Computationally Efficient Local Optima Network Construction

Jonathan E. Fieldsend
J.E.Fieldsend@exeter.ac.uk
University of Exeter, EX4 4QF, UK

ABSTRACT

There has been an increasing amount of research on the visualisation of search landscapes through the use of exact and approximate local optima networks (LONs). Although there are many papers available describing the construction of a LON, there is a dearth of code released to support the general practitioner constructing a LON for their problem. Furthermore, a naive implementation of the algorithms described in work on LONs will lead to inefficient and costly code, due to the possibility of repeatedly reevaluating neighbourhood members, and partially overlapping greedy paths. Here we discuss algorithms for the efficient computation of both exact and approximate LONs, and provide open source code online. We also provide some empirical illustrations of the reduction in the number of recursive greedy calls, and quality function calls that can be obtained on NK model landscapes, and discretised versions of the IEEE CEC 2013 niching competition tests functions, using the developed framework compared to naive implementations. In many instances multiple order of magnitude improvements are observed.

CCS CONCEPTS

• **Mathematics of computing → Combinatorial optimization**; **Graph theory**;

ACM Reference Format:
Jonathan E. Fieldsend. 2018. Computationally Efficient Local Optima Network Construction. In *GECCO '18 Companion: Genetic and Evolutionary Computation Conference Companion, July 15–19, 2018, Kyoto, Japan.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3205651.3208263

1 INTRODUCTION

Compact visualisations of fitness landscapes associated with an optimisation problem are useful to both optimisation practitioners and to problem owners. In recent years the local optima network (LON) has been developed as an intuitive and helpful graph-based visualisation of the fitness landscape, with vertices indicating local modes, vertex sizes the corresponding basin size of a mode, vertex colour indicating mode quality and the occurrence and weight of a directed edge between vertices conveying some measure of how *easy* it is to move from one optima or basin to the other.

We assume without loss of generality that the optimisation problem is to find an $\mathbf{x} \in \mathcal{X}$ which maximises a quality measure $f(\mathbf{x})$. Here we illustrate the legal search domain $\mathcal{X}$ as containing all bit strings with d elements, though other alphabets may be employed (in the later empirical section we also illustrate results using arrays of integers).

A LON is a graph $G = (V, E)$ which describes the landscape induced by the triple $(\mathcal{X}, f, N)$, where $N(\mathbf{x})$ returns the *neighbourhood set* of $\mathbf{x}$ [6]. The vertices V of the graph represent local optima — those solutions which are no worse than any of its neighbours[1] — and their respective basins of attraction. These basins are essentially the volume of $\mathcal{X}$ where repeated moves from locations in a basin, e.g. greedy selection under the neighbourhood function N, lead to the corresponding optima. The edges E represent the probability of moving from one optima/basin to another.

Two proposed edge definitions are:

- **Basin transitions** [6] — which weight the edge from one vertex V_l to another V_k (including when $l = k$) as the proportion of the basin members of V_l whose neighbours' basin is V_k. This may be computed through e.g. greedy neighbourhood moves under N from each location. An illustration is provided in Figure 1 for a random NK model [4].
- **Escape edges** [8] — which weight the edge from vertex V_l to another V_k as the proportion of the solutions from $\mathcal{X}$ who lie in the kth basin *and* whose distance to the lth mode location is $\leq m$. Typically m is chosen so that is is equal or greater than the distance of neighbours under N. For instance, $N(\mathbf{x})$ used in greedy moves may return all single bit flip (hamming distance 1) neighbours of $\mathbf{x}$, whereas the escape edge calculation may be for $m = 2$, and therefore include both single- and two-bit flip neighbours. We use the notation $N_m(\mathbf{x})$ to denote all members of $\mathcal{X}$ with a distance of m or less from $\mathbf{x}$ to disambiguate these two potentially different neighbourhood functions. An illustration is provided in Figure 2, using the same underlying data as used in Figure 1 for a range of m.

Although there is a growing body of published work on LONs, there is a dearth of accessible code implementations to enable the wider community to use the technique, creating a barrier to their use. Some publications have published data on constructed LONs, but not the software directly employed (e.g. [7][2]), and others have published code for particular problems (e.g. [2][3]), but the author has yet to find any general purpose code for LON construction online. In response to this we present a general implementation in Java which constructs exact and estimated LON graphs (V and E) for any problem and solution types conforming to the provided Java interfaces.

In constructing this package it became apparent that a direct naive implementation of the high-level algorithms presented in

ACM ISBN 978-1-4503-5764-7/18/07…$15.00
https://doi.org/10.1145/3205651.3208263

[1]The reader interested in neutrality in LONs is advised to read [9].
[2]https://datastorre.stir.ac.uk/handle/11667/89
[3]http://www-lisic.univ-littoral.fr/~verel/RESEARCH/codeLON-0.1.zip

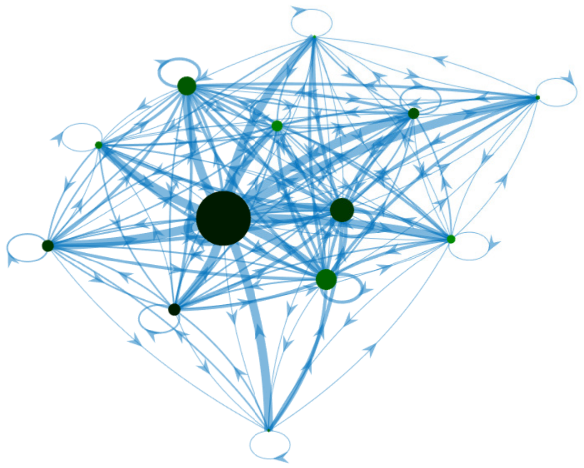

Figure 1: Exact LON of a random nk model instance with $n = 10$ and $k = 3$ using basin transition edges.

much of the LON literature would lead to inefficient code, and we now detail some of the computational efficiency considerations in the Java implementation, which should also be of interest to those developing/refactoring implementations in other languages. We then describe the package itself, followed by some empirical results and discussion.

To avoid confusion with notation for neighbourhoods, we will eschew the standard convention of referring to '*NK models*', and instead refer to '*nk models*' in later sections.

2 EFFICIENCY CONSIDERATIONS

First, let us consider the computational task of defining the vertices in a LON. For illustration we use the score obtained by flipping the ith bit if using a hamming distance 1 neighbourhood [1], i.e.:

$$S_i(\mathbf{x}) = f(\mathbf{x} \oplus 1_i) - f(\mathbf{x})$$

where $\oplus$ denotes the *exclusive or* operator. This score can be used to identify whether a solution is a local optima, and added to the set of vertices V as denoted in Algorithm 1 for a complete enumeration of $\mathcal{X}$.

If there is enough space to store the entire domain in memory, then a list, L_f, may be used to store the fitness under the quality function f for any putative solution. This may be indexed by the integer represented by the solution's bit string:

$$K(\mathbf{x}) = \sum_{i=0}^{|\mathbf{x}|-1} x_i 2^i.$$

In the case of exact LON generation by complete enumeration of the search domain, accessing the neighbours of a solution is as simple as accessing the relevant indices in L_f via the index function K.

If the LON is to be estimated rather than generated by complete enumeration, a hash map, $H_{K,f}$, can be used to store all $\mathbf{x}$ evaluated under f, and their corresponding quality (with $K(\mathbf{x})$ used as the hash code). $H_{K,f}(\mathbf{x})$ may be queried prior to computing $f(\mathbf{x})$ for any

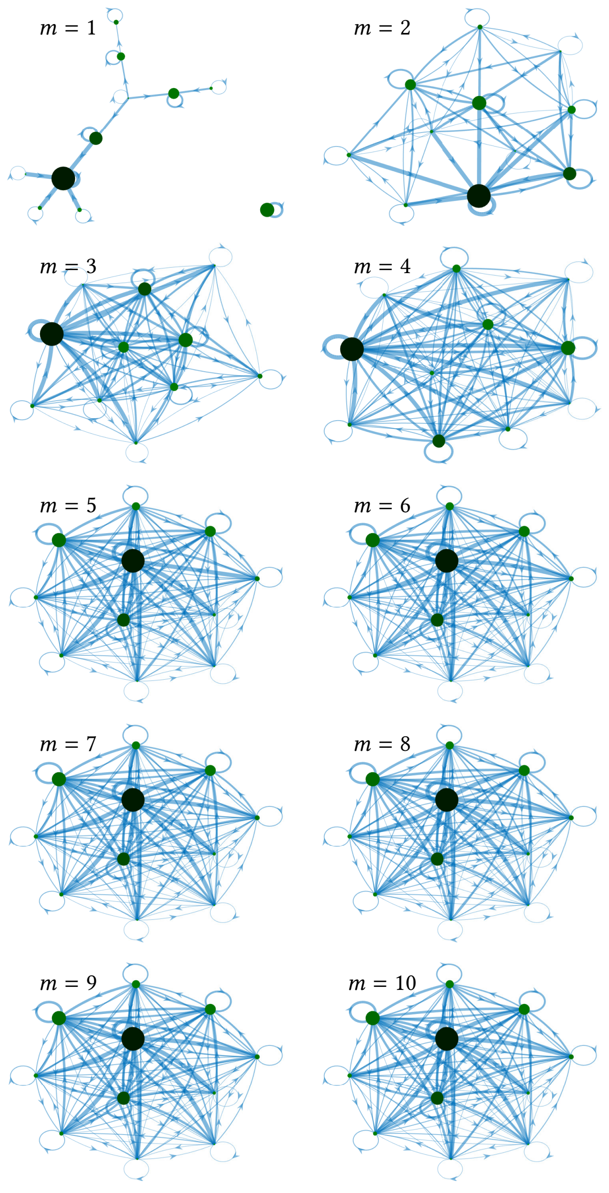

Figure 2: Exact LON of an nk model instance (the same instance as Figure 1) with $n = 10$ and $k = 3$ using escape edges, $m = \{1, \ldots, 10\}$.

solution to save duplicated evaluation under f (as e.g. a previously evaluated point may be a neighbour of a current solution).

As well as defining V, the construction of a LON requires the computation of E (the adjacency matrix, or adjacency list, and weights). A high level procedure is provide in Algorithm 2 for basin transition edges and Algorithm 3 for escape edges. Algorithm 4 details the recursive hill-climbing routine used by both.

In these algorithms there is the potential to both repeat the same hill-climb (as they iterate over all members of $\mathcal{X}$, and all neighbours of these members — which are also from $\mathcal{X}$), and also overlapping

```
Result: Local optima, $V$
$V := \emptyset$;
for $\mathbf{x} \in \mathcal{X}$ do
    if $S_i(\mathbf{x}) \leq 0 \quad \forall 1 \leq i \leq d$ then
        $V := V \cup \{\mathbf{x}\}$
    end
end
```

Algorithm 1: Identifying modes (LON vertices) via exhaustive search

```
Data: vertices (local optima), $V$
Result: adjacency matrix, $E$, and basin sizes, $B$
$E := 0_{|V|,|V|}$;
$B := 0_{|V|}$;
for $\mathbf{x} \in \mathcal{X}$ do
    $\mathbf{v}$ = hillclimb($\mathbf{x}, \mathcal{X}, f, N$);
    $B_{\mathbf{v}} := B_{\mathbf{v}} + 1$;
    for $\mathbf{x}' \in N(\mathbf{x})$ do
        $\mathbf{v}'$ = hillclimb($\mathbf{x}', \mathcal{X}, f, N$);
        $E_{\mathbf{v},\mathbf{v}'} := E_{\mathbf{v},\mathbf{v}'} + 1$;
    end
end
```

Algorithm 2: Computing basin sizes and basin transitions

```
Data: vertices (local optima), $V$, distance $m$
Result: adjacency matrix, $E$, and basin sizes, $B$
$E := 0_{|V|,|V|}$;
$B := 0_{|V|}$;
for $\mathbf{x} \in \mathcal{X}$ do
    $\mathbf{v}$ = hillclimb($\mathbf{x}, \mathcal{X}, f, N$);
    $B_{\mathbf{v}} := B_{\mathbf{v}} + 1$;
end
for $\mathbf{v} \in V$ do
    $\mathbf{v}'$ = hillclimb($\mathbf{v}, \mathcal{X}, f, N_m$);
    $E_{\mathbf{v},\mathbf{v}'} := E_{\mathbf{v},\mathbf{v}'} + 1$;
end
```

Algorithm 3: Computing basin sizes and edge escapes

```
Data: solution $\mathbf{x}$, domain $\mathcal{X}$, quality measure $f$,
      neighbourhood function $N$
Result: local optima $\mathbf{v}$
$\mathbf{v} := \mathbf{x}$;
for $\mathbf{u} \in N(\mathbf{x})$ do
    if $f(\mathbf{u}) > f(\mathbf{v})$ then
        $\mathbf{v} := \mathbf{u}$;
    end
end
if $\mathbf{v} \neq \mathbf{x}$ then
    $\mathbf{v}$ := hillclimb($\mathbf{v}, \mathcal{X}, f, N$);
end
```

Algorithm 4: The `hillclimb` routine

segments of hill-climbs (where paths meet at a coincident solution). Here, as well as using $H_{K,f}$ to save repeated queries under f for

```
Data: Number of restart locations, $n$
Result: Approximated modes, $M$, and basin sizes, $B$
        (indexed by mode location)
$counter := 0$;
$M := \emptyset$;
$B := \emptyset$;
while $counter < n$ do
    $counter := counter + 1$;
    $\mathbf{v}'$ := hillclimb(random_draw($\mathcal{X}$), $\mathcal{X}, f, N$);
    if $\{\mathbf{v}'\} \notin M$ then
        $M := M \cup \{\mathbf{v}'\}$;
        $B_{\mathbf{v}'} := 1$;
    else
        $B_{\mathbf{v}'} := B_{\mathbf{v}'} + 1$;
    end
end
$B := B/n$;
```

Algorithm 5: Approximating mode number and basin size.

the same solution, another hashmap, $H_{K,\mathbf{v}}$, may be stored linking any $\mathbf{x}$ queried on a path during a hill-climb traversal to the optima reached at the end of the path. This necessitates a dynamically updated list of solutions whose neighbourhoods are enumerated during a hill-climb, which may all be subsequently mapped to the $\mathbf{v}$ returned. Such dynamic lists typically have amortised $O(1)$ update costs in standard language implementations. $H_{K,\mathbf{v}}$ may be queried in later hill-climbing calls, as if a location has had its optima previously identified, the optima may be immediately returned without repetition of the climb segment (the result of overlapping hill-climbing paths) — assuming a non-neutral landscape.

Algorithm 5 shows a high-level procedure for the unbiased approximation of a LON. As with the exact LON algorithm the use of $H_{K,\mathbf{v}}$ can reduce both calls on f and on `hillclimb`, however unlike in the exact case, at algorithm completion it is highly likely that $|H_{K,f}| \neq |H_{K,\mathbf{v}}|$ (as not all evaluated neighbours, which will enter $H_{K,f}$, will be a path root in $H_{K,\mathbf{v}}$).

Although not explored here, the same exploitation of hash maps may also be employed for biased LON approximations, such as the popular choice of using iterative local hill-climbers.

3 THE JAVA PACKAGE

The open source Java `lons` package may be found in the author's GitHub page[4]. A snapshot of the package corresponding with the version at publication of this paper, in the form of a *jar* file, may be found at Open Research Exeter[5] and the University of Exeter hosted project page[6]. The `lons` package contains a number of abstract classes, interfaces, and concrete classes to allow LON generation for arbitrary problem, neighbourhood and solution definitions. Some concrete implementations illustrating the use of the core LON generator are provided in the `lons.examples` package.

[4]https://github.com/fieldsend
[5]https://ore.exeter.ac.uk/repository
[6]http://pop-project.ex.ac.uk

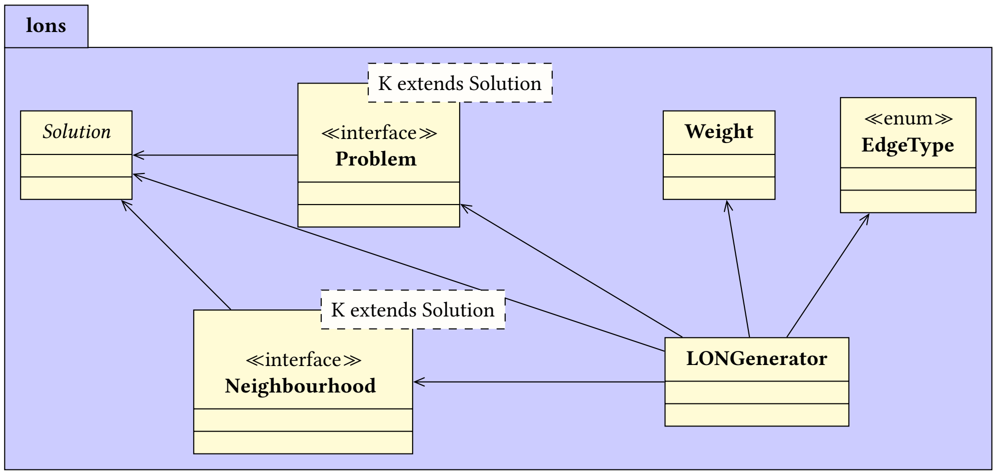

Figure 3: UML class diagram of the `lons` package.

The generator exploits the efficiency gains detailed in earlier sections. The current release version of `lons` contains the following core members:

- `Solution`: an abstract class which requires concrete subtypes to provide methods to (i) return a non-negative integer to representation of the solution — this is used as the key for the hash code [7] and (ii) the number of elements in a solution (e.g. bit-string length). It also provides concrete methods for the hash code generation via the first abstract method .
- `Problem<K extends Solution>`: a parameterised (i.e. generic) interface which requires implementors to provide methods to (i) return the quality of a provided `Solution` subtype `K`, (ii) return a random instance of the `Solution` subtype `K`, and (iii) return an array containing all possible `Solution` subtype `K` instances (used in exact LON generation).
- `Neighbourhood<K extends Solution>`: a parameterised interface which requires implementors to provide methods which (i) return an array of all immediate neighbour instances of `Solution` subtype `K` corresponding to the `Solution` subtype `K` argument (i.e. function N), and (ii) return an array of all neighbour instances of `Solution` subtype `K` corresponding to the `Solution` subtype `K` argument up to a given distance (i.e. function N_m).
- `EdgeType`: an enumerated type which holds a value representing either basin transition or escape edge.
- `Weight`: a concrete class, instances of which are used to store and update basin sizes and edge weights.
- `LONGenerator`: a concrete class which provides parametrised (`K extends Solution`) methods to calculate LONs (either exhaustively or via sampling) for a given problem, solution, neighbourhood, and edge type. It also provides a method to write out a LON's details (optima indices, optima quality, basin sizes, adjacency lists and corresponding weights) to plain text files for manipulation by the user's preferred graphing software.

Additionally, example classes which represent (discretised) IEEE CEC 2013 niching competition multi-modal test problems (which wrap Java implementations of these problems are published online by the competition authors [5][8]) are also provided as part of the `lons.examples` package, along with classes representing *nk* landscape problems and associated solution implementations. These are used in the later empirical section. Specifically this package includes:

- `BinarySolution`: an abstract subtype of `Solution`.
- `BinaryProblem`: an interface subtype of `Problem<BinarySolution>`.
- `ConcreteBinarySolution`: an implementor of `BinarySolution`.
- `BinaryHammingNeighbourhood`: an implementor of `Neighbourhood<BinarySolution>`.
- `NKModelProblem`: an implementor of `Problem<BinarySolution>`.
- `IntegerVectorSolution`: an abstract subtype of `Solution`.
- `IntegerVectorProblem`: an interface subtype of `Problem <IntegerVectorSolution>`.
- `ConcreteIntegerVectorSolution`: an implementor of `IntegerVectorSolution`.
- `IntegerVectorNeighbourhood`: an implementor of `Neighbourhood<IntegerVectorSolution>`.
- `DiscretisedCEC2013NichingProblem`: an implementor of `Problem<IntegerVectorSolution>`.
- `Experiments`: a concrete class which recreate the empirical work of this paper.
- Various `Neighbourhood` and `Problem` subtypes which use the *decorator pattern* [3] to track calls to methods of implementations required in the experiments here, but which contain overhead costs not needed in general usage.

A Unified Modelling Language (UML) class diagram of the package is provided in Figure 3, illustrating the interdependencies between the various `lon` package members.

[7]The maximum of the Java `int` type is 2 147 483 647. If $|\mathcal{X}|$ is larger than this then care should be taken in the definition of K's hash code, to minimise clash situations.

[8]https://github.com/mikeagn/CEC2013

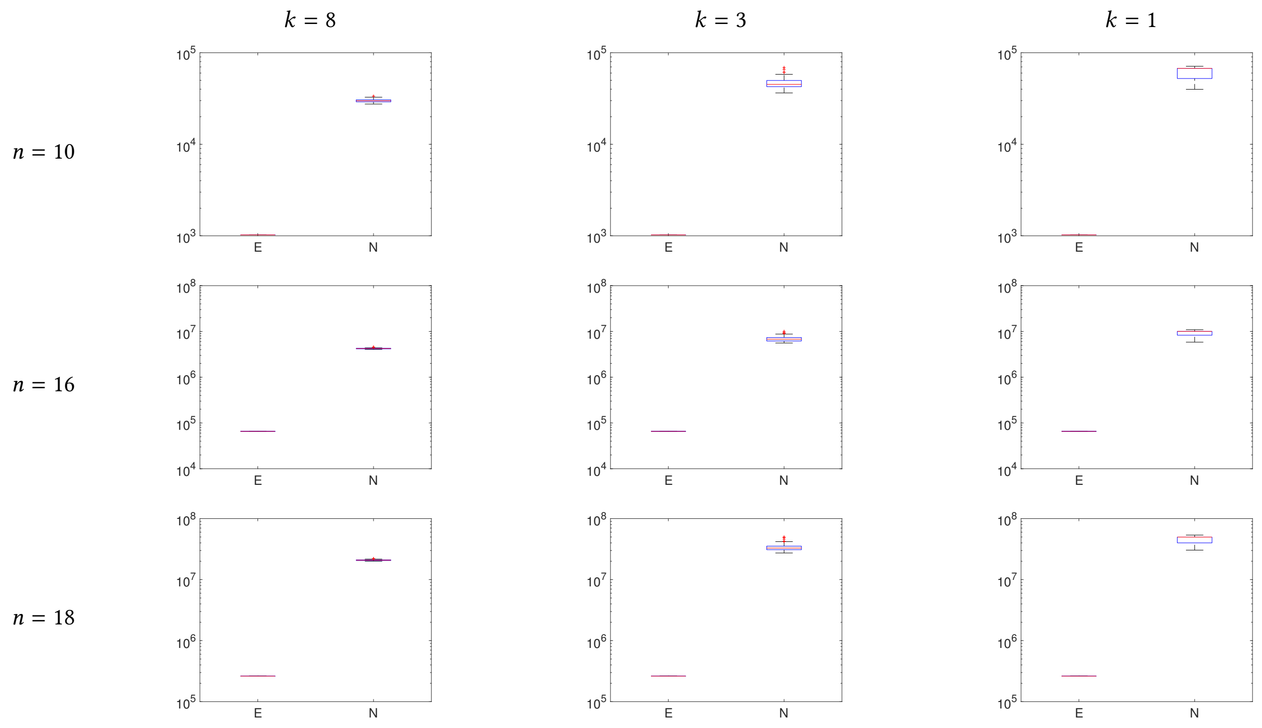

Figure 4: Number of greedy hill-climber routine calls for exact LON generation of 100 instances of 9 different *nk* models using basin transition edges. 'E' denotes efficient implementation, 'N' denotes naive implementation. Note, as K increases the number of modes tends to increase and the length of paths tends to decrease — as smoother landscapes tend to have larger basins and therefore longer paths to local optima.

4 EFFICIENCY GAINS: EMPIRICAL ASSESSMENT

We now detail some observed efficiency gains exploiting the package for various exhaustive and sampled LONs for a range of *nk* model instances, and for discretised versions of IEEE CEC 2013 niching competition multi-modal test problems.

4.1 Exact LONs

In the first set of experiments we evaluate the benefit of using $H_{K,\mathbf{v}}$ for exact LON calculation (i.e. exhaustive enumeration) using basin transitions and escape edges. We run efficient and naive implementations for 100 landscape instances induced from nine different *nk* models: $\{(10,8),(10,3),(10,1),(16,8),(16,3),(16,1),(18,8),(18,3),(18,1)\}$. These correspond to $|\mathcal{X}|$ ranging from 1 024 up to 262 144. Figure 4 shows box plots of the numbers of calls to the greedy hill-climber routine `hillclimb` (Algorithm 4) — including recursive calls — in the two implementations for LON generation with basin transition edges. Figure 5 provides the corresponding box plots for escape edges ($m = 2$).

Note there is no variation across runs for the efficient implementation, as the number of (recursive) calls of the greedy hillclimber is always exactly $|\mathcal{X}|$ — as when climbing from a solution $H_{K,\mathbf{v}}$ is first queried and if it has been visited in a previous path the corresponding end optima is returned. In both figures the number of greedy hill-climb calls is markedly lower in the efficient implementation, often by multiple orders of magnitude.

We next run the generator on discretised versions of the CEC 2013 Niching Competition test functions, specifically functions 1–12 for 10–1 000 equally spaced values per variable (i.e. various *grid resolutions*) for problems 1–7, and 10–200 equally spaced values per variable for problems 8–12. $N(\mathbf{x})$ in this case are all neighbours of the integer vector representing a grid point, $\mathbf{x}$, which differ by ± 1 grid step on one variable only (i.e. the von Neumann neighbourhood). Figure 6 shows how the number of greedy hill-climber calls varies with resolution for basin transition edges using the two LON generator implementations. Here the reduction in calls to `hillclimb`, depending on the problem, is between one and three orders of magnitude for the maximum resolution investigated.

4.2 Estimated LONs

In the next set of experiments we evaluate the benefit of using $H_{K,\mathbf{v}}$ and $H_{K,f}$ for approximate LON generation. Here we take 10 000 random starts from the $\mathcal{X}$ associated with 100 different instances of the *nk* models $\{(24,8),(24,3),(24,1)\}$. Figure 7 shows box plots of the number of calls to the greedy hill-climber routine (including recursive calls) and to f in these two implementations for LON generation with basin transition edges. Figure 8 provides the corresponding box plots for escape edges ($m = 2$). Note that, unlike in the exact case, there is variation in the greedy hill-climb

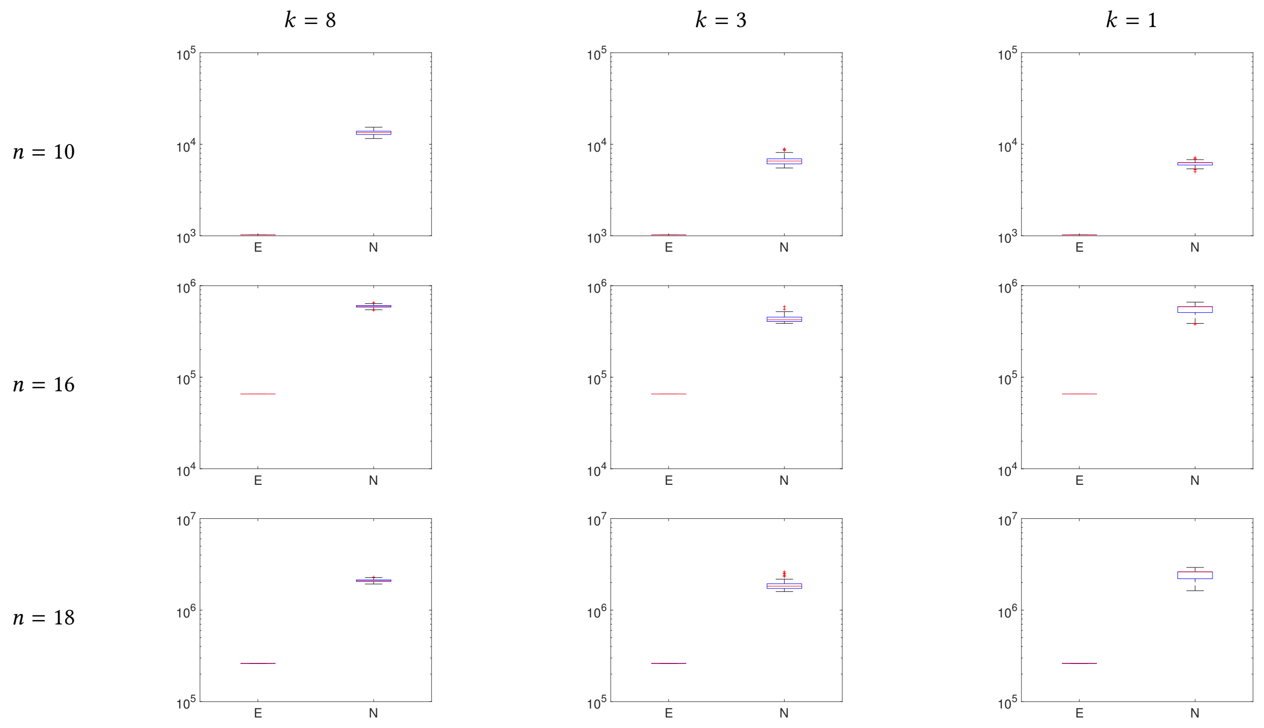

Figure 5: Number of greedy hill-climber routine calls for exact LON generation of 100 instances of 9 different *nk* models using escape edges. 'E' denotes efficient implementation, 'N' denotes naive implementation.

calls for the efficient implementation. This is because the number of hill-climb calls will depend on the paths lengths encountered during the sampling of $\mathcal{X}$. This variation will tend to zero as the relative proportion of $\mathcal{X}$ visited approaches 1, and the maximum value is bounded by the efficient complete enumeration implementation, i.e. $|\mathcal{X}|$.

As with the exact LONs, the efficiency gains are readily apparent – with large reductions in calls to both `hillclimb` and f experienced.

5 DISCUSSION

We have presented some computational efficiency considerations for the generation of the LON graph. The empirical examples illustrate the marked reduction in greedy hill-climber routine calls, and queries under the quality function f, that may be enjoyed when exploiting previous visit information. This does of course come at the cost of *memory*. Certainly, when tackling large problems with high sample numbers, it is possible to exceed the memory capacity of a machine — as, due to path lengths, the number of evaluations under f may be multiple orders of magnitude larger than the number of samples. For example, on the $n = 24$ and $k = 8$ model, 10 000 random starting solutions for generating a LON with basin transitions typically resulted in around 6 000 000 unique solutions being queried (over a third of the corresponding $|\mathcal{X}|$), with 600 000 of these residing on paths (meaning $|H_{K,\mathbf{f}}| \approx 10^7$, and $|H_{K,\mathbf{v}}| \approx 10^6$). Where the space is large and machine memory is low, a 'middle-road' option would be to update $H_{K,\mathbf{f}}$ *only* with locations that fall on paths. This would still fully benefit from the reduction in calls to `hillclimb`, and some of the reduction in repeated calls to f.

As the package implementation relies on the user providing concrete implementations of the `Problem` and `Neighbourhood` interfaces, and a subtype of the abstract `Solution` class, grey-box approaches (e.g. [1, 10]) may be utilised to further speed up performance.

In this work a landscape without neutrality has been assumed. With neutral landscapes the approach proposed to improve efficiency using $H_{K,\mathbf{v}}$ will result in erroneous LONs, as the optima for a putative solution is non-deterministic (as, depending on the stochastic choice of equally best quality neighbours, a different climb path will be taken). In the case of the generation of an exact LON with neutrality there is scope for book-keeping approaches to compute the re-weightings needed (although this is not a feature of the software package currently). For the approximate case however an efficient solution is much more problematic — simply exploring all neutral branches at the greedy hillclimbing stage could at the worst case result in an attempt to enumerate the prohibitively large $\mathcal{X}$. As such the efficiency gains of utilising $H_{K,\mathbf{v}}$ cannot be realised in landscapes with neutrality, without the imposition of a bias in the resultant LON. However, even for neutral landscapes, $H_{K,f}$ can still be usefully exploited.

Finally, there is the additional question with approximated LONs regarding to how many samples are required for the resultant LON

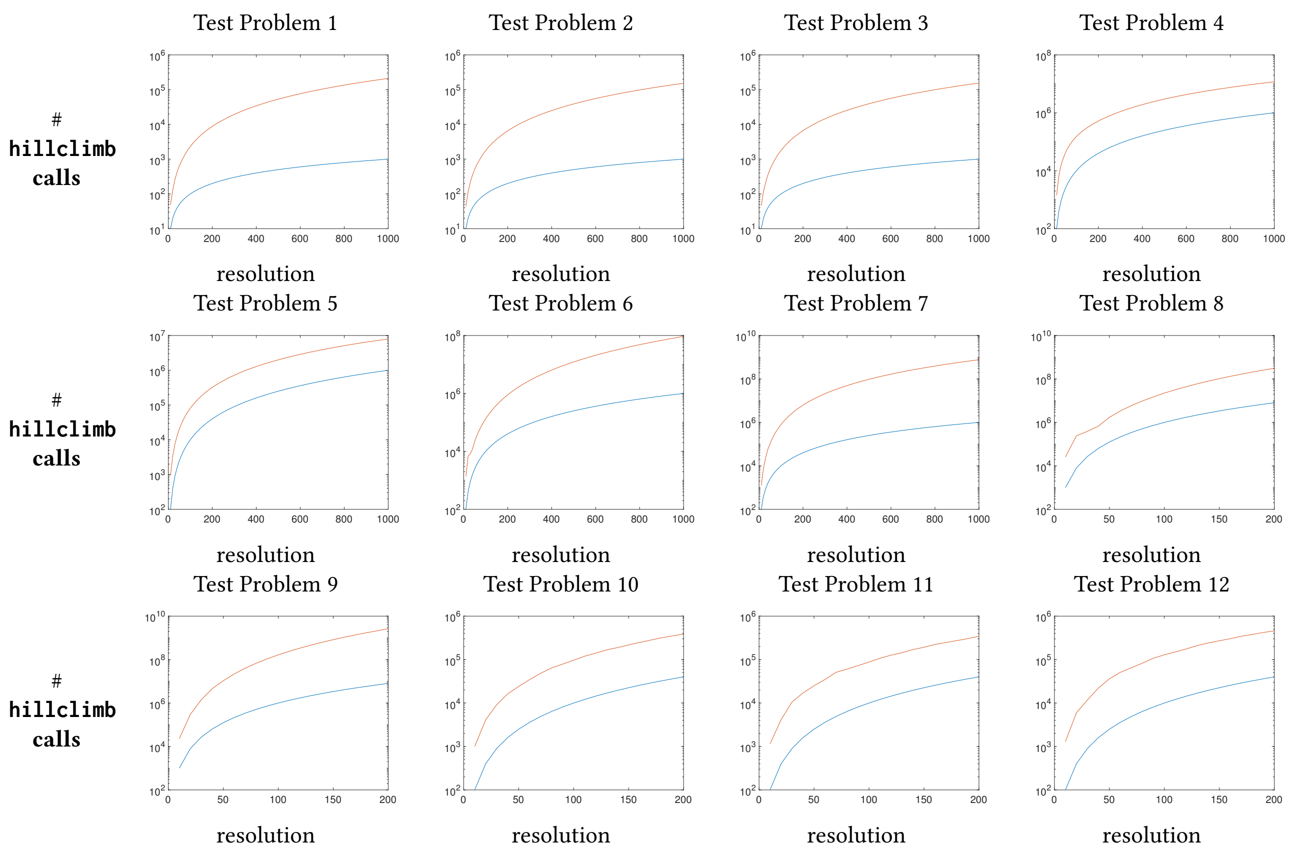

Figure 6: Number of greedy hill-climber routine calls for exact LON generation of discretised CEC 2013 Niching competition problems using basin transition edges. Blue line denotes efficient implementation, red line denotes naive implementation.

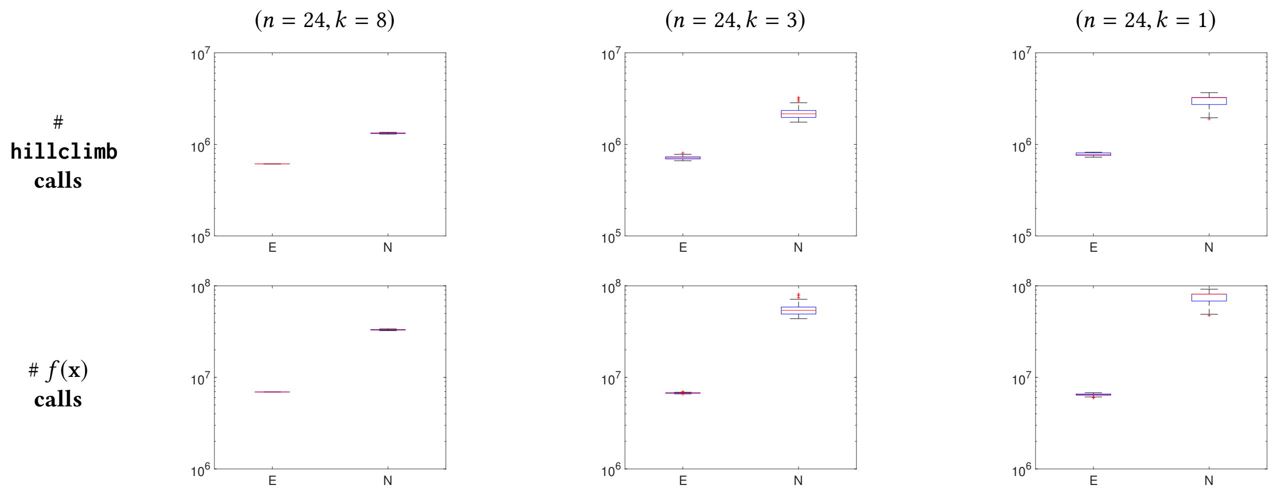

Figure 7: Number of greedy hill-climber routine calls and quality function queries for estimated LONS of 100 instances of 3 different *nk* models using basin transition edges. 'E' denotes efficient implementation, 'N' denotes naive implementation. 10 000 random locations initially sampled.

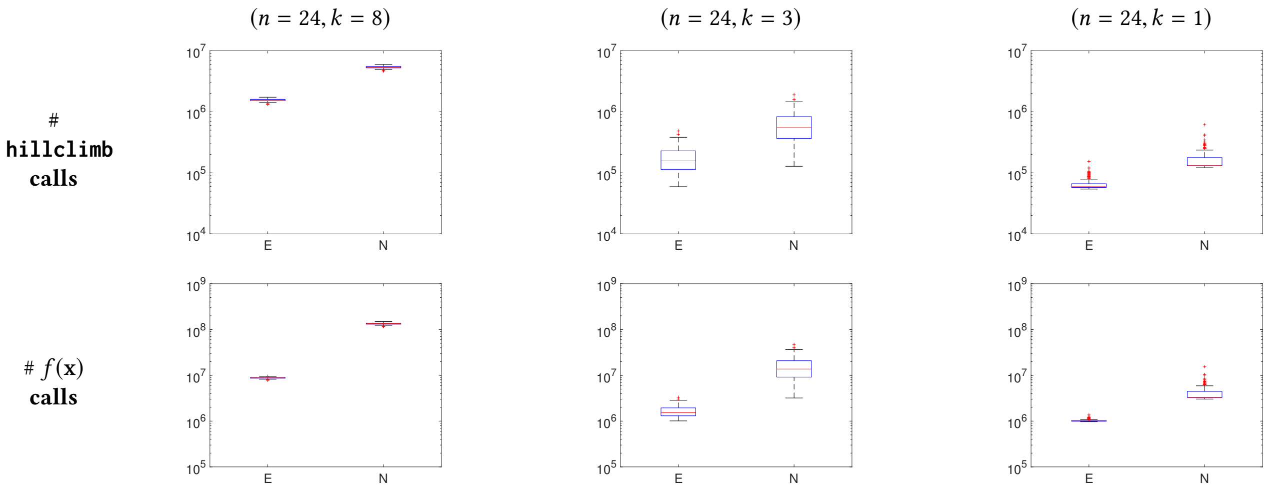

Figure 8: Number of greedy hill-climber routine calls and quality function queries for estimated LONS of 100 instances of 3 different *nk* models using escape edges. 'E' denotes efficient implementation, 'N' denotes naive implementation. 10 000 random locations initially sampled.

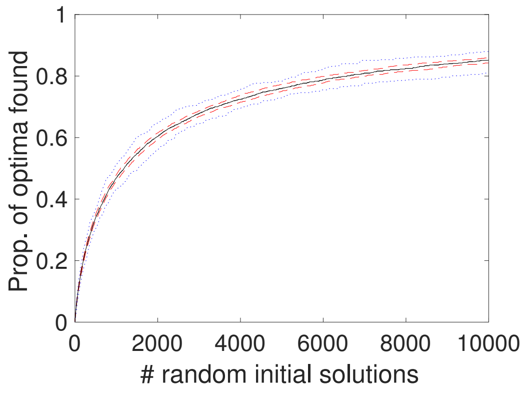

Figure 9: Proportion of local optima detected as samples increase. Results over 100 runs of approximate LON generation. The same $n = 24$ and $k = 3$ model instance for each run, but different random seeds used for random solution generation. Solid line denotes median, dashed line the inter quartile range and dotted line the maximum and minimum.

to be a *reasonable* approximation of the exact LON. This will undoubtably depend on the size of $|\mathcal{X}|$ and the ruggedness of the landscape. However as the number of solutions queried for a random sample in an unbiased search also depends upon the ruggedness and domain size, investigating this further will be an interesting strand of future work. Figure 9 shows how the number of modes in an estimated LON varies with sample number for the same $n = 24$ and $k = 3$ model instance, over 100 different runs with different random seeds for the sampling. For this particular instance, over 50% of the optima are identified with 2 000 random initial solutions, and over 80% after 10 000. This is only a coarse comparison, and using distance metrics between graphs is likely to be more informative.

ACKNOWLEDGMENTS

This work was supported by the Engineering and Physical Sciences Research Council [grant number EP/N017846/1]. The author would like to thank Sébastien Vérel and Gabriela Ochoa for providing inspirational invited talks on LONs at the University of Exeter during this grant, and also Ozgur Akman, Khulood Alyahya and Kevin Doherty.

REFERENCES

[1] F. Chicano, D. Whitley, and A.M. Sutton. 2014. Efficient Identification of Improving Moves in a Ball for Pseudo-boolean Problems. In *Proceedings of the 2014 Annual Conference on Genetic and Evolutionary Computation (GECCO '14)*. ACM, 437–444.
[2] F. Daolio, S. Verel, G. Ochoa, and M. Tomassini. 2010. Local optima networks of the quadratic assignment problem. In *Evolutionary Computation (CEC), 2010 IEEE Congress on*. IEEE, 1–8.
[3] E. Freeman, E. Freeman, K. Sierra, and B. Bates. 2004. *Head First Design Patterns.* O'Reilly.
[4] S. Kauffman and E. Weinberger. 1989. The NK Model of rugged fitness landscapes and its application to the maturation of the immune response. *Journal of Theoretical Biology* 141, 2 (1989), 211—245.
[5] X. Li, A. Engelbrecht, and M. G. Epitropakis. 2013. Benchmark functions for CEC'2013 special session and competition on niching methods for multimodal function optimization. *RMIT University, Evolutionary Computation and Machine Learning Group, Australia, Tech. Rep* (2013).
[6] G. Ochoa, M. Tomassini, S. Vérel, and C. Darabos. 2008. A study of NK landscapes' basins and local optima networks. In *Proceedings of the 10th annual conference on Genetic and evolutionary computation.* ACM, 555–562.
[7] N. Veerapen, F. Daolio, and G. Ochoa. 2017. Modelling Genetic Improvement Landscapes with Local Optima Networks. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion (GECCO '17).* ACM, 1543–1548.
[8] S. Vérel, F. Daolio, G. Ochoa, and M. Tomassini. 2012. Local Optima Networks with Escape Edges. In *Artificial Evolution*, J.-K. Hao, P. Legrand, P. Collet, N. Monmarché, E. Lutton, and M. Schoenauer (Eds.). Springer Berlin Heidelberg, 49–60.
[9] S. Verel, G. Ochoa, and M. Tomassini. 2011. Local optima networks of NK landscapes with neutrality. *IEEE Transactions on Evolutionary Computation* 15, 6 (2011), 783–797.
[10] L. D. Whitley, F. Chicano, and B. W. Goldman. 2016. Gray Box Optimization for Mk Landscapes (NK Landscapes and MAX-kSAT). *Evolutionary Computation* 24, 3 (2016), 491–519.