holds, since it converts to it (`parameters().bound` above); where a template deduces its argument, as `(std::max)(a, b)` does, `value()` spells the value out.

Every configurable class follows the same rule: declare a parameter block as `parameters_type` and accept it in the constructor. `parameters_base<ExcessParameters>` supplies both, stores the block and exposes it through `parameters()`. You can also write these members yourself.

Pass the block to the recipe, which keeps it until binding constructs the class. Use `cost::apply<Excess>(parameters, children...)` for a function or `component<C>(parameters, args...)` for a component:

```
// The same hierarchical cost, its bound a parameter: cost.excess.bound.
auto limited =
    el::make_runner<runners::FirstImprovement>(runners::FirstImprovementParameters{})
    | (el::solution_manager<TourManager>())
    | el::cost::hard_soft(
        el::cost::apply<Excess>({.bound = 8.0}, el::component<MaxEdge>()),
        el::component<TourLength>())
    | nhe;
```

A function or a component is configured under its name(), which must be static: cost.excess.bound here, wherever it is in the expression. Added to the program's set as limited, the bound is --limited.cost.excess.bound:

```
$ ./easylocal_tutorial --limited.cost.excess.bound=5
...
limited 3, 26
```

The same convention applies to runners (chapter 8), explorers and SolutionManagers. Pass blocks through neighborhood<NHE>(parameters, args...) or solution_manager<SM>(parameters, args...); the latter constructs SM(input, parameters, args...). Their paths begin with neighborhood and solution_manager. A class exposing parameters() or configuration() without the required parameters_type produces a diagnostic explaining the missing declaration.

Choose prefixes to keep independent configurations distinct. With one runner, configuration.add(sa.configuration()) gives paths such as --search.*. Adding it under "solver" gives --solver.search.*.

For run limits, the library provides RunParameters: target as cost text, timeout in seconds and max_evaluations. Add it with configuration.add("run", run_parameters) to accept --run.target=0. Then run_parameters.options<Cost>(input) produces the run options, just as timeout(seconds), max_evaluations(n) and stop_at(cost) do individually.

Apps (chapter 12) collect these settings automatically, using runners.<name> for algorithms. The command line, TextUI and REST service all use the same paths and validation rules.

Your own parameters take part by describing themselves with a schema:

```
struct AppParameters
{
    // No default: the program does not run until --application.instance_file
    // gives it one.
    easylocal::parameter<std::filesystem::path, "instance_file"> instance_file{};
    easylocal::parameter<std::uint64_t, "seed"> seed{0};

    static consteval auto parameter_schema()
    {
        return easylocal::config::fields(
            easylocal::config::field<&AppParameters::instance_file>(
                "TSP instance", easylocal::unlimited),
            easylocal::config::field<&AppParameters::seed>(
                "RNG seed", easylocal::unlimited));
    }

    easylocal::config::validation_result validate() const;
};

easylocal::config::parameter_set configuration;
```

```
configuration.add("application", app_parameters); // --application.instance_file
configuration.add("solver", runner.configuration()); // --solver.search.*
```

A block may nest another one, as a group of the schema: `config::group<"temperature", &Parameters::temperature>("...")` puts the nested block's fields under `temperature.`, and validates it together with the enclosing one.

Domains, conditions and requirements

A schema also describes allowed values and relationships between fields. Declare these once to use them for both validation and automatic tuning. Simulated Annealing's classic schedule shows the three kinds of rule:

```
static consteval auto parameter_schema()
{
    return config::fields(
        config::field<&ClassicParameters::initial_temperature>(
            "Initial annealing temperature",
            config::range(0.0, easylocal::unlimited).open_low()),
        config::field<&ClassicParameters::final_temperature>(
            "Final annealing temperature",
            config::range(0.0, easylocal::unlimited).open_low()),
        config::field<&ClassicParameters::cooling_rate>(
            "Multiplicative cooling factor", config::range(0.0, 1.0).open()),
        // ... samples_per_temperature, calibration_samples
        config::field<&ClassicParameters::initial_acceptance>(
            "Acceptance probability of an average worsening move at the "
            "estimated initial temperature",
            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"));
}
```

- A **domain**, the second argument to `field`, gives the allowed values. Use `config::range(low, high)`, `config::one_of("fixed", "random")` or `easylocal::unlimited` for an unrestricted value, such as a filename. Except for booleans, every field needs an explicit domain to pass `check(app, ...)` (chapter 15).
- A field of an **enum** type takes the names its `easylocal::enumeration` declares, which are also its domain: `--start greedy,{random, initial, greedy}` in `--help`, a categorical parameter for `irace`. See [Enumerations](#).
- A **condition**, `.only_if(...)`, says when a field applies. Initial acceptance matters only during temperature calibration; when calibration is disabled, that field's domain is not checked.
- A **requirement**, `config::require(expression, message)`, relates fields. Validation reports the message if the relationship does not hold.

Ranges include both endpoints by default. Use `.open()`, `.open_low()` or `.open_high()` to exclude endpoints, and `.log()` for values spanning orders of magnitude. `config::range(0.0, easylocal::unlimited)` has no upper bound; it includes infinity unless you add `.open_high()`.

Build conditions and requirements from field references such as `config::value<"name">` or `value<"temperature.cooling_rate">` for a nested group. Combine them with comparisons, arithmetic, `&&`, `||` and `!`.

Set validation checks these rules automatically. In the block's own `validate()`, call `config::check_schema` before any checks that the schema cannot express:

```

config::validation_result validate() const
{
    if (const auto schema = config::check_schema(*this); !schema)
        return schema;
    // ... the checks the schema does not declare
    return config::validation_result::success();
}

```

Automatic configurators read the same declarations: irace (chapter 13) tunes the fields with a domain, only when their condition holds, and never proposes values that break a requirement.

Configuration files

load_and_apply also reads a file given as --config <file>, with one path = value per line and # for comments, which take whole lines (a # after a value is part of the value):

```

# annealing.conf
solver.search.temperature.cooling_rate = 0.9
solver.search.temperature.samples_per_temperature = 100

```

Values on the command line win over the file.

TOML files

For files with sections, types and comments, the optional ConfigTOML component reads TOML. Each table is a prefix and each key a parameter, so the file mirrors the paths:

```

# Simulated Annealing for the tutorial's TSP (chapter 10).
# Each table is a prefix of the paths, each key a parameter:
# [solver.search.temperature] cooling_rate is solver.search.temperature.cooling_rate.

[solver.search.temperature]
initial_temperature = 20.0
final_temperature = 0.05
cooling_rate = 0.9
samples_per_temperature = 100

# The weights of cost::sum(TourLength, MaxEdge), in order.
[solver.cost]
weights = [1.0, 5.0]

```

The program reads the file with load_toml_file, which turns every key into the same path = value override as the command line, and applies them with the set's apply:

```

el::config::parameter_set configuration;
configuration.add("solver", sa.configuration()); // paths solver.*

// Read the file: every key becomes a "path = value" override.
const auto file =
    el::config::load_toml_file(argc > 1 ? argv[1] : EASYLOCAL_TUTORIAL_CONFIG);
if (!file)
{
    // Each error with its line; a value's names its path.
    el::config::print_diagnostics(std::cerr, file);
    return 2;
}

```

```

// Apply the overrides: all of them, validated, or none.
const auto applied = configuration.apply(file.overrides);
if (!applied)
{
    for (const auto& diagnostic : applied.diagnostics)
        std::cerr << diagnostic.path << ": " << diagnostic.message << '\n';
    return 2;
}

```

- `load_toml_file` fails on a malformed file, whose diagnostic gives its line and column, and on values that have no parameter counterpart, such as dates or an array of strings; their diagnostics give the path. An array of arrays, `[[1, 2], [3]]`, sets a list of lists.
- Each value is read by its TOML type: `true` and `false` set a `bool`, an integer sets a number, a float sets only a floating-point number (`3.0` does not set an integer).
- `apply` checks every path and validates every value before changing anything: an unknown key (a typo such as `coolin_rate`) or an invalid value (`cooling_rate = 2.0`) leaves the parameters as they were.
- The runner here has its temperatures set, since they have no default, and a cost with two weights; with the file, Simulated Annealing finds the cost 66, the length 26 plus 5 times the longest edge, 8.
- To let the command line override the file, give `load_toml_file` to `load_and_apply`, which reads it for `--config: el::config::load_and_apply(argc, argv, configuration, el::config::load_toml_file)`; an app does the same with `cli::options::read_config` (chapter 12).

The program is `examples/tutorial/toml_main.cpp`, built when the component is enabled (`-DEASYLOCAL_ENABLE_CONFIG_TOML=ON`), and linked with `EasyLocal::ConfigTOML`:

```

find_package(EasyLocal CONFIG REQUIRED COMPONENTS Core ConfigTOML)
target_link_libraries(tsp PRIVATE EasyLocal::Core EasyLocal::ConfigTOML)

```

See also

- [Configuration](#).

Next steps

[Chapter 11](#) checks the components.

11. Testing your components

Before trusting search results, check that moves preserve valid solutions and deltas agree with full evaluation. <easylocal/testing.hpp> supplies these checks so you can run them in your test suite. They share a fixture containing a small Input, a valid Solution and its SolutionManager:

```
namespace elt = easylocal::testing;
// Not the identity tour: on 0, 1, 2, 3, 4 position k holds city k, and a
// delta that mixes up positions and cities would still be right.
const elt::fixture<tutorial::TourManager> tsp{
    tutorial::five_cities(),
    tutorial::Tour{{0, 2, 4, 1, 3}},
};
```

Each check names the type under test and builds it from the fixture, as an app would: a component or a delta cost component from the Input, a NeighborhoodExplorer from the SolutionManager.

```
using namespace tutorial;
return elt::run_checks(
    elt::check_solution_manager(tsp),
    elt::check_cost_component<TourLength>(tsp),
    elt::check_neighborhood<SwapExplorer>(tsp),
    elt::check_neighborhood<TwoOptExplorer>(tsp),
    elt::check_delta_cost_component<TwoOptExplorer, TourLength, TwoOptLengthDelta>(
        tsp),
    // No delta cost component: the check uses the component's own delta_evaluate.
    elt::check_delta_cost_component<TwoOptExplorer, TourLengthWithDelta>(tsp));
```

Check	Verifies
check_solution_manager(f)	the fixture, initial and random solutions are valid
check_cost_component<Component>(f)	evaluating the same solution twice gives equivalent values
check_neighborhood<NHE>(f)	enumerated and sampled moves are valid, keep the solution valid and change it; random moves are enumerated ones, drawn from the generator given
check_delta_cost_component<NHE, Component, Delta>(f)	value + delta equals the full re-evaluation after each valid move

Move checks use both the fixture's solution and random solutions followed by random walks. This helps catch deltas that work on one tour but fail on another.

The fixture deliberately uses a shuffled tour. On 0, 1, 2, 3, 4, city identifiers equal their positions, so a delta could incorrectly use `distance[i][j]` instead of `distance[order[i]][order[j]]` and still pass.

Failures identify the move, starting solution and disagreeing values. The report prints up to three failures per check. A hook that throws fails its check without stopping the remaining checks.

For a co-located delta (chapter 5) omit the third type, as for `TourLengthWithDelta` above: the check then uses the component's own `delta_evaluate`.

`fixture<SM>{input}` uses the initial solution when none is supplied. A third argument controls sampling, random walks and seed, for example `{.random_samples = 16, .max_enumerated_moves = 256, .random_solutions = 4, .walk_length = 8, .seed = 7}`.

Comparisons use `testing::approximately`, with relative and absolute floating-point tolerances of 1e-9 by default. This allows for rounding drift from incremental updates. Other values are compared exactly. Set `{.tolerance = {.relative = 0, .absolute = 0}}` for exact comparison, or provide a second fixture template argument to replace the comparison policy.

If a component needs more than the `Input` to construct it, pass an object: `check_cost_component(f, TourLength{...})`. Explorers and deltas support the same form. Include `testing.hpp` in your test executable; the Core umbrella provides the options and reports for `check(app)`, but not these individual checks.

For bugs that appear only during a longer search, enable `EASYLOCAL_VERIFY_DELTAS` in a debug build. It compares incremental and full evaluation after every committed move.

For a whole composed problem, `easylocal::check(app, input)` runs the same checks against an app ([chapter 15](#)).

See also

- [Testing](#).

Next steps

[Chapter 12](#) packages the problem as an application.

12. Applications

An **app** packages the problem recipes and gives each available algorithm a name. A **Session** uses that app on one Input and keeps a current solution. Together they let the command line, interactive tester and REST service reuse the same problem definition and parameter settings.

The app

```
auto application =
  el::app("tsp")
    .with_solution_manager(sm)
    .with_neighborhood(nhe)
    .with_runner<runners::FirstImprovement>("fi")
    .with_runner<runners::SimulatedAnnealing<Classic>>(
      "sa",
      {.temperature = {
        .initial_temperature = 10.0,
        .final_temperature = 0.1,
        .samples_per_temperature = 50}});

auto piped_application = el::app("tsp") | sm | dhe
  | el::runner<runners::FirstImprovement>("fi")
  | el::runner<runners::SimulatedAnnealing<Classic>>(
    "sa",
    {.temperature = {
      .initial_temperature = 10.0,
      .final_temperature = 0.1,
      .samples_per_temperature = 50}});
```

- The app is a description. Binding it will construct the services.
- Register a runner with its algorithm class, a name and optional parameters. The app stores the parameters; change them later through `runner_parameters<Algorithm>("name")`.
- Simulated Annealing is registered with its parameters, those of its temperature policy as a group: `runners::SimulatedAnnealing<Classic>` takes `SimulatedAnnealingParameters<ClassicParameters>`, whose temperature is the policy's and whose `max_evaluations` bounds the run.
- The two spellings are equivalent: `with_*` calls, or pipes with `el::runner<Algorithm>(name, parameters)` for each registration.
- Registration checks compatibility at compile time. For example, First Improvement needs an explorer that can enumerate moves, through `moves()` or a cursor.
- An app also registers pipelines of stages, `el::pipeline("name", stages...)`, run by name like a runner; a stage may be an algorithm on the app's recipes, `stage<A>(name, parameters)` (chapter 9).
- `application.bind(tsp)` builds the services once for an Input, a `BoundApp` whose `run("fi", solution, rng)` runs a registration on them; the Session below does this for you.

The Session

A Session owns its Input, constructs the services and keeps a *current solution*. You can inspect and change that solution one move at a time, or run an algorithm to improve it:

```
// The app on one Input, which the session owns, with a seeded RNG.
el::Session session{application, tsp, /* seed */ 2026};
session.use_initial_solution();

// A step by hand: select the first improving move, if any, and apply it.
if (session.use_first_improving_move())
  session.apply_move();
```

```
// A registered runner, by name, from the current solution; false means no
// runner has that name.
if (!session.run("sa")) // receives the session's RNG
    return 1;
const double session_cost = session.evaluate();
const Tour session_tour = session.solution(); // a copy: the session goes on
```

- `el::Session{application, tsp, seed}` copies the app and Input, so the session can outlive both source variables. Its seed defaults to 0 and controls random solutions, moves and run seeds. Repeating the same commands with the same seed reproduces the session. Create a new Session for another Input.
- `use_initial_solution()` makes the initial tour the current solution (`use_random_solution()` makes a random one, with the session's RNG).
- Select a move with `use_first_improving_move()`, then apply it with `apply_move()`. Selection returns false if no improvement exists. Other selectors include `use_first_move`, `use_next_move`, `use_best_move` and `use_random_move`. Before applying a move, `evaluate_move()` shows its resulting cost.
- `run("sa")` runs the runner registered as "sa" from the current solution, and makes its result the current solution. It returns false when no runner has that name.
- `evaluate()` and `solution()` read the current cost and solution. `solution()` returns a reference to the session's solution, which the next command may replace: copy it to keep it, as here.

Selection functions and `run` are `[[nodiscard]]`. Check their results so a missing move or misspelt runner name does not silently leave the session unchanged.

Each run builds fresh services from the app, with the runner's current parameters, so runs share nothing but the immutable Input.

What the app and the Session hold, and what each run builds:



Syntax error in text mermaid version 11.17.2

Parameters and targets

The app gathers its parts' parameters under these paths:

- `runners.<name>.*`: the named algorithm;
- `cost.*`: expression weights and configurable components or functions;
- `solution_manager.*`: the SolutionManager;
- `neighborhood.*`: the explorer or union biases.

Use the Session to update these values and run toward a target cost:

```
// The app's parameters by path; they change this session's app only.
const std::array changes{
    el::config::text_override{"runners.sa.temperature.cooling_rate", "0.9"};
if (!session.configure(changes)) // all or none, checked
    return 1;

// A run that stops at the first tour of length 26 or less.
session.use_initial_solution();
```

```
if (!session.run("sa", el::stop_at(session.read_cost("26"))))
    return 1;
```

- `configure(changes)` validates all `path = value` overrides before applying any. Successful changes update the session's app and rebuild its services, while keeping the current solution. Check the returned diagnostics for errors such as unknown paths. `configuration()` lists the current settings.
- `read_cost("26")` parses a target. Use a number for a scalar cost or `[hard, soft]` for a hierarchical one. A problem can supply its own parser, `read_cost(const Tsp&, std::string_view)`. Pass the result to `el::stop_at` to stop when the search reaches that cost or better.

A command-line program

`el::cli::run`, from `<easylocal/app/cli.hpp>`, handles command-line parsing, instance loading, execution and output. Define the app and let it provide the standard program flow, as in `examples/tutorial/cli_main.cpp`:

```
auto application = el::app("tsp")
    | (el::solution_manager<TourManager>() | el::component<TourLength>())
    | (el::neighborhood<TwoOptExplorer>()
        | el::delta<TourLength, TwoOptLengthDelta>())
    | el::runner<runners::FirstImprovement>("fi")
    | el::runner<runners::SimulatedAnnealing<Classic>>(
        "sa",
        {.temperature = {
            .initial_temperature = 10.0,
            .final_temperature = 0.1,
            .samples_per_temperature = 50}});

return el::cli::run(application, argc, argv);
```

```
$ easylocal_tutorial_cli --instance five.tsp --runner sa --seed 7
cost 26
time 0.0119483
iterations 6750
evaluations 6751
termination completed
2 3 1 0 4
```

Switch	
<code>--instance <file></code>	the Input, read with the <code>read_input</code> hook (chapter 6); required
<code>--seed <n></code>	the seed of the random generator given to random solutions and stochastic runners
<code>--runner <name></code>	a registered runner; the first one when empty
<code>--start random, --start initial</code>	the starting solution; random by default when the problem has <code>random_solution</code>
<code>--solution <file></code>	the starting solution read from a file, with <code>read_solution</code>
<code>--output <file></code>	where the solution goes, with <code>write_solution</code> ; standard output by default
<code>--target <cost></code>	stop at the first solution that reaches this cost, as <code>session.read_cost</code> reads it
<code>--timeout <seconds></code>	stop the run after this many seconds, such as 10 or 2.5; the termination is then time limit reached
<code>--max_evaluations <n></code>	stop the run after n evaluations (unlimited by default); a runner's own budget, if smaller, still applies
<code>--runners.<name>.*, --cost.*, --neighborhood.*</code> <code>solution_manager.*, --neighborhood.*</code>	the app's parameters, as <code>configuration()</code> lists them
<code>--report true</code>	also print the value of each cost component, see below
<code>--trace <file></code>	record the trace of the run, with timestamps: JSON Lines for a <code>.jsonl</code> name, ELTR otherwise
<code>--config <file></code>	the same settings from a file (chapter 10); <code>--help</code> lists them all

- Output includes the cost, elapsed seconds and solution. Built-in algorithms also report iterations, evaluations and the termination reason. Exit status is 0 after a run, 2 for invalid arguments and 1 for an execution failure, such as an unreadable file.
- Add program-specific parameters with `el::cli::run(application, argc, argv, {program_parameters = own})`. The blocks referenced by `own` must outlive the call (chapter 10).
- Set default switches through defaults, a `cli::CommandLineParameters`. For example: `{.defaults = {.instance = "...", .seed = 2026, .start = "initial"}}`.
- A program that needs more, such as several runs or a solver, builds the same steps from `el::config::load_and_apply` and a `Session`; see the [reference](#).

The TextUI (chapter 14) and the REST service (chapter 16) take the same parameter paths and targets.

A report of the cost components

Use `--report true` to see how each component contributes to the final cost. Two optional members give a component a name and explanatory text. Here `TourLength` lists the edges it adds up:

```

// Optional, for people: the name of the component and a text that
// explains its value on a tour, here the edges it adds up (chapter 12).
static std::string_view name()
{
    return "TourLength";
}

std::string describe(const Tour& tour) const
{
    const auto n = tour.order.size();
    std::string text;
    for (std::size_t k = 0; k < n; ++k)
    {
        const auto edge = input().distance[tour.order[k]][tour.order[(k + 1) % n]];
        text += (k == 0 ? "" : " + ") + std::format("{:}", edge);
    }
    return text;
}

```

```

$ easylocal_tutorial_cli --instance five.tsp --runner fi --seed 1 --report true
cost 26
time 1.5e-05
iterations 2
evaluations 9
termination local optimum
component TourLength 26
  2 + 4 + 6 + 5 + 9
0 1 3 4 2

```

- Both members are optional. Without `name()`, a component is shown by its position in the recipe, #1; without `describe`, only its value is shown.
- The value is the component's own, before the weights of the cost expression.
- `describe` is the place for what EasyLocal 3's `PrintViolations` printed: the violations a constraint counts, for example.
- The same report is `session.cost_report()` in a program, and the solution window of the `TextUI` (chapter 14) shows it.

See also

- [Apps and tools](#): the complete app and Session interfaces.

Next steps

[Chapter 13](#) tunes the parameters of the program with `irace`.

13. Tuning with irace

Good parameter values depend on the problem and its instances. `irace` searches for them by running candidate configurations on training instances and comparing their results. An app using `cli::run` can generate the scenario from its parameter schema, saving you from describing the same parameters again for the tuner.

`irace` is an R package: install [R](#), then `Rscript -e 'install.packages("irace")'`.

What is tuned

Tuning needs a finite domain for each parameter. Built-in schemas already provide these for rates and probabilities, such as `cooling_rate` in $(0, 1)$. Temperatures need a range suited to the scale of your cost; their general domain allows any positive value.

You can narrow domains for an experiment. Schema conditions and requirements carry over too: inactive parameters are not tuned, and invalid combinations are excluded. In `examples/tsp/sa_main.cpp`, the program sets a tuning range for the initial temperature:

```
// --tuning.irace=DIR writes an irace scenario: the parameters with a domain
// are tuned, here also the initial temperature, on a logarithmic scale.
return easylocal::cli::run(
  application,
  argc,
  argv,
  {.defaults =
    {
      .instance = EASYLOCAL_TSP_INSTANCE_FILE,
      .seed = 2026,
      .start = "initial",
    },
    .tuning = {
      {"runners.sa.temperature.initial_temperature",
       easylocal::config::range(1.0, 100.0).log()},
    }
  });
```

The scenario

`--tuning.irace=DIR` writes the scenario without starting a search. Other switches on the command line supply the baseline settings for its runs:

```
$ easylocal_tsp_sa --tuning.irace=tuning --
runners.sa.temperature.allowed_iterations=20000
wrote tuning/parameters.txt
wrote tuning/fixed.conf
wrote tuning/target-runner
wrote tuning/instances.txt
wrote tuning/scenario.txt
updated tuning/configurations.txt
2 parameters to tune, 3 more to complete in parameters.txt; list the instances in
instances.txt; then run irace in tuning
```

File	
parameters.txt	the parameters to tune, one per line with its switch, type, range and condition; the others are commented out, with a range to start from; the requirements as [forbidden] combinations
configurations.txt	the current values, which irace tries first
fixed.conf	the values given on the command line, read by every run with --config
target-runner	the script irace calls: it runs the program with --tuning.print=cost on an instance, a seed and the candidate values
instances.txt	the instances to tune on, one per line: the program's default instance to begin with
scenario.txt	irace's settings: the files above and the budget, maxExperiments, the number of runs

parameters.txt

```
# Multiplicative cooling factor (default 0.75)
runners.sa.temperature.cooling_rate "--runners.sa.temperature.cooling_rate=" r (0.0001,
0.9999)

# Maximum number of annealing iterations (default 20000)
# no finite domain: a range around the default to start from
# runners.sa.temperature.allowed_iterations "--
runners.sa.temperature.allowed_iterations=" i,log (2000, 200000)
```

Initial acceptance is commented out here because calibration is disabled: `calibration_samples` is fixed at 0. The generated condition is ready if you later enable tuning for both fields. The temperature requirement becomes a forbidden combination, using the fixed final temperature:

parameters.txt

```
[forbidden]
# final_temperature must be smaller than initial_temperature
# with every parameter it names tuned: !(runners.sa.temperature.final_temperature <
runners.sa.temperature.initial_temperature)
!(0.25 < runners.sa.temperature.initial_temperature)
```

Running irace

Prepare the generated scenario for your experiment:

1. List the training instances in `instances.txt`.
2. Set the run budget in `scenario.txt`.
3. Select parameters and adjust ranges in `parameters.txt`. Add any extra invalid combinations to [forbidden].
4. Run `--tuning.irace=DIR` again. It preserves your scenario edits, checks that the selected parameters exist, and refreshes `configurations.txt`. Starting values outside the selected ranges are moved into range.

Then run irace in the directory:

```
$ cd tuning && irace
...
# Best configurations as commandlines (first number is the configuration ID; listed
from best to worst according to the sum of ranks):
11 --runners.sa.temperature.initial_temperature=4.1621 --
runners.sa.temperature.cooling_rate=0.6166
30 --runners.sa.temperature.initial_temperature=5.4712 --
runners.sa.temperature.cooling_rate=0.6343
```

The best configurations are switches of the program: give them on its command line, or as `path = value` lines in a configuration file (chapter 10).

The cost as one number

`irace` compares a single number per run. `--tuning.print=cost` prints it; `cost_time` also prints elapsed seconds. Numeric costs are used directly. Hierarchical costs become `hard * W + soft`, while lexicographic costs weight each level by a power of `W`. Set `W` with `--tuning.hard_weight` (default 10^9), large enough to exceed every soft cost.

This conversion uses a double, which represents integers exactly up to 2^{53} (about 9×10^{15}). With the default weight, three or more lexicographic levels, or a hard cost above about 9×10^6 , can lose lower-priority information to rounding. In that case, define `scalar_cost(const Input&, const Cost&)` alongside the `Input` to provide a suitable scalar measure.

Cost parameters, such as sum weights, are excluded from tuning: changing them would change the objective used to compare configurations.

See also

- [Apps and tools](#): the tuning parameters, the generated files and the cost as one number.

Next steps

[Chapter 14](#) explores the app in the interactive tester.

14. The interactive tester

The **interactive tester**, or TextUI, lets you explore the app without writing a debugging interface. Load a solution, inspect individual moves, change parameters and watch a search. It uses the same Session operations and checks as your code.

Enable the optional TUI component, based on FTXUI:

```
find_package(EasyLocal CONFIG REQUIRED COMPONENTS Core TUI)
target_link_libraries(tsp_tui PRIVATE EasyLocal::TUI)

auto application = el::app("tsp")
| (el::solution_manager<TourManager>() | el::component<TourLength>())
| (el::neighborhood<TwoOptExplorer>()
  | el::delta<TourLength, TwoOptLengthDelta>())
| el::runner<runners::FirstImprovement>("fi")
| el::runner<runners::SimulatedAnnealing<Classic>>(
  "sa",
  {.temperature = {.initial_temperature = 10.0, .final_temperature = 0.1}});

el::tui::run(
  application,
  {
    .title = "TSP tester",
    .seed = 2026, // the RNG for random solutions, moves and stochastic runners
    .input_path = EASYLOCAL_TUTORIAL_INSTANCE, // loaded with the read_input hook
  });
```

`tui::run(application, options)` opens the tester and returns when you quit. If `input_path` is set, it loads the instance through your `read_input` hook. Otherwise, load an instance from the Input/Output page.

The program is `examples/tutorial/tui_main.cpp`, built when the TUI component is enabled (`-DEASYLOCAL_ENABLE_TUI=ON`).

A tour of the pages

The tester has three pages, switched with F3, F4 and F5; the header shows the instance, the current seed and the cost of the current solution.

On the **Input/Output** page, I creates the initial solution and C runs the app check of chapter 15:

```

TSP tester | five.tsp [seed=2026] Input/Output Move Run COST 29
-----
Input / Output -----
Setup complete - use F4 Move or F5 Run

Instances
> five.tsp
L Load input Browse...
Create solution
I Initial R Random
Solutions
Browse... Shift-L Load solution W Save
Diagnostics
C Check

OK Check passed: 503 semantic checks
L Load input I Initial R Random Shift-L Load solution W Save C Check
F3 I/O F4 Move F5 Run | F1 Input F2 Solution S Show solution | ? Help | q quit |
Tab/Shift-Tab focus

```

Figure 1: The Input/Output page after creating the initial solution and running the check

On the **Move** page, B selects the best move; the panel shows its incremental and full evaluation, and A would apply it:

```

TSP tester | five.tsp [seed=2026] Input/Output Move Run COST 29
-----
Actions -----
Select move
B Best
I First improving
F First
N Next
R Random
Commit
A Apply

Move - 2-opt -----
Valid: yes
2-opt(1, 3)
Candidate (incremental): 26
Candidate (full): 26
Delta check: OK

Diagnostics -----
P List T Stats C Costs D Indep U Distribution

OK Selected best move
B Best I First improving F First N Next R Random A Apply P List T Stats C Costs D Indep
U Distribution
F3 I/O F4 Move F5 Run | F1 Input F2 Solution S Show solution | ? Help | q quit |
Tab/Shift-Tab focus

```

Figure 2: The Move page with the best 2-opt move and its delta check

On the **Run** page, press G or Enter on the runner list to edit the selected runner's parameters. These are the same settings exposed in chapter 10, with their descriptions and current values. Enter starts the run; Esc cancels:

TSP tester	Parameters of sa		COST 29
	Enter Run Tab next field Esc cancel * changed since start		
Runners	temperature.initial_temperature 10 Initial annealing temperature temperature.final_temperature 0.1 Final annealing temperature temperature.cooling_rate 0.95 Multiplicative cooling factor temperature.samples_per_temperature 100 Proposals evaluated at each temperature temperature.calibration_samples 0 Moves sampled to estimate the initial temperature (0: none) temperature.initial_acceptance 0.5 Acceptance probability of an average worsening move at the estimated ini max_evaluations unlimited Maximum number of solution evaluations, or unlimited		ly seed
OK Selecte F3 I/O	Run Cancel		

Figure 3: The parameters of Simulated Annealing, before running it

Validation happens before the run. An invalid value, such as a cooling rate of 2, keeps the window open and displays an error without applying changes. Accepted values remain for later runs; the Run page lists settings changed since startup. A runner with no parameters starts directly.

The runner runs in the background with live progress and can be stopped with X; here Simulated Annealing improves the tour from 29 to 26:

TSP tester five.tsp [seed=2026]		COST 26
Input/Output Move Run		
Runners		
Runner	fi	
> sa		
Execute	G Run selected	
Problem	P Problem parameters	
Random seed	2026	
Target cost	none	
current cost:	26	
Stop after	seconds none evaluations none	
Last run sa: 29 -> 26 (completed, 9001 evaluations)		
OK Runner completed: sa: 29 -> 26 (completed, 9001 evaluations) G Run selected P Problem parameters F3 I/O F4 Move F5 Run F1 Input F2 Solution S Show solution ? Help q quit Tab/Shift-Tab focus		

Figure 4: The Run page after a Simulated Annealing run

Use *Target cost* to stop at a known optimum or lower bound. A target of 26 ends this search at the first optimal tour. Enter a number for a scalar cost or [hard, soft] for a hierarchical one; the current cost below the field shows the format. Leave the field empty for no target.

The *Stop after* fields set limits in seconds and evaluations, including the initial evaluation. Empty fields leave the runner's own limits in effect. After a run, the controls and status line show the before/after costs and termination reason.

Press P to edit shared problem parameters, such as cost weights (chapter 2) or neighborhood biases (chapter 7). Applying them updates the header cost and Move page evaluations as well as future runs.

On a small terminal, pages scroll to keep the focused control visible. Move between controls with Tab or the arrow keys. The screenshots above come from the actual tutorial program; the same interactions are covered by the TUI's end-to-end tests.

Loading, saving and displaying

The TextUI reuses the I/O and display hooks from [chapter 6](#): `read_input`, `read_solution`, `write_solution` and `describe`. The solution window (S or F2) also shows the component report from chapter 12.

Commands requiring missing hooks are hidden. For display, a Solution without `describe` falls back to `write_solution`; a value with neither appears as *not printable*.

The page title of the Move page, *Move - 2-opt*, comes from a static `name()` member of `TwoOptExplorer`:

```
// The name of the neighborhood in the interactive tester (chapter 14).
static std::string_view name()
{
    return "2-opt";
}
```

Several apps on the same problem

The moves the tester shows are those of the app's neighborhood. To explore several neighborhoods, one per app, open them from a **launcher**: a list of apps that share the Input and the current solution. The tutorial's `launcher_main.cpp` has one app for the 2-opt moves and one for the swaps, over the same `SolutionManager` recipe:

```
// The SolutionManager recipe of both apps: the launcher passes the Input and
// the solution from one app to the other, so they must have the same one.
auto tour_manager()
{
    return el::solution_manager<tutorial::TourManager>()
        | el::component<tutorial::TourLength>();
}

// One app per neighborhood. The type of an app spells out all its recipes, so
// the functions let auto deduce it.
auto two_opt_app()
{
    return el::app("tsp-two-opt") | tour_manager()
        | (el::neighborhood<tutorial::TwoOptExplorer>()
            | el::delta<tutorial::TourLength, tutorial::TwoOptLengthDelta>())
        | el::runner<el::runners::FirstImprovement>("fi");
}

auto swap_app()
{

```

```

return el::app("tsp-swap") | tour_manager()
    | el::neighborhood<tutorial::SwapExplorer>()
    | el::runner<el::runners::FirstImprovement>("fi");
}

```

```

el::tui::run_launcher(
{
    .title = "TSP launcher",
    .tester =
    {
        .seed = 2026,
        .input_path = EASYLOCAL_TUTORIAL_INSTANCE,
    },
},
two_opt_app(),
swap_app());

```

- The launcher owns the Input, read from `tester.input_path` when it is set, and the current solution. Its first entry, *Input and solution*, opens the Input/Output page alone: load another instance, load or save a solution, create the initial or a random one.
- Each app opens on the shared Input and solution. When you leave it with q, its state returns to the launcher. You can improve a tour with swaps, then open it in the 2-opt app to inspect further moves. Loading an instance or solution also updates the shared state.
- Since solutions pass from one app to the other, the apps must have the same SolutionManager recipe, cost included: they differ in the neighborhood and the runners. A launcher of apps with different recipes does not compile.
- To search with both neighborhoods at once, compose them in one app with `neighborhood_union` (chapter 7); the launcher is for examining them one at a time.
- To run searches with different neighborhoods from one app, give a runner a neighborhood of its own, the third argument of runner: it is built over the app's SolutionManager, next to the app's neighborhood, and its parameters are under `runners.<name>.neighborhood.*`:

```

// One app with both neighborhoods: its own is 2-opt, the runner "fi" uses it,
// and the runner "fi-swap" brings its own, built over the same
// SolutionManager.
inline auto tsp_app()
{
    return easylocal::app("tsp") | tsp_solution_manager()
        | (easylocal::neighborhood<TwoOptNeighborhoodExplorer>()
            | easylocal::delta<TourLengthComponent, TwoOptTourLengthDelta>())
        | easylocal::runner<easylocal::runners::FirstImprovement>(
            "fi",
            {.max_evaluations = 100})
        | easylocal::runner<easylocal::runners::FirstImprovement>(
            "fi-swap",
            {.max_evaluations = 100},
            easylocal::neighborhood<SwapCitiesNeighborhoodExplorer>()
                | easylocal::delta<TourLengthComponent, SwapTourLengthDelta>());
}

```

Options

Use `tui::options` to set the title, random seed and initial `input_path` and `solution_path`. Relative paths use `path_base`, or the working directory if it is empty. `path_display` controls how paths appear.

Three options bound display and diagnostic work:

Option	Default	Purpose
<code>max_render_chars</code>	4096	maximum displayed bytes per Input, solution or move; 0 removes the limit
<code>max_diagnostic_entries</code>	256	maximum entries in the neighbors list
<code>random_distribution_rounds</code>	20	random samples per valid move in the distribution check

The seed controls random solutions, moves and stochastic runs. To reset it during a session, edit *Random seed* on the Run page and press Enter or *Apply seed*. The RNG restarts and the header shows the new seed.

See also

- [Apps and tools](#).

Next steps

[Chapter 15](#) checks the composed problem, from code and from the tester.

15. Checking a composed problem

Chapter 11 tested components individually. Now check that they work together: `check` validates the app, while `Session` checks inspect the neighborhood of a particular solution. Both are also available in the interactive tester.

check: the app's contracts

```
const auto report = el::check(application, tsp); // also: check(app, input, solution)
el::print_report(std::cout, report);
if (!report)
    return 1;
```

```
EasyLocal tsp check: 503 checks passed
composition: solution_managers=1, cost_components=1, neighborhoods=1, delta_bindings=1,
runner_registrations=2
```

`easylocal::check(app, input)` builds the services and tests them from the initial solution, or one supplied as the third argument. It also tries random solutions, as in chapter 11.

An optional final `testing::check_options` argument controls sample counts, random walks, seed and comparison tolerance. The default tolerance allows for floating-point rounding in incremental costs. The report covers:

Check	What fails it
input binding	a service bound to a different Input
check solution	the solution is not valid for the SolutionManager
repeat evaluation	evaluating the same solution twice gives non-equivalent values
random solution	random_solution returns an invalid solution
neighborhood move validity, move application	an enumerated move that is not valid, or that breaks the solution
incremental evaluation	a delta disagreeing with the full re-evaluation
delta sign	with a compare at the root of the cost expression, a cost::delta whose sign disagrees with it (chapter 2)
random proposal, random proposal application	a sampled move that is not valid, or that breaks the solution
random move in the neighborhood, availability, reproducibility	with moves that compare, a sampled move that the enumeration does not contain, a random_move that finds nothing while the neighborhood has moves, or one that does not draw from the generator given
registration names	a runner or pipeline name that is empty, repeated, or made of other characters than letters, digits, _ and -
runner parameters	a registered runner whose parameters_type is not a parameter block (chapter 8), so no frontend can change it; an empty one has nothing to configure
runner configuration, runner construction	invalid registered parameters, or a runner that cannot be built
app configuration	invalid parameters of the app, or pipeline stages without distinct names
parameter domain	a parameter of the app that declares no domain (chapter 10): give it a range, one_of, or easylocal::unlimited for any value

Test the report as a bool to check success. failures() gives diagnostics, including the starting solution and move for move-related failures. composition() counts components, neighborhoods, delta bindings and registrations.

Use print_report for readable output or pass the report to testing::run_checks alongside the component checks from chapter 11. Exceptions from hooks are reported as check failures.

Session checks: the neighborhood of a solution

While check uses sampling limits, a Session can inspect the entire neighborhood of its current solution. This is useful when investigating a specific tour:

```

// Each check enumerates the neighborhood of the current solution and
// returns a struct of counters (Session::..._result).

// The delta evaluation of each move against the full evaluation of the
// solution it leads to: moves (enumerated), mismatches (the two costs
// differ), invalid (moves that are not valid or lead to an invalid
// solution).
const auto costs = session.check_neighborhood_costs();

// What each move does to the solution: moves, null_moves (moves that leave
// it unchanged), repeated_states (moves that lead to a solution an earlier
// move reached), invalid.
const auto independence = session.check_move_independence();

// random_move against the enumeration: neighborhood_size (valid enumerated
// moves), samples (draws, 20 per move by default), out_of_neighborhood
// (draws that return no move or one the enumeration does not contain),
// unseen (enumerated moves never drawn), min_frequency and max_frequency
// (how often the least and the most drawn moves came up).
const auto sampling = session.check_random_move_distribution();

if (costs.mismatches != 0 || costs.invalid != 0 || sampling.out_of_neighborhood != 0)
    return 1;

```

Check	Result	What fails it
check_neighborhood_costs()	moves, mismatches, invalid	a move whose delta evaluation disagrees with the full evaluation, or an invalid enumerated move
check_move_independence()	moves, null_moves, repeated_states, invalid	moves that change nothing (<i>null moves</i>), and different moves that lead to the same solution
check_random_move_distribution([rounds_per_move], neighborhood_size, samples, out_of_neighborhood, unseen, min_frequency, max_frequency)		a sampled move that is not in the enumerated neighborhood; moves never sampled (unseen) hint at a biased random_move

All three need the moves to be enumerable; the last one also needs `random_move`. `check_neighborhood_costs(tolerance)` compares the costs within a `cost::tolerance` (1e-9 relative and absolute by default; {0, 0} compares exactly). The interactive tester of chapter 14 runs the same checks from its Move page: C, D and U.

The last two checks need equality. `check_move_independence` compares resulting solutions using the `SolutionManager`'s `equal` or the `Solution`'s `==`. `check_random_move_distribution` compares sampled and enumerated moves.

To reduce comparisons, independence checks group solutions by hash, or by cost when costs are ordered. Distribution checks compare moves with the same cost. Add equality to the tutorial's types to enable them:

```

// Equality of solutions and moves, for the neighborhood checks of a Session
// (chapter 15).

```

```

inline bool operator==(const Tour& a, const Tour& b)
{
    return a.order == b.order;
}

inline bool operator==(const SwapCities& a, const SwapCities& b)
{
    return a.i == b.i && a.j == b.j;
}

inline bool operator==(const TwoOpt& a, const TwoOpt& b)
{
    return a.i == b.i && a.j == b.j;
}

```

When you can edit a type, a defaulted member such as `bool operator==(const Tour&) const = default;` is shorter. Free functions also work for types you cannot change. C++20 derives `!=` from `==`.

Without equality, these two checks do not compile and the tester hides their `D` and `U` commands. Other neighborhood diagnostics remain available.

See also

- [Testing](#) for component checks.
- [Apps and tools](#).

Next steps

[Chapter 16](#) offers the same searches over HTTP.

16. A REST service

The **REST** adapter makes the same app available over HTTP. Clients can submit runs, monitor progress, cancel searches and retrieve solutions without changes to the problem components. A bounded worker pool runs searches separately from the HTTP threads.

Enable the optional REST component, based on Crow:

```
find_package(EasyLocal CONFIG REQUIRED COMPONENTS Core REST)
target_link_libraries(tsp_rest PRIVATE EasyLocal::REST)
```

Without a codec

The quickest setup reuses the I/O hooks from chapter 6. Inputs and tours travel as JSON strings containing your existing text format; this TSP's numeric costs travel as JSON numbers. No extra serialization code is needed:

```
// The Input and the tours in the text of the I/O hooks (chapter 6).
auto text_api = el::rest::blueprint("/tsp-text", application);
```

```
$ curl -X POST localhost:18080/tsp-text/runners/sa/runs \
-H 'Content-Type: application/json' \
-d '{"input": "5\n0 2 9 10 7\n2 0 6 4 3\n9 6 0 8 5\n10 4 8 0 6\n7 3 5 6 0\n",
"seed": 7}'
{"id": "1", "runner": "sa", "seed": 7, "status": "queued", ...}

$ curl localhost:18080/tsp-text/runs/1/solution
{"solution": "...", "cost": 26, ...}
```

The codec

A **codec** lets clients use structured JSON instead of text strings. Define only the conversions you need; values without a codec conversion still use their I/O hooks:

```
// Translates between JSON and the problem values.
struct TspCodec
{
    // {"distance": [[0, 2, 9, 10, 7], [2, 0, 6, 4, 3], ...]}, one array per row
    tutorial::Tsp decode_input(const crow::json::rvalue& payload) const
    {
        if (payload.t() != crow::json::type::Object || !payload.has("distance"))
            throw std::invalid_argument{"input must have 'distance'"};
        const auto& rows = payload["distance"];
        if (rows.t() != crow::json::type::List)
            throw std::invalid_argument{"'distance' must be a list of rows"};
        tutorial::Tsp tsp;
        for (const auto& row : rows)
        {
            if (row.t() != crow::json::type::List)
                throw std::invalid_argument{"'distance' must be a list of rows"};
            auto& values = tsp.distance.emplace_back();
            for (const auto& value : row)
            {
                if (value.t() != crow::json::type::Number)
                    throw std::invalid_argument{"'distance' must hold numbers"};
                values.push_back(value.d());
            }
        }
    }
};
```

```

    }
}
for (const auto& values : tsp.distance)
    if (values.size() != tsp.cities())
        throw std::invalid_argument{"'distance' must be a square matrix"};
// The 2-opt delta assumes that a reversed segment costs the same.
for (std::size_t i = 0; i < tsp.cities(); ++i)
    for (std::size_t j = 0; j < i; ++j)
        if (tsp.distance[i][j] != tsp.distance[j][i])
            throw std::invalid_argument{"'distance' must be symmetric"};
return tsp;
}

crow::json::wvalue encode_solution(
    const tutorial::Tsp&,
    const tutorial::Tour& tour) const
{
    crow::json::wvalue json;
    json["order"] = tour.order;
    return json;
}

crow::json::wvalue encode_cost(double length) const
{
    crow::json::wvalue json;
    json["length"] = length;
    return json;
}
};

```

Member	Required	Purpose
decode_input(const crow::json::rvalue&) -> Input	without read_input	the input of a run request; throw std::invalid_argument to reject it
encode_solution(const Input&, const Solution&) -> crow::json::wvalue	without write_solution	the solution of a finished run
encode_cost(const Cost&) -> crow::json::wvalue	for a cost that is not text	its cost
decode_initial_solution(const Input&, const crow::json::rvalue&) -> Solution	no	accepts an initial_solution in the request, as read_solution does from a string; without either such requests are rejected, and runs start as their start says
decode_cost(const crow::json::rvalue&) -> Cost	no	accepts a target in the request; without it a target is a JSON number, for arithmetic costs only

The service

```
auto application = el::app("tsp")
| (el::solution_manager<TourManager>() | el::component<TourLength>())
| (el::neighborhood<TwoOptExplorer>()
  | el::delta<TourLength, TwoOptLengthDelta>())
| el::runner<runners::FirstImprovement>("fi")
| el::runner<runners::SimulatedAnnealing<Classic>>(
  "sa",
  { .temperature = { .initial_temperature = 10.0, .final_temperature = 0.1 } });

// The Input and the tours in the text of the I/O hooks (chapter 6).
auto text_api = el::rest::blueprint("/tsp-text", application);

// The same app, in the JSON of TspCodec.
auto api = el::rest::blueprint(
  "/tsp",
  std::move(application),
  TspCodec{},
  el::rest::blueprint_options{
    .workers = 2,           // background search threads
    .queue_capacity = 16,   // waiting runs
    .seed = 2026,           // runs without an explicit "seed"
  });

crow::SimpleApp server;
server.register_blueprint(text_api.crow_blueprint());
server.register_blueprint(api.crow_blueprint());
// Listening on this machine only; a port that cannot be bound is an error.
auto serving = server.bindaddr("127.0.0.1").port(port).multithreaded().run_async();
server.wait_for_server_start();
if (serving.wait_for(std::chrono::milliseconds{200}) == std::future_status::ready)
{
  std::cerr << "error: cannot listen on 127.0.0.1:" << port << '\n';
  return 1;
}
serving.get(); // until a signal stops the server
```

The program is `examples/tutorial/rest_main.cpp`, built when the REST component is enabled (`-DEASYLOCAL_ENABLE_REST=ON`); a test starts it and runs the session below.

```
$ curl -X POST localhost:18080/tsp/runners/sa/runs \
  -H 'Content-Type: application/json' \
  -d '{"input": {"distance": [[0,2,9,10,7], [2,0,6,4,3], [9,6,0,8,5],
[10,4,8,0,6], [7,3,5,6,0]]}, "seed": 7}'
{"id": "1", "runner": "sa", "seed": 7, "status": "queued", ...}

$ curl localhost:18080/tsp/runs/1
{"id": "1", "status": "succeeded", "progress": {"evaluations": ...}, ...}

$ curl localhost:18080/tsp/runs/1/solution
{"solution": {"order": [...]}, "cost": {"length": 26}, ...}
```

Route	Meaning
GET /tsp/	application metadata
GET /tsp/runners	registered runner names
GET /tsp/parameters	the parameters a run may set, with their values
POST /tsp/runners/<runner>/runs	enqueue a run (202, with a Location)
GET /tsp/runs/<id>	status and progress
GET /tsp/runs/<id>/solution	the solution and its cost
POST /tsp/runs/<id>/cancel	cooperative cancellation
DELETE /tsp/runs/<id>	forget a finished run

Set `seed` to reproduce a stochastic run, including its random start. Without an explicit seed, the service uses `blueprint_options::seed` plus the run id. Runs start from a random tour unless the request supplies `initial_solution` or `"start": "initial"`.

A `target` stops the search when it reaches that cost or better. For this TSP, `"target": 26` stops at the known optimum of the five-city instance.

See also

- [REST](#): the complete HTTP contract, errors and the concurrency model.

Next steps

[Chapter 17](#) watches and stops a run from your own code.

17. Observing and controlling a run

You can monitor a search, stop it from another thread or record its decisions without changing the algorithm. Pass a control and a tracer as the final argument to the run:

```
std::stop_source stop;
auto observer = [] (const el::run_progress& progress) {
    (void)progress.evaluations; // also: iterations, evaluation_limit
};
el::run_control control{stop.get_token(), observer};
el::trace::memory_recorder<double> trace;

auto descent_search = descent.bind(tsp);
const auto observed = descent_search.run(
    descent_search.initial_solution(),
    rng,
    el::with(control, trace));
```

Control and progress. Call `stop.request_stop()` from any thread to request cancellation. Each runner checks the request through `search_run::should_stop()` and reports `termination_reason::cancelled`. The observer receives `run_progress`, with evaluations, iterations and any evaluation budget.

Tracing. A tracer receives typed events such as `run_started`, `move_evaluated`, `move_accepted`, `incumbent_updated` and `run_finished`. Other events describe local optima, neighborhood selection, visited solution hashes, and Tabu Search's aspiration, escapes and tenure changes. Solvers add `run_context` before each run. Without a tracer, event construction is removed at compile time.

Match the recorder to the cost type: `memory_recorder<double>` records this TSP's events, including events with no cost value. Pass a control, a tracer or both with `el::with(control)`, `el::with(tracer)` or `el::with(control, tracer)`.

Run limits. The same options work across algorithms:

Option	Stops when	Termination reason
<code>el::stop_at(cost)</code>	the best cost reaches the target or improves on it	<code>target_reached</code>
<code>el::timeout(5s)</code> or <code>el::timeout(2.5)</code>	the time limit expires; numeric values are seconds	<code>time_limit_reached</code>
<code>el::max_evaluations(n)</code>	the evaluation budget is ex- hausted	<code>evaluation_budget_exhausted</code>

Timeouts accept any `std::chrono` duration. The run checks the clock alongside its other stopping conditions; no timer thread is needed. An evaluation limit can tighten, but cannot raise, the runner's own budget.

Combine options as needed, for example `el::with(control, tracer).timeout(5s).stop_at(cost)`. Solvers accept them too: `solver.solve(input, el::with(control))`. A solver's time and evaluation limits cover all its runs together. Pipeline stages can also have their own limits, such as `& el::timeout(10s)` or `& el::max_evaluations(5000)`.

Use `trace::memory_recorder` to inspect events in code, `trace::jsonl_recorder` for JSON Lines, or `binary_recorder` and `async_binary_recorder` for ELTR files. `scripts/eltr.py` decodes ELTR traces; see [Tracing](#).

Diagnostic logging (`<easylocal/utils/logging.hpp>`) is separate from tracing.

See also

- [Tracing and Logging](#).
- [Runners](#).

Next steps

The [reference](#) describes every component in full; [Coming from EasyLocal 3](#) maps EasyLocal 3 to this framework.

18. Building without CMake

EasyLocal Core is header-only and depends only on the standard library: a compiler, the C++23 flag and one include path build a program. This chapter gives the whole workflow for a build driven by a makefile or by a single command — what to do before the first compilation, the makefile of the tutorial's programs, and the headers and libraries each optional component needs. CMake stays the supported way to consume the library (the [quick start](#) shows `find_package` and `add_subdirectory`); a makefile is for a course machine, an existing build system, or a program you compile by hand.

Before you start

A compiler. The same ones the [quick start](#) lists: GCC 15 or 16, Clang 22 or 23, AppleClang from Xcode 26. The flags below are the GCC and Clang spelling; with `clang-cl`, use `/std:c++latest` and `/I<dir>`.

The headers. Two sources, with the same content: the build generates no header.

- *An installation.* Configure EasyLocal once with CMake, then install it where you like:

```
cmake -S easylocal -B build/release -DCMAKE_BUILD_TYPE=Release
cmake --install build/release --prefix ~/.local
```

The headers land in `~/.local/include/easylocal/`, together with the adapters of the optional components the build enabled and with the third-party headers it fetched (see [the optional components](#)).

- *A source checkout.* `git clone https://github.com/iolab-uniud/easylocal` and point the include path at `easylocal/include`. Nothing is built, and the adapters are there too; only the third-party libraries of the optional components are missing, since a checkout fetches nothing.

The libraries of the optional components, if the program uses one: FTXUI for the TextUI, Crow for the REST service, toml++ for the TOML configuration. They are the only link-time dependency EasyLocal ever has; Core needs none.

No `-ffast-math`, which lets the compiler assume away the NaN and infinite values the library checks for, and **`-O2` or better**: the library relies on inlining, and a search can be an order of magnitude slower with `-O0`.

One command

A single-file program, such as the one of the quick start, needs no makefile:

```
c++ -std=c++23 -O2 -Wall -I$HOME/.local/include main.cpp -o tsp
./tsp
```

`-I` points at the directory that *contains* `easylocal/`, so that `#include <easylocal/easylocal.hpp>` resolves. There is nothing to link and no `-D` to define. Core starts no thread of its own; add `-pthread` if your platform requires it for the standard library, or if your program uses threads.

A makefile

The tutorial's programs are built by `examples/tutorial/Makefile`, which is the recipe for any EasyLocal program:

```
# Builds the tutorial's programs without CMake (docs/tutorial/18-building.md).
#
#     make EASYLOCAL=/usr/local                the programs of Core
#     make EASYLOCAL=/usr/local tsp_tui        one optional component
#
```

```

# EASYLOCAL is an installation prefix or a source checkout: both hold the
# headers under include/. The optional components also need their own library,
# whose prefix is FTXUI, CROW or TOMLPP.

EASYLOCAL ?= /usr/local
CXX        ?= c++
CXXFLAGS   ?= -std=c++23 -O2 -Wall -Wextra
CPPFLAGS   += -I$(EASYLOCAL)/include

FTXUI      ?= /usr/local
CROW       ?= /usr/local
TOMLPP     ?= /usr/local

# The flags of each optional component, derived from its prefix: override a
# prefix, or the flags themselves, on the command line. The third-party
# headers come in with -isystem, which leaves their own warnings out.
TUI_CPPFLAGS ?= -isystem $(FTXUI)/include
TUI_LDLIBS    ?= -L$(FTXUI)/lib -lftxui-component -lftxui-dom -lftxui-screen -pthread
REST_CPPFLAGS ?= -isystem $(CROW)/include
REST_LDLIBS   ?= -pthread
TOML_CPPFLAGS ?= -isystem $(TOMLPP)/include

# The programs of Core, built by default, and those of the optional components.
PROGRAMS := tsp tsp_cli tsp_checks
OPTIONAL  := tsp_tui tsp_rest tsp_toml

all: $(PROGRAMS)

tsp: main.o
tsp_cli: cli_main.o
tsp_checks: checks.o
tsp_tui: tui_main.o
tsp_rest: rest_main.o
tsp_toml: toml_main.o

# A target-specific variable reaches the object files of its program too.
tsp_tui: CPPFLAGS += $(TUI_CPPFLAGS)
tsp_tui: LDLIBS += $(TUI_LDLIBS)
tsp_rest: CPPFLAGS += $(REST_CPPFLAGS)
tsp_rest: LDLIBS += $(REST_LDLIBS)
tsp_toml: CPPFLAGS += $(TOML_CPPFLAGS)

$(PROGRAMS) $(OPTIONAL):
    $(CXX) $(CXXFLAGS) $(LDFLAGS) -o $@ $^ $(LDLIBS)

%.o: %.cpp tsp.hpp
    $(CXX) $(CXXFLAGS) $(CPPFLAGS) -c -o $@ $<

clean:
    rm -f $(PROGRAMS) $(OPTIONAL) *.o

.PHONY: all clean

```

From the example directory, the three programs of Core are one command away:

```

cd examples/tutorial
make EASYLOCAL=$HOME/.local
./tsp                # the searches of chapters 6 and 7

```

```
./tsp_cli --help      # the command line of chapter 12
./tsp_checks          # the component checks of chapter 11
```

Three things are worth copying into a makefile of your own:

- **CPPFLAGS carries the include path**, and the compilation rule passes it to every object file. Adding a program means adding its .o to the list; the %.o: %.cpp tsp.hpp rule rebuilds an object when the header of the problem changes, which a header-only library makes frequent. For more headers, list them there, or generate the dependencies with -MMD -MP.
- **Every translation unit uses the same flags**. EasyLocal's classes are shared across them, and compiling two of them with different standards or different -D definitions breaks the one-definition rule, usually at link time or, worse, not at all.
- **A target-specific variable** (tsp_tui: CPPFLAGS += ...) reaches the object files built for that program, so the flags of an optional component stay out of the programs that do not use it.

The optional components

Each component is a header of easylocal/adapters/ plus the third-party library it adapts. The component's own header needs no -D definition and no EasyLocal library:

Component	Header	Library	Compile	Link
Core	<easylocal/ easylocal.hpp>	—	-I<prefix>/ include	—
TextUI	<easylocal/ adapters/ tui.hpp>	FTXUI ≥ 7	-isystem <ftxui>/include	-L<ftxui>/lib - lftxui-component -lftxui-dom - lftxui-screen - pthread
REST	<easylocal/ adapters/ rest.hpp>	Crow ≥ 1.3, with standalone Asio	-isystem <crow>/ include -isystem <asio>/include	-pthread
ConfigTOML	<easylocal/ adapters/ toml.hpp>	toml++ ≥ 3.4	-isystem <toml+ +>/include	—

The three FTXUI libraries are listed in that order: component uses dom, which uses screen. Crow and toml++ are header-only; Crow includes Asio, which is header-only as well when it is the standalone version (the one EasyLocal uses).

Where the libraries come from:

- **A package manager**. FTXUI, Crow and toml++ are in Homebrew, vcpkg and several Linux distributions. The prefix is then the package manager's own (/opt/homebrew or /usr/local with Homebrew, /usr with a distribution package).
- **The EasyLocal installation itself**. A build configured with -DEASYLOCAL_FETCH_DEPENDENCIES=ON downloads what the enabled components need and cmake --install puts it in the same prefix: FTXUI under include/ftxui/ with its libraries in lib/, Crow under include/crow/, and toml++ and Asio as private copies under include/easylocal/third_party/. The prefixes of the makefile then all point at the installation:

```
make EASYLOCAL=$HOME/.local FTXUI=$HOME/.local tsp_tui
make EASYLOCAL=$HOME/.local \
    REST_CPPFLAGS="-isystem $HOME/.local/include -isystem $HOME/.local/include/
```

```

easylocal/third_party/asio" \
    tsp_rest
make EASYLOCAL=$HOME/.local \
    TOML_CPPFLAGS="-isystem $HOME/.local/include/easylocal/third_party/tomlplusplus"
\
    tsp_toml

```

- **A source build** of the library, installed wherever you like; its prefix is what the makefile wants.

The [dependency policy](#) says which versions each component accepts and why. The rest of the framework — tracing, logging, the command line, the [tuning](#) files for irace — is Core, so it needs no flag beyond the include path.

What CMake does that a makefile does not

A makefile build compiles the same code, with the same behaviour. What you give up is the checking around it:

- the C++23 and compiler-version check of `find_package(EasyLocal)`, whose message says what is missing instead of a wall of template errors;
- the check that a component you ask for is in the installation (`COMPONENTS TUI` fails when the `TextUI` was not enabled, whereas a missing header is a compiler error);
- fetching the third-party libraries, and passing their include paths and libraries to the compiler for you;
- the deployment-target check on macOS, where AppleClang needs Xcode 26 and a target of 26.0 or later for the floating-point `std::from_chars` its standard library uses.

The makefile of this chapter is built and run by the test suite (`easylocal.makefile-build`, which installs EasyLocal, compiles the quick start with one command and builds every program the installation's components allow), so the flags on this page are the flags that work.

19. Advanced: customizing Simulated Annealing

Simulated Annealing separates three decisions: how to propose a move, whether to accept it, and how to change the temperature. The neighborhood handles the first; acceptance and temperature policies handle the other two. You can change a policy while keeping the search loop, delta evaluation, best solution, run limits and tracing.

This chapter reads the actual EL4 implementation, enables its built-in reheating schedule, then writes a different reheating rule. It builds on [runners](#), [custom algorithms](#) and [parameter blocks](#). The complete program is `examples/tutorial/annealing_main.cpp`; the custom policy is in `reheat_on_rejection.hpp` beside it.

Inside the search loop

The algorithm is `runners::SimulatedAnnealing<TemperaturePolicy, Acceptance>`. Its defaults are `temperature::Classic` and `MetropolisAcceptance`. Here is its run member, taken directly from `include/easylocal/runners/simulated_annealing.hpp`. This is the implementation used by the library, including calibration and temperature tracing:

```
template<class Run, std::uniform_random_bit_generator RNG>
requires detail::simulated_annealing_context<
    typename Run::context_type, Acceptance, RNG>
[[nodiscard]]
auto run(Run& run, typename Run::solution_type solution, RNG& rng) const
{
    static_assert(
        detail::delta_agrees_with_better<typename Run::context_type>(),
        "Simulated Annealing decides with the sign of cost::delta, which"
        " must agree with better(): the compare of the cost expression "
        "finds a larger cost better");
    run.limit_evaluations(max_evaluations_);
    auto temperature = temperature_policy_;
    auto current = run.start(solution);
    if constexpr (calibrating_temperature_policy<TemperaturePolicy>)
        calibrate(run, temperature, solution, current, rng);
    temperature.reset();
    // The temperature as the trace last saw it, for a tracer that
    // observes its changes.
    constexpr bool traces_temperature =
        trace::observes<typename Run::tracer_type, trace::event::temperature_changed>;
    [[maybe_unused]] double traced_temperature = 0.0;
    if constexpr (traces_temperature)
        traced_temperature = report_temperature(run, 0.0, temperature.temperature());

    best_so_far best{solution, current.cost()};

    while (!temperature.finished() && !run.should_stop())
    {
        auto move = run.random_move(solution, rng);
        if (!move.has_value())
        {
            // No move to try: as Hill Climbing, a local optimum.
            return run.finish(
                std::move(best.solution),
                std::move(best.cost),
                termination_reason::local_optimum);
        }

        run.next_iteration();
    }
}
```

```

    auto candidate = run.evaluate_move(solution, current, *move);

    const auto accepted = static_cast<bool>(acceptance_.accept(
        candidate.cost(),
        current.cost(),
        temperature.temperature(),
        rng));

    if (accepted)
    {
        run.commit(solution, current, std::move(candidate), *move);

        best.update(run, solution, current);
    }

    temperature.on_iteration(accepted);
    if constexpr (traces_temperature)
    {
        if (temperature.temperature() != traced_temperature)
        {
            traced_temperature = report_temperature(
                run,
                traced_temperature,
                temperature.temperature());
        }
    }
}

return run.finish(std::move(best.solution), std::move(best.cost));
}

```

Read it in four parts:

1. **Start.** Apply the evaluation limit and evaluate the initial solution. Copy the temperature policy for this run. If it supports calibration, estimate the initial temperature from sampled moves, then reset the schedule.
2. **Propose and evaluate.** Draw a move from the neighborhood and evaluate it through `search_run`. This uses the deltas attached to the recipe and counts the evaluation. An empty neighborhood ends the run.
3. **Accept and remember.** Ask `acceptance_.accept` whether to keep the move. Metropolis accepts non-worsening moves and accepts a finite worsening delta with probability $\exp(-\text{delta} / \text{temperature})$. After a commit, `best.update` records an improvement of the best solution found.
4. **Advance.** Tell the schedule whether the proposal was accepted. Report any temperature change, then continue until the schedule or run control stops the search. Return the best solution, which may differ from the last visited one.

`if constexpr` selects calibration and tracing at compile time. An unused feature adds no branch to the proposal loop. The temperature and acceptance types are also known at compile time, allowing their small functions to be inlined.

The `calibrate` helper evaluates random moves at the starting solution without applying them; those evaluations consume the same budget. `report_temperature` emits `trace::event::temperature_changed`. The bookkeeping lives in `search_run`, so a new policy automatically retains cancellation, progress, evaluation limits and best-solution reporting.

First, use the built-in reheating

A colder search accepts fewer worsening moves. Reheating raises the temperature to let it explore again, while retaining the best solution found. EL4 already supplies `temperature::Reheating<Descent>`: when a descent ends, it starts another cooling phase. We can wrap `Classic` and reuse the TSP recipes from the earlier chapters:

```
const auto tsp = five_cities();
const auto sm =
    el::solution_manager<TourManager>().with_cost(el::component<TourLength>());
const auto nhe =
    el::neighborhood<TwoOptExplorer>().with_delta<TourLength, TwoOptLengthDelta>();

using Reheated = runners::temperature::Reheating<runners::temperature::Classic>;
auto scheduled = el::make_runner<runners::SimulatedAnnealing<Reheated>>(
    {
        .temperature =
            {
                .descent =
                    {
                        .initial_temperature = 10.0,
                        .final_temperature = 0.1,
                        .cooling_rate = 0.9,
                        .samples_per_temperature = 20,
                    },
                .allowed_reheats = 2,
                .reheat_ratio = 0.5,
            },
        .max_evaluations = 5000,
    })
    .with_solution_manager(sm)
    .with_neighborhood(nhe);
```

This performs the first cooling phase from 10.0, then up to two more from 5.0 ($10.0 * \text{reheat_ratio}$). Reheating changes the temperature, not the current tour. With `Classic`, each phase gets its own cooling schedule; the outer `max_evaluations` still bounds the entire run. For a budgeted descent such as `Hybrid`, the wrapper instead divides the descent's budget between the phases (see [temperature policies](#)).

The named composition above also has the pipe spelling `make_runner<Algorithm>(parameters) | sm | nhe`. Both build the same runner.

A different rule: reheat after rejected moves

Suppose we want to reheat before the current cooling phase ends, after a fixed number of consecutive rejected proposals. This is a change to the temperature policy: the search already calls `on_iteration(accepted)` after every proposal.

Our rule is:

- continue geometric cooling after every proposal;
- clear the rejection streak whenever a move is accepted;
- at the rejection threshold, restore the initial temperature and restart cooling, provided a reheat remains;
- after the reheat allowance is spent, let the final cooling phase end.

This measures **rejection**, not lack of improvement: an accepted equal-cost or worsening move also clears the streak. Reheating may help exploration, but does not guarantee a better solution. Compare schedules over several seeds with the same evaluation budget.

Describe the parameters

Reuse `ClassicParameters` as a nested group, then add the trigger and the maximum number of reheats. The schema validates the threshold and makes both settings available to configuration and tuning:

```
// The cooling schedule and the trigger for a bounded number of reheats.
struct ReheatOnRejectionParameters
{
    easylocal::runners::temperature::ClassicParameters descent{};
    easylocal::parameter<std::size_t, "rejections_before_reheat">
        rejections_before_reheat{100};
    easylocal::parameter<std::size_t, "allowed_reheats"> allowed_reheats{2};

    static constexpr auto parameter_schema()
    {
        namespace cfg = easylocal::config;
        return cfg::fields(
            cfg::group<"descent", &ReheatOnRejectionParameters::descent>(
                "The geometric cooling schedule"),
            cfg::field<&ReheatOnRejectionParameters::rejections_before_reheat>(
                "Consecutive rejected proposals before reheating",
                cfg::range(1, easylocal::unlimited)),
            cfg::field<&ReheatOnRejectionParameters::allowed_reheats>(
                "Maximum reheats (0: cooling only)",
                cfg::range(0, easylocal::unlimited)));
    }

    easylocal::config::validation_result validate() const
    {
        return easylocal::config::check_schema(*this);
    }
};
```

Implement the temperature policy

The required interface has four members: `reset`, `temperature`, `on_iteration` and `finished`. `initial_` keeps a fresh cooling schedule; `descent_` is the one currently running. The two counters track consecutive rejections and consumed reheats.

```
// Restart geometric cooling after a streak of rejected proposals.
class ReheatOnRejection : public
    easylocal::parameters_base<ReheatOnRejectionParameters>
{
public:
    explicit ReheatOnRejection(const ReheatOnRejectionParameters& parameters)
        : parameters_base{easylocal::config::require_valid(parameters)},
          initial_{parameters.descent},
          descent_{initial_}
    {
    }

    void reset()
    {
        descent_ = initial_;
        descent_.reset();
        rejected_ = 0;
        reheats_ = 0;
    }

    double temperature() const
```

```

{
    return descent_.temperature();
}

void on_iteration(bool accepted)
{
    descent_.on_iteration(accepted);
    if (accepted)
        rejected_ = 0;
    else if (reheats_ < parameters().allowed_reheats
        && ++rejected_ >= parameters().rejections_before_reheat)
    {
        descent_ = initial_;
        descent_.reset();
        rejected_ = 0;
        ++reheats_;
    }
}

bool finished() const
{
    return descent_.finished();
}

std::size_t reheats() const
{
    return reheats_;
}

private:
    easylocal::runners::temperature::Classic initial_;
    easylocal::runners::temperature::Classic descent_;
    std::size_t rejected_{0};
    std::size_t reheats_{0};
};

```

The custom logic is in `on_iteration`. It advances the underlying schedule, then decides whether to replace it with a fresh copy. A reheat can therefore also occur on the proposal that would otherwise finish cooling. If the schedule ends before the rejection threshold is reached, the search ends without using the remaining reheats.

Setting `allowed_reheats` to zero gives ordinary Classic cooling. Each phase is finite, and the reheat count is bounded; an evaluation or time limit can still stop the run earlier. Resetting the temperature never resets the run's evaluation count or discards its best solution.

Preserve optional calibration

Because the nested block exposes Classic's calibration settings, forward its optional calibration interface too. These members are also part of `ReheatOnRejection`; they were omitted from the block above for readability:

```

std::size_t calibration_samples() const
{
    return initial_.calibration_samples();
}

void calibrate(std::span<const double> deltas)
{
    initial_.calibrate(deltas);
}

```

```

    reset();
}

```

Calibration updates `initial_`, so later reheats use the estimated initial temperature. The source checks both `temperature_policy<ReheatOnRejection>` and `calibrating_temperature_policy<ReheatOnRejection>` with `static_assert`.

Run and configure the variant

Select the policy as the template argument of the existing algorithm:

```

using CustomAnnealing = runners::SimulatedAnnealing<ReheatOnRejection>;
auto custom = el::make_runner<CustomAnnealing>(
{
    .temperature =
    {
        .descent =
        {
            .initial_temperature = 10.0,
            .final_temperature = 0.1,
            .cooling_rate = 0.9,
            .samples_per_temperature = 20,
        },
        .rejections_before_reheat = 10,
        .allowed_reheats = 2,
    },
    .max_evaluations = 5000,
})

    .with_solution_manager(sm)
    .with_neighborhood(nhe);

std::mt19937_64 rng{42};
auto search = custom.bind(tsp);
const auto result = search.run(search.initial_solution(), rng);

```

The same variant can be registered in an app. Its nested schema provides paths such as `runners.reheat.temperature.rejections_before_reheat` and `runners.reheat.temperature.descent.cooling_rate`:

```

auto application =
    el::app("tsp-reheating")
        .with_solution_manager(sm)
        .with_neighborhood(nhe)
        .with_runner<CustomAnnealing>("reheat", custom.parameters());
el::Session session{application, tsp, 42};
const std::array changes{
    el::config::text_override{
        "runners.reheat.temperature.rejections_before_reheat",
        "20"},
    el::config::text_override{"runners.reheat.temperature.allowed_reheats", "3"},
};
const auto configured = session.configure(changes);
if (!configured)
    return 1;
session.use_initial_solution();
if (!session.run("reheat"))
    return 1;

```

The command line, TextUI and REST adapter can use this app as they do the earlier TSP app. Give count parameters finite ranges when exporting them for tuning ([chapter 13](#)).

Build and run the complete example:

```
cmake --build build/dev --target easylocal_tutorial_annealing
./build/dev/examples/tutorial/easylocal_tutorial_annealing
```

It runs both scheduled and rejection-triggered reheating. The five-city instance makes the mechanics easy to inspect; use larger instances to assess solution quality and speed. The tests exercise acceptance breaking a streak, the reheat limit, reset, calibration, invalid parameters and evaluation-budget handling, without relying on a lucky sequence of random moves.

When to change the loop itself

A temperature policy sees only whether a proposal was accepted. A rule based on iterations since the *best solution improved* needs more information. For that change, write a runner on `search_run`, following the loop above and [chapter 8](#). Likewise, change the acceptance policy if the decision depends on candidate cost, current cost and temperature; its member is `accept(candidate, current, temperature, rng)`.

Choose the smallest extension that has the information you need. A temperature policy is enough for this reheating rule, and keeps the rest of SA's tested implementation intact.

See also

- [Runners](#): temperature and acceptance contracts.
- [Observing and controlling a run](#): recording temperature changes and applying common run limits.