In Hypercubo Nigrae Capsulae Optimum

Arnaud Berny
arnaud@courros.fr

ABSTRACT

HNCO consists of a C++ library, command-line tools, and scripts for the optimization of black box functions defined on fixed-length bit vectors. It aims at being flexible, fast, simple, and robust. The library provides classes for functions, populations, neighborhoods, and algorithms. It currently includes 22 concrete functions and 18 concrete algorithms. The command-line tools expose most of the library to the user without the need for programming. One of the goals of HNCO is to automate experiments and favor reproducible research. HNCO comes with experiments designed to tune or compare algorithms. Scripts run all the simulations in an experiment and generate a report. The source code of HNCO is published under the GNU LGPL 3 license.

CCS CONCEPTS

•Mathematics of computing → Optimization with randomized search heuristics; •Computing methodologies → Discrete space search; •Software and its engineering → Object oriented frameworks;

KEYWORDS

Black box optimization; bit vectors; evolutionary algorithms; local search; C++ framework; benchmarking

ACM Reference format:
Arnaud Berny. 2017. In Hypercubo Nigrae Capsulae Optimum. In *Proceedings of GECCO '17 Companion, Berlin, Germany, July 15-19, 2017,* 8 pages.
DOI: http://dx.doi.org/10.1145/3067695.3082472

1 INTRODUCTION

Experimental study of algorithms for the optimization of black box functions is difficult. Considerable effort is spent in the development and test of algorithms. It involves different skill sets and very often is conducted by individuals or small teams, at least in early stage research. Sometimes knowledge of algorithms or functions is spread across different scientific or engineering fields spanning many decades. Hence frameworks are needed to help developers and users of these algorithms. They usually provide collections of functions, operators, and algorithms. See [29] for a survey of black box (or metaheuristic) optimization frameworks.

HNCO consists of a C++ library, command-line tools, and scripts for the optimization of black box functions defined on fixed-length bit vectors. Only unconstrained single objective optimization is addressed. HNCO aims at being flexible, fast, simple, and robust. The source code of HNCO [7] is published under the GNU LGPL 3 license. The library offers classes for functions, algorithms, maps (e.g. permutations of bit vector components), populations, and neighborhoods for random and exhaustive local search. Functions and maps can be composed. The population class provides experimental support for parallel evaluation of bit vectors. The library implements many evolutionary algorithms but is by no means restricted to evolutionary computation. It includes simulated annealing and other local search algorithms. The command-line tools expose most of the library to the user without the need for programming.

One of the goals of HNCO is to automate experiments. An experiment in HNCO is a list of simulations specified in a plan file. Scripts process this file, run the simulations, and generate a report. HNCO comes with a few experiments and their respective scripts. These experiments are designed to tune and compare algorithms. By providing collections of functions, algorithms, and experiments, HNCO aims at favoring reproducible research. Another goal of HNCO is to help bridge the gap between theory and practice. Experiments can be used to make conjectures about runtimes or reproduce theoretical results. Finally, algorithms in the library can also be applied to practical problems.

HNCO departs from most black box optimization frameworks in that it is restricted to bit vectors. It is also smaller and more limited in scope than they are. Other C++ frameworks include ParadisEO [11], EasyLocal++ [13], and Mallba [1]. HNCO shares some of the goals of COCO [16] in the performance assessment of black-box algorithms.

The article is organized as follows. The library is presented in Sec. 2. The command-line tools are presented in Sec. 3. Sec. 4 briefly explains how the library and the tools are built. Sec. 5 presents experiments and fragments of generated reports. Sec. 6 points at some current limitations of HNCO and concludes the article.

2 LIBRARY

The library is organized into hierarchies of classes for exceptions, maps, functions, populations, neighborhoods, iterators, and algorithms. The library is documented with Doxygen.

2.1 Namespaces

The library declares the top-level namespace `hnco` and the nested namespaces `hnco::random`, `hnco::function`, `hnco::algorithm`, `hnco::exception`, and `hnco::neighborhood`.

2.2 Data structures

2.2.1 Bit vector. The library defines the type `bit_vector_t` as `vector<char>`. We have also considered `vector<bool>` (standard library) and `dynamic_bitset` (Boost library). We have compared their runtime with various functions, sizes, and algorithms. It appears that `vector<char>` is always the fastest implementation. The other two implementations are 5% to 70% slower, with the majority

GECCO '17 Companion, Berlin, Germany

DOI: http://dx.doi.org/10.1145/3067695.3082472

of overheads between 10% and 30%. Apart from the fact that `vector <char>` uses more memory than the other two implementations, it is clear that some algorithms could benefit from `vector<bool>` or `dynamic_bitset`, in particular those intensively using bit-wise operations. However, there is currently no such algorithm in the library. Finally, `vector<char>` provides iterators, which allows to use algorithms in the standard C++ library. This is not the case for `vector<bool>` or `dynamic_bitset`.

The library provides functions for bit vectors to flip a single bit, compute the Hamming weight of a bit vector, add two bit vectors etc. All such functions are prefixed by `bv_`.

The library defines the type `permutation_t` as `vector<int>`. The corresponding functions are prefixed by `perm_`.

The library defines the type `point_value_t` as a `std::pair` made of a `bit_vector_t` and a `double`.

2.2.2 Bit matrix. The library offers basic support for linear algebra on bit vectors. It defines the type `bit_matrix_t` as a vector of `bit_vector_t`. The corresponding functions are prefixed by `bm_`. The library provides functions to resize or transpose a bit matrix, add rows, swap rows, solve a linear system, invert a square bit matrix, multiply a bit matrix and a bit vector etc.

2.3 Exceptions

All exceptions derive from the class Exception which is declared in the namespace `hnco::exception`.

- Error
- LastEvaluation is thrown by a function when the budget has been spent.
- PointValueException declares a `point_value_t` data member. It has four derived classes:
  - MaximumReached is thrown by a function when it has evaluated a bit vector to its known maximum (see Sec. 2.5.2).
  - TargetReached is thrown by a function when it has evaluated a bit vector to a value greater or equal to some given target (see Sec. 2.5.2).
  - LocalMaximum is thrown by an algorithm when it has detected a local maximum (exact or approximate).

2.4 Maps

Map classes implement various transformations from bit vectors to bit vectors which can be freely composed. They prove themselves most useful when checking invariance properties of algorithms.

The class Map is declared in the namespace `hnco`. It defines the following methods:

- `void map(const bit_vector_t&, bit_vector_t&)`
- `size_t get_input_size()`
- `size_t get_output_size()`
- `bool is_surjective()`

The following concrete maps are defined in the library:

- AffineMap is defined by $\varphi(x) = Ax + b$, where A is a $n \times n$ bit matrix, b is a n-dimensional bit vector, and operations are understood modulo 2.
- LinearMap is defined by $\varphi(x) = Ax$, where A is a $n \times n$ bit matrix. It is a special case of AffineMap.
- Translation is defined by $\varphi(x) = x + b$, where b is a n-dimensional bit vector. It is a special case of AffineMap.
- Permutation is defined by $\varphi(x) = y$, where $y_i = x_{\sigma_i}$, and σ is a permutation of $0, \ldots, n-1$ (indices start at 0).
- MapComposition implements the composition $\varphi \circ \psi$ of φ and ψ; it requires their dimensions to be compatible.

Affine and linear maps are declared not surjective; the correct answer depends on the rank of A and will be implemented in a future release of the library. Translations and permutations are surjective. The composition of two maps is surjective if both of them are surjective.

2.5 Functions

2.5.1 Abstract function. The abstract class Function is declared in the namespace `hnco::function`. It defines the following methods:

- `size_t get_bv_size()` returns the bit vector size, useful for example if the given instance has been read from a file.
- `double eval(const bit_vector_t&)`
- `double safe_eval(const bit_vector_t&)`
  This method must not throw any exception as it is called during parallel evaluation.
- `void update(const bit_vector_t&, double)`
  This method updates the object state after a safe evaluation.
- `bool has_known_maximum()`
- `double get_maximum()`

Functions are defined for all $x \in \{0,1\}^n$.

2.5.2 Function decorators. The class FunctionDecorator derives from Function. We first present function decorators which modify decorated functions. Let $f : \{0,1\}^n \to \mathbb{R}$ denote the decorated function.

- AdditiveGaussianNoise implements $f + \mathcal{N}(0, \sigma)$. Successive calls to `eval` add samples from independent and identically distributed random variables with centered Gaussian distribution of standard deviation σ.
- Negation implements $-f$. This is useful when minimizing a function since algorithms all maximize some function.
- FunctionMapComposition implements $f \circ \varphi$ where $\varphi : \{0,1\}^m \to \{0,1\}^n$ is a map (see Sec. 2.4). The decorator and the decorated function can have different sizes. It should be noted that $f \circ \varphi$ has a known maximum only if f has a known maximum and φ is a surjective map.

The library also defines function decorators which do not modify the decorated function. They are useful as control flow statements or to collect information on the trajectory taken by an algorithm.

- CallCounter derives from FunctionDecorator. It counts the number of calls to the method `eval` of the decorated function.
- ProgressTracker derives from CallCounter. It keeps track of the maximum so far and of the time when it has been found. This information can be sent to the standard output or any output stream.
- OnBudgetFunction derives from CallCounter. If some given budget (number of calls) has been spent then it throws a LastEvaluation exception.

- StopOnMaximum derives from FunctionDecorator. It throws an exception MaximumReached as soon as it has evaluated a bit vector to its known maximum. A error is thrown by the constructor if the maximum of the decorated function is not known.
- StopOnTarget derives from FunctionDecorator. It throws an exception TargetReached as soon as it has evaluated a bit vector to a value greater or equal to some given target.
- Cache derives from FunctionDecorator. It implements a simple memoization technique in which the first call `eval(x)` stores the ordered pair $(x, f(x))$ in an associative array. Subsequent calls `eval(x)` with the same argument x retrieve the value $f(x)$ from the associative array and avoid its actual computation. Depending on function complexity, caching values can be beneficial.

2.5.3 *Concrete functions.*

Functions without instance files.

- OneMax [19].
- LeadingOnes [19]
- Ridge [19]
- Needle (needle in a haystack) [19]
- Jump or BigJump in [26]
- DeceptiveJump or $\textsc{Jump}_k$ in [19]
- FourPeaks [2]
- SixPeaks [8]
- Labs (low autocorrelation binary sequence) [24]
- Cancellation (summation cancellation) [3]
- SinusCancellation (summation cancellation with sinus) [30]
- Trap [12]
- Hiff (hierarchical if and only if) [19]
- Plateau [19]
- LongPath [19]
- Plugin allows to load a dynamic shared object and call a function loaded into memory and identified by its name. Its prototype must be of the form:

```
double f(const char[], size_t);
```

Functions with instance files. Serialization is mostly achieved through the Boost serialization library [9] with text archives. For problems with standard file formats, custom `load` and `save` methods have been developed.

The library provides the following functions:

- MaxSat [28]. Dimacs file format [23] is used to load and save instances.
- Qubo (quadratic unconstrained binary optimization) [10] implements $f(x) = \sum_i q_{ii}x_i + \sum_{i<j} q_{ij}x_ix_j$. The dimacs-like file format specified in the reference is used to load and save instances.
- LinearFunction implements $f(x) = \sum_i a_ix_i$ where a_i are real numbers. It uses the Boost serialization library.
- QuadraticFunction implements $f(x) = \sum_i a_i(2x_i - 1) + \sum_{i<j} q_{ij}(2x_i - 1)(2x_j - 1)$. It uses the Boost serialization library.
- NkLandscape [20]. It uses the Boost serialization library.
- EqualProducts [3]. It uses the Boost serialization library.
- Factorization recasts the problem of factorizing some given integer z into a maximization problem. It is defined by $f(x) = -\sum_i(i+1)(y_i \oplus z_i)$ where z_i is the ith binary digit of z, $y = u \times v$, and x is the concatenation of u and v. This particular class uses the GNU Multiple Precision Arithmetic Library [15]. The number to factorize, written in decimal, is loaded from a text file.

2.6 Population

The class Population hides away the details of evaluating a population of bit vectors with a given function, sorting the population, and performing selection. It defines the following methods:

- `void random()` initializes the population with random bit vectors.
- `void eval(Function *)` implements sequential evaluation of the population.
- `void eval(const vector<Function *>& functions)` This method implements parallel evaluation of the population.
- `void sort()` sorts the population without copying any bit vector.
- `const bit_vector_t& get_best_bv(int i)` This method returns a reference to the ith fittest bit vector in the population.
- `void plus_selection(const Population&)` This method implements plus-selection in the sense of evolutionary algorithms [19].
- `void comma_selection(const Population&)` This method implements comma-selection in the sense of evolutionary algorithms [19].

The library also provides the class `TournamentSelection` which derives from Population.

2.7 Neighborhoods

Neighborhood classes are used by random local search algorithms, including simulated annealing. For each bit vector x, a neighborhood defines an implicit set of bit vectors, usually not containing x itself, and a distribution on this set. A local search algorithm can then repeatedly ask the neighborhood for a random candidate bit vector.

The abstract class Neighborhood is declared in the namespace `hnco::neighborhood`. It defines the following methods:

- `void set_origin(const bit_vector_t&)`
- `void propose()`
- `const bit_vector_t& get_candidate()`
- `void keep()`
- `void forget()`

The following concrete neighborhoods are defined in the library:

- SingleBitFlip. Exactly one bit of x is flipped.
- Binomial. Every bit of x is flipped with a given probability until at least one bit has been flipped.
- HammingSphere. The candidate is uniformly sampled from the set of bit vectors y such that $d_H(x, y) = r$, where r is

the radius of the Hamming sphere: exactly r bits of x are flipped.
- HammingBall. Given the radius r of the ball, exactly k bits of x are flipped, where k is uniformly sampled between 1 and r.

2.8 Iterators

Iterators allow to enumerate implicit sets of bit vectors. They are used by the complete search algorithm which enumerates the whole hypercube and by local search algorithms such as hill climbers which enumerate neighborhoods.

The abstract class Iterator is declared in the namespace `hnco`. It defines the following methods:

- `void init()`
- `const bit_vector_t& get_current()`
- `bool has_next()`
- `void next()`

The following concrete iterators are defined in the library:

- HypercubeIterator
- SingleBitFlipIterator
- HammingBallIterator

2.9 Algorithms

2.9.1 Abstract algorithm. All algorithms in the library maximize some given function. They are responsible for keeping track of the solution so far, as decorated functions can fulfill the same task but only globally.

The abstract class Algorithm is declared in the namespace `hnco::algorithm`. It defines the following methods:

- `void set_function(Function *)`
- `void set_functions(const vector<Function *>)`
- `void init()`
- `void maximize()`
- `const point_value_t& get_solution()`

The class IterativeAlgorithm derives from Algorithm and declares the additional methods `iterate` and `log`.

2.9.2 Concrete algorithms. The library provides the following concrete algorithms:

- CompleteSearch
- RandomSearch
- NonStrictRandomLocalSearch
- StrictRandomLocalSearch
- SteepestAscentHillClimbing
- SimulatedAnnealing [21]
- OnePlusOneEa, (1 + 1) evolutionary algorithm [19]
- MuPlusLambdaEa, $(\mu + \lambda)$ EA [19]
- MuCommaLambdaEa, (μ, λ) EA [19]
- GeneticAlgorithm [18]
- Pbil, population-based incremental learning [2]
- NpsPbil, PBIL with negative and positive selection [4]
- Umda, univariate marginal distribution algorithm [25]
- CompactGa, compact genetic algorithm [17]
- NonStrictMmas, non strict max-min ant system [31]
- StrictMmas, strict max-min ant system [31]
- BmPbil, Boltzmann machine PBIL [5]
- Hea, herding evolutionary algorithm [6]
- Restart is actually an algorithm decorator which indefinitely restarts its decorated algorithm.

2.10 Parallel evaluation

The library provides experimental support for parallel evaluation of bit vectors. Parallel evaluation is implemented in the class Population with OpenMP [27]. More precisely, it uses a parallel for loop which automatically distributes iterations across a predefined number of threads. The main objective is that parallel and sequential evaluations produce the same effects.

To achieve thread-safety, each thread calls its own copy of the function. However, this is insufficient because we have seen that function decorators update their states and even throw exceptions. This is why each thread calls `safe_eval` instead of `eval`. At the end of the parallel loop, all side-effects (including exceptions) are accounted for by calling `update` on a single copy of the function. In the base class Function, `safe_eval` simply calls `eval` and `update` does nothing. They are redefined only in decorators.

The following algorithms can use parallel evaluation: MuPlusLambdaEa, MuCommaLambdaEa, Umda, GeneticAlgorithm, Pbil, NpsPbil, Hea, BmPbil.

2.11 Example

We give a complete example of how to use the library, including a custom function and a custom algorithm. The different parts of the source code below should be merged into a single file to be compiled and linked against the library as indicated in Sec. 4.

The source code starts with header inclusions and namespace declarations:

```
#include <iostream>
#include <hnco/algorithms/algorithm.hh>
#include <hnco/functions/function.hh>
using namespace hnco::algorithm;
using namespace hnco::function;
using namespace hnco::random;
using namespace hnco;
```

The following custom function implements the function defined by $f(x) = \sum_{i=1}^{n} i \cdot x_i$:

```
class CustomFunction: public Function {
  size_t _bv_size;
public:
  CustomFunction(int bv_size):
    _bv_size(bv_size) {}
  size_t get_bv_size() { return _bv_size; }
  double eval(const bit_vector_t& x) {
    double result = 0;
    for (size_t i = 0; i < _bv_size; i++)
      if (x[i])
        result += i + 1;
    return result;
  }
};
```

The custom algorithm implements an iterative random search. Only the method `iterate` must be defined. The base class Algorithm defines a default method `init` which randomly sets the solution. It also defines many `update_solution` methods. In particular, `update_solution(_candidate)` evaluates the candidate and sets it as the new solution provided its value is strictly greater than that of the current solution.

```
class CustomAlgorithm: public IterativeAlgorithm {
  bit_vector_t _candidate;
protected:
  void iterate() {
    bv_random(_candidate);
    update_solution(_candidate);
  }
public:
  CustomAlgorithm(int n):
    IterativeAlgorithm(n),
    _candidate(n) {}
};
```

Finally, the custom algorithm is applied to the custom function. Observe that the custom function is decorated with a progress tracker which logs improvement to the standard output. At the end of the search, the program prints the solution found by the custom algorithm.

```
int main()
{
  Random::engine.seed(0); // Or entropy source

  const int bv_size = 100;
  CustomFunction function(bv_size);
  ProgressTracker tracker(&function);

  CustomAlgorithm algorithm(bv_size);
  algorithm._num_iterations = 1000;
  algorithm.set_function(&tracker);
  algorithm.init();
  algorithm.maximize();

  point_value_t solution =
    algorithm.get_solution();
  bv_display(solution.first, std::cout);
  std::cout << std::endl;
  return 0;
}
```

3 COMMAND LINE TOOLS

HNCO comes with the command-line tools `ffgen`, `mapgen`, and `hnco`. With the flag `--help`, they print a list of available options. Auto-completion is available for bash.

3.1 ffgen

`ffgen` generates random instances of functions and saves them to files which can be later loaded by `hnco` and used by any algorithm.

For example, to generate a random instance of Nk landscape with $n = 100$ and $k = 4$ and save it to the file `nk.100.4`, run the command:

```
ffgen -s 100 -F 60 --nk-k 4 --path nk.100.4
```

3.2 mapgen

`mapgen` generates random instances of maps and saves them to files. It can generate a translation, a permutation, the composition of a permutation and a translation, a linear map, or an affine map (see Sec. 2.4).

Random maps are useful to check whether a given algorithm is invariant under the action of those maps. `hnco` can generate a random map instance but subsequent runs get different instances. However, it might be necessary to ensure that different algorithms share the same instance previously generated by `mapgen`.

3.3 hnco

`hnco` allows to apply any algorithm to any function in the library. Nearly every parameter can be set with an option at the command-line. An option takes the form of a flag or a key-value pair; there is a default value for every key. This simple command-line interface makes it easy to use `hnco` inside a script. Results are written to the standard output. Warning and error messages are written to the standard error stream. By default `hnco` writes to the standard output the values of all parameters as comments. The same simulation (including the same seed for random numbers) can then be run again later.

For example, to apply (1+1) EA to the previous Nk landscape instance, run the command

```
hnco -A 300 -F 60 --path nk.100.4 \
-b 200000 --print-performance
```

where we have set the budget to 200000 function evaluations.

For the purpose of scripting, `hnco` returns an exit status with the following meaning: 0 indicates success; 1 indicates a runtime error; 2 indicates that the flag `--stop-on-maximum` has been set and the maximum has not been reached; 3 indicates that the flag `--target` has been set and the target has not been reached.

4 BUILDING THE SYSTEM

The library has been developed under Linux Ubuntu 16.04 but should be successfully built under many unix-like operating systems. It uses the Boost serialization library, libdl (for plugin), and libgmp (for factorization, see Sec. 2.5.3). The compiler must implement the OpenMP API (for parallel evaluation, see Sec. 2.10).

Autotools are used to build the system. The package comes with 31 tests which cover functions with known maximum (using complete search), serialization, linear algebra, and steepest ascent hill climbing (applied to OneMax).

5 EXPERIMENTS

HNCO provides four experiments. For each experiment, there is a script for running the simulations and another one for computing the statistics. The scripts are written in Perl although Python is considered for future developments.

The script for computing the statistics generates a few gnuplot scripts and a LaTeX file. The gnuplot scripts in turn generate graphics files in eps, pdf, and png formats. The LaTeX and graphics files are finally compiled into a single pdf report. The whole process is controlled by a make file.

A JSON plan file specifies the experiment, that is the set of algorithms, the set of functions, their respective parameters, and various options. Every experiment discussed hereafter has a plan file which can be found in its own directory under `experiment`.

Some problems are originally expressed in terms of minimization, for example min g, then recast as the maximization problem max $-g$ to fit into HNCO. The scripts can apply a symmetry to the data before generating the tables and the graphics. They can also use a logarithmic scale to better represent performance of algorithms. Finally, the user can control how numbers are rounded and printed in the tables.

5.1 Benchmark

The purpose of this experiment is to compare the performance of a set of algorithms applied to a set of functions. Every algorithm is run the same number of times on every function. Algorithms are ranked according to their median performance (other quartiles are also considered for tiebreak) on every function (see Tab. 1). They are also globally ranked according to their rank distribution (see Tab. 2). Graphics similar to Fig. 1 and 2 are generated for every function.

Here is a fragment of the plan file showing two functions and two algorithms. The value of the key `col` represents the number format used in LaTeX tables such as Tab. 1. The plan file is read by a Perl script which interprets the backslash as an escape character hence the need for a double backslash sequence.

```
{ "exec": "hnco",
  "opt": "--no-header --print-performance --map 1
      --map-random -s 100 -i 0 -b 300000",
  "num_runs": 20,
  "results": "results",
  "graphics": "graphics",
  "report": "report",
  "functions": [
  { "id": "one-max",
    "opt": "-F 0 --stop-on-maximum",
    "col": ">{{\\nprounddigits{0}}}N{3}{0}" },
  { "id": "leading-ones",
    "opt": "-F 10 --stop-on-maximum",
    "col": ">{{\\nprounddigits{0}}}N{3}{0}" } ],
  "algorithms": [
  { "id": "rls",
    "opt": "-A 100 --restart" },
  { "id": "ea-1p1",
    "opt": "-A 300" } ]
}
```

5.2 Dynamics

The purpose of this experiment is to visualize the dynamics of the performance of a set of algorithms applied to a set of functions.

algorithm	performance						time (s)
	min	Q_1	med.	Q_3	max	rk	mean
rls	4.17	4.39	4.44	4.52	4.91	5	1.65
hc	4.46	4.59	4.81	4.88	5.31	2	1.66
sa	4.47	4.63	4.76	5.08	5.75	3	1.68
ea-1p1	3.54	3.81	4.03	4.28	4.87	8	2.27
ea-1p10	3.71	3.89	4.13	4.35	4.73	7	2.02
ea-10p1	4.16	4.49	4.60	4.69	4.80	4	2.04
ea-1c10	4.57	4.79	4.85	4.90	5.05	1	2.03
ga	3.64	4.05	4.33	4.48	5.11	6	2.55
pbil	3.50	3.68	3.89	4.07	4.60	10	2.16
umda	3.51	3.86	3.91	4.10	4.41	9	2.05

Table 1: Algorithms ranked according to their performance on Labs (20 runs). Labs stands for low autocorrelation binary sequence, rls for random local search, hc for steepest ascent hill climber, sa for simulated annealing, ea-1p1 for (1+1) evolutionary algorithm, ea-1p10 for (1+10) EA, ea-10p1 for (10+1) EA, ea-1c10 for (1,10) EA, ga for genetic algorithm, pbil for population-based incremental learning, umda for univariate marginal distribution algorithm. Best results are in blue.

algorithm	rank distribution									
	1	2	3	4	5	6	7	8	9	10
pbil	10	0	1	2	2	0	1	1	0	2
sa	8	2	3	2	0	1	0	0	2	1
umda	7	2	1	0	2	0	2	1	3	1
rls	6	4	2	2	1	1	0	1	0	2
ga	6	2	1	0	1	3	5	0	0	1
ea-1c10	5	5	3	5	0	0	0	0	1	0
hc	5	5	1	2	1	0	1	2	0	2
ea-1p1	5	3	1	2	1	0	3	3	0	1
ea-10p1	4	2	5	5	0	2	1	0	0	0
ea-1p10	4	2	2	2	0	1	4	0	3	1

Table 2: Algorithms ranked according to their rank distribution. For example, PBIL is ranked #1 ten times and #5 twice.

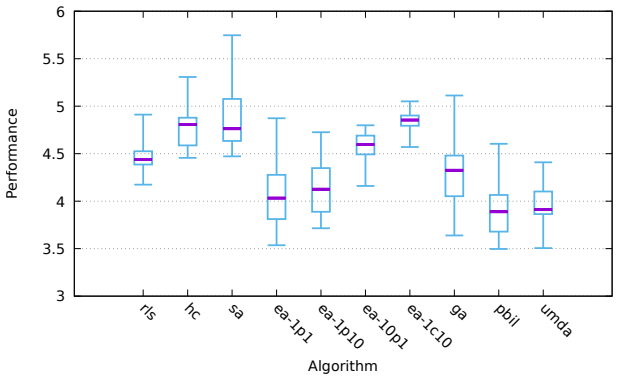

Figure 1: Performance of algorithms applied to Labs (20 runs).

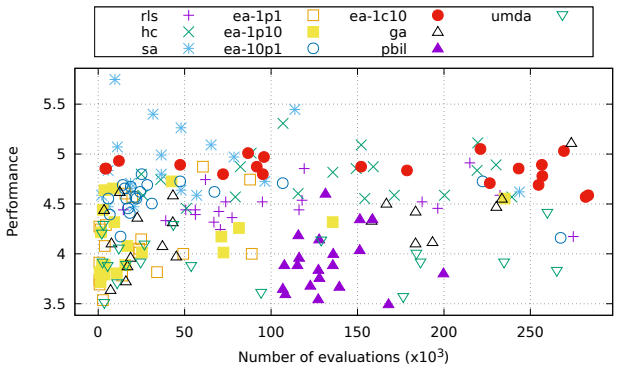

Figure 2: Performance vs number of evaluations for algorithms applied to Labs (20 runs).

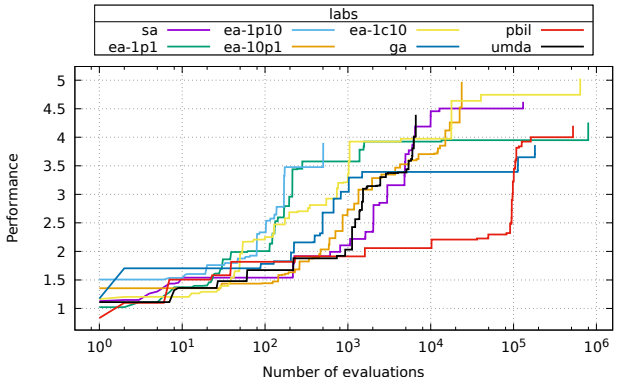

Figure 3: Dynamics of the performance of algorithms on Labs (single run). The end of a line indicates that an algorithm has not improved its solution any further within the allocated budget.

Each algorithm is run once on each function. See Fig. 3 for an example.

5.3 Parameter

The purpose of this experiment is to study the influence of a single parameter on the performance of a given algorithm applied to a set of functions. For each value (taken in a list) of the parameter, the algorithm is run the same number of times on every function. The report is similar to the one presented in Sec. 5.1. For every function, the mean, standard deviation, and quartiles of the performance of the algorithm as a function of the parameter are plotted as in Fig. 4-5.

5.4 Runtime

The purpose of this experiment is to study the influence of a single parameter on the runtime of a set of algorithms applied to a set of functions. By runtime is meant the number of function evaluations needed to reach the maximum of a function which must then have a known maximum as seen in Sec. 2.5.2. For every function and

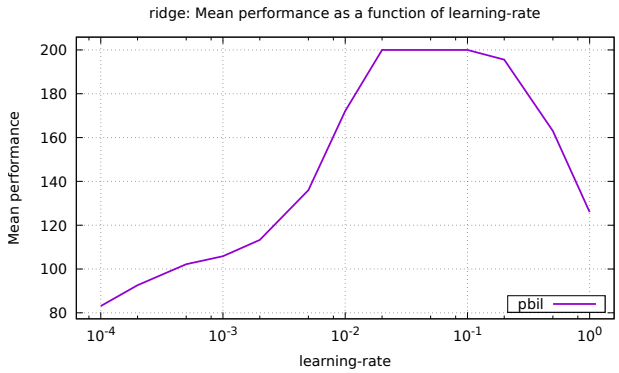

Figure 4: Mean performance of PBIL applied to Ridge as a function of the learning rate (20 runs).

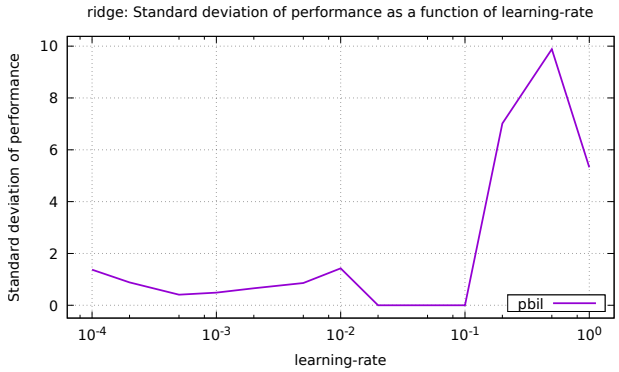

Figure 5: Standard deviation of the performance of PBIL applied to Ridge as a function of the learning rate (20 runs).

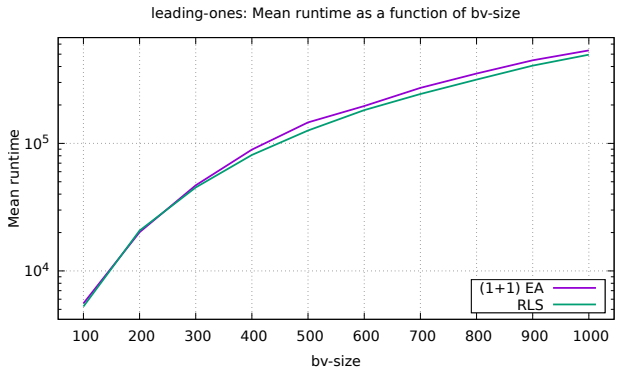

Figure 6: Mean runtime of (1+1) EA and RLS applied to LeadingOnes as a function of size (20 runs).

every algorithm, the mean, standard deviation, and quartiles of the runtime as a function of the parameter are plotted as in Fig. 6-7.

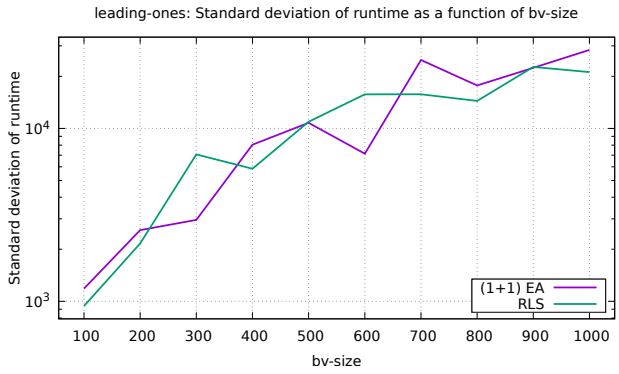

Figure 7: Standard deviation of the runtime of (1+1) EA and RLS applied to LeadingOnes as a function of size (20 runs).

6 LIMITATIONS AND FUTURE DEVELOPMENTS

The restriction to fixed-length bit vectors might appear as the greatest limitation of HNCO. At the same time, it offers the opportunity to experiment with and directly compare a great diversity of algorithms and functions, as most black box algorithms can be expressed in terms of bit vectors.

Any additional algorithm enriches the library and can be tested against all the functions already in the library, thus increasing the significance of the results. Two important classes of algorithms are missing in the library and should be added to it in a future release: Tabu search [14] and estimation of distribution algorithms (EDA) [22]. Additional functions are also needed.

An algorithm in HNCO is responsible for producing a single solution. It could be desirable to also get a set of solutions. For example, evolutionary algorithms could give access to their population.

As noted before, support for parallel evaluation is still experimental. Beyond the case of populations in evolutionary algorithms, algorithms yet to be developed could benefit from parallel evaluation.

One of the most pressing issues related to experiments is to make scripts run independent simulations in parallel. This should be addressed in a future release.

The content and presentation of reports could be improved. Beyond algorithm ranking and simple descriptive statistics, statistical techniques are available for the performance assessment of black-box algorithms. In particular, empirical cumulative distribution functions of runtimes as used in COCO [16] are not restricted to continuous optimizers and should be included in reports generated by HNCO.

REFERENCES

[1] Enrique Alba, Gabriel Luque, Jose Garcia-Nieto, Guillermo Ordonez, and Guillermo Leguizamon. 2007. MALLBA: a Software Library to Design Efficient Optimisation Algorithms. *Int. J. Innov. Comput. Appl.* 1, 1 (April 2007), 74–85. DOI: http://dx.doi.org/10.1504/IJICA.2007.013403

[2] S. Baluja and R. Caruana. 1995. Removing the genetics from the standard genetic algorithm. In *Proceedings of the 12th Annual Conference on Machine Learning*. 38–46.

[3] S. Baluja and S. Davies. 1997. *Using optimal dependency-trees for combinatorial optimization: learning the structure of the search space.* Technical Report CMU-CS-97-107. Carnegie-Mellon University.

[4] Arnaud Berny. 2001. Extending selection learning toward fixed-length d-ary strings. In *Artificial Evolution (Lecture Notes in Computer Science)*, P. Collet and others (Eds.). Springer, Le Creusot.

[5] Arnaud Berny. 2002. Boltzmann machine for population-based incremental learning. In *ECAI 2002*. IOS Press, Lyon.

[6] Arnaud Berny. 2015. Herding Evolutionary Algorithm. In *Proceedings of the Companion Publication of the 2015 Annual Conference on Genetic and Evolutionary Computation (GECCO Companion '15)*. ACM, New York, NY, USA, 1355–1356.

[7] Arnaud Berny. 2017. *HNCO version 0.5.* http://github.com/courros/hnco Last accessed April 26, 2017.

[8] J. S. De Bonet, C. L. Isbell, and P. Viola. 1996. MIMIC: finding optima by estimating probability densities. In *Advances in Neural Information Processing Systems*. Vol. 9. MIT Press, Denver.

[9] Boost 2017. *Boost C++ Libraries.* http://www.boost.org/ Last accessed April 26, 2017.

[10] Michael Booth, Steven P. Reinhardt, and Aidan Roy. 2017. *Partitioning Optimization Problems for Hybrid Classical/Quantum Execution.* Technical Report. D-Wave.

[11] S. Cahon, N. Melab, and E.-G. Talbi. 2004. ParadisEO: A Framework for the Reusable Design of Parallel and Distributed Metaheuristics. *Journal of Heuristics* 10, 3 (2004), 357–380. DOI: http://dx.doi.org/10.1023/B:HEUR.0000026900.92269.ec

[12] Kalyanmoy Deb and David E. Goldberg. 1993. Analyzing Deception in Trap Functions. In *Foundations of Genetic Algorithms 2*, L. Darrell Whitley (Ed.). Morgan Kaufmann, San Mateo, CA, 93–108.

[13] Luca Di Gaspero and Andrea Schaerf. 2003. EasyLocal++: an object-oriented framework for the flexible design of local-search algorithms. *Software: Practice and Experience* 33, 8 (2003), 733–765. DOI: http://dx.doi.org/10.1002/spe.524

[14] F. Glover. 1989. Tabu search (part I). *ORSA Journal on Computing* 1, 3 (1989), 190–206.

[15] Torbjörn Granlund and the GMP development team. 2012. *GNU MP: The GNU Multiple Precision Arithmetic Library* (5.0.5 ed.). http://gmplib.org/.

[16] Nikolaus Hansen, Anne Auger, Dimo Brockhoff, Dejan Tusar, and Tea Tusar. 2016. COCO: Performance Assessment. *CoRR* abs/1605.03560 (2016). http://arxiv.org/abs/1605.03560

[17] Georges R. Harik, Fernando G. Lobo, and David E. Goldberg. 1999. The Compact Genetic Algorithm. *IEEE Trans. on Evolutionary Computation* 3, 4 (November 1999), 287–297.

[18] J. H. Holland. 1975. *Adaptation in natural and artificial systems.* University of Michigan Press, Ann Arbor.

[19] Thomas Jansen. 2013. *Analyzing Evolutionary Algorithms.* Springer.

[20] S. A. Kauffman. 1993. *The origins of order: self-organisation and selection in evolution.* Oxford University Press.

[21] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. 1983. Optimization by simulated annealing. *Science* 220, 4598 (May 1983), 671–680.

[22] P. Larrañaga and J.A. Lozano. 2002. *Estimation of Distribution Algorithms. A New Tool for Evolutionary Computation.* Kluwer Academic Publishers.

[23] Max-SAT 2016. *The Eleventh Evaluation of Max-SAT Solvers.* http://maxsat.ia.udl.cat Last accessed April 26, 2017.

[24] S Mertens. 1996. Exhaustive search for low-autocorrelation binary sequences. *Journal of Physics A: Mathematical and General* 29, 18 (1996), L473. http://stacks.iop.org/0305-4470/29/i=18/a=005

[25] H. Mühlenbein. 1997. The equation for response to selection and its use for prediction. *Evolutionary Computation* 5, 3 (1997), 303–346.

[26] H. Mühlenbein and T. Mahnig. 2001. Evolutionary Algorithms: From Recombination to Search Distributions. In *Theoretical Aspects of Evolutionary Computing*, Leila Kallel, Bart Naudts, and Alex Rogers (Eds.). Springer Berlin Heidelberg, 135–174.

[27] OpenMP Architecture Review Board. 2015. OpenMP Application Program Interface Version 4.5. (November 2015). http://www.openmp.org/wp-content/uploads/openmp-4.5.pdf

[28] Christos M. Papadimitriou. 1994. *Computational complexity.* Addison-Wesley, Reading, Massachusetts.

[29] José Antonio Parejo, Antonio Ruiz-Cortés, Sebastián Lozano, and Pablo Fernandez. 2012. Metaheuristic optimization frameworks: a survey and benchmarking. *Soft Computing* 16, 3 (2012), 527–561. DOI: http://dx.doi.org/10.1007/s00500-011-0754-8

[30] M. Sebag and M. Schoenauer. 1997. A society of hill-climbers. In *Proc. IEEE Int. Conf. on Evolutionary Computation*. Indianapolis, 319–324.

[31] Thomas Stützle and Holger H. Hoos. 2000. MAX–MIN Ant System. *Future Generation Computer Systems* 16, 8 (2000), 889–914.