EasyLocal++ workflow

December 14, 2018

The directory `SeedProject` contains a complete EasyLocal++ project in which all methods are empty. The project can be compiled and executed as is, but obviously the execution does not produce meaningful behavior.

The naming convention of EasyLocal++ is to name every class with its role in the framework with the name of the problem as a prefix. In the seed project the name of the problem is set to `XYZ`.

The code is split into the following source files `XYZ_Data.cc`, `XYZ_Basics.cc`, `XYZ_Helpers.cc`, `XYZ_Main.cc`, and the corresponding header files `XYZ_Data.hh`, `XYZ_Basics.hh`, `XYZ_Helpers.hh` (the file `XYZ_Main.cc` defines only the `main` function, and thus has no header).

The first step is therefore to copy the seed project and to replace the string `XYZ` in all files (including `makefile`) with the name (or the acronym) of our problem.

The development consists in filling the about files in the following sequence.

1 `Data` file

This file contains `Input` and `Output` classes, for representing the input data and a solution of an instance, respectively. These classes are provided to the framework as templates, therefore they do not inherit form framework classes.

The should contain

- `Input` class:
  - constructor that takes the file name, reads its content and stores it into appropriate data structures
  - data structures and corresponding getters
- `Output` class:
  - constructor that takes a reference to an `Input` object, stores it in the member `in`, and resizes the data structures
  - data structure for storing the solution and corresponding getters

2 Basics file

- Define `State` class:
  - main data structures
  - redundant data structures (invariant) [for accelerating delta evaluations]
  - assignment operator `=` [for copying objects (in presence of constant members)]
  - equality operator `==` [for debugging]
  - constructor that takes input as parameter
- Define `Move` class: [This step must be repeated if more than one move is defined]
  - move attributes
  - constructor that initializes the attributes
  - operators: `==`, `!=`, `<` [for the tester]

3 Helpers file

- Define `StateManager` class:
  - `RandomState`
  - `CheckConsistency` [optional]
- Define `OutputManager` class:
  - `InputState`
  - `OutputState`
- Define `NeighborhoodExplorer` class:
  - `FirstMove`
  - `NextMove`
  - `RandomMove`
  - `MakeMove`
  - `FeasibleMove` [optional]
- Define `CostComponent` classes (one for each cost component):
  - `ComputeCost`
  - `PrintViolations`
- Define `DeltaCostComponent` classes (one for each cost component):
  - `ComputeDeltaCost`

4 Main file

- Write the `main()` function including:
  - definition of command-line parameters
  - declaration of the `Input` object
  - declaration of helper objects
  - links between helper objects (for example, see `AddCostComponent`)
  - declaration of runner, solver, and tester objects
  - invocation tester or solver