The Journal of Open Source Software

nimCSO: A Nim package for Compositional Space Optimization

Adam M. Krajewski [1¶], **Arindam Debnath** [1], **Wesley F. Reinhart** [1,2], **Allison M. Beese** [1], and **Zi-Kui Liu** [1]

1 Department of Materials Science and Engineering, The Pennsylvania State University, USA **2** Institute for Computational and Data Sciences, The Pennsylvania State University, USA ¶ Corresponding author

DOI: 10.xxxxxx/draft

Software

- Review
- Repository
- Archive

Editor: Open Journals

Reviewers:

- @openjournals

Submitted: 01 January 1970
Published: unpublished

Summary

`nimCSO` is a high-performance tool implementing several methods for selecting components (data dimensions) in compositional datasets, which optimize the data availability and density for applications such as machine learning. Making said choice is a combinatorically hard problem for complex compositions existing in highly dimensional spaces due to the interdependency of components being present. Such spaces are encountered, for instance, in materials science, where datasets on Compositionally Complex Materials (CCMs) often span 20-45 chemical elements, 5-10 processing types, and several temperature regimes, for up to 60 total data dimensions.

At its core, `nimCSO` leverages the metaprogramming ability of the Nim language (Rumpf, 2023) to optimize itself at the compile time, both in terms of speed and memory handling, to the specific problem statement and dataset at hand based on a human-readable configuration file. As demonstrated in Methods and Performance section, `nimCSO` reaches the physical limits of the hardware (L1 cache latency) and can outperform an efficient native Python implementation over 400 times in terms of speed and 50 times in terms of memory usage (*not* counting interpreter), while also outperforming NumPy implementation 35 and 17 times, respectively, when checking a candidate solution.

`nimCSO` is designed to be both (1) a user-ready tool, implementing two efficient brute force approaches (for handling up to 25 dimensions), a custom search algorithm (for up to 40 dimensions), and a genetic algorithm (for any dimensionality), and (2) a scaffold for building even more elaborate methods in the future, including heuristics going beyond data availability. All configuration is done with a simple human-readable `YAML` config file and plain text data files, making it easy to modify the search method and its parameters with no knowledge of programming and only basic command line skills.

Statement of Need

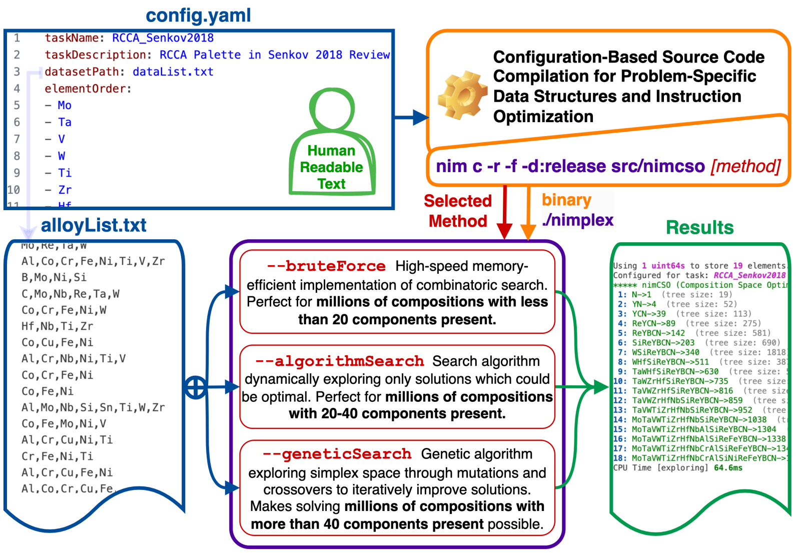
`nimCSO` is an interdisciplinary tool applicable to any field where data is composed of a large number of independent components and their interaction is of interest in a modeling effort, ranging from social sciences like economics, through medicine where drug interactions can have a large impact on the treatment, to chemistry and materials science, where the composition and processing history are critical to resulting properties. The latter has been the root motivation for the development of `nimCSO` within the ULTERA Project (ultera.org) carried under the US DOE ARPA-E ULTIMATE program which aims to develop a new generation of ultra-high temperature materials for aerospace applications, through generative machine learning models (Debnath et al., 2021) driving thermodynamic modelling and experimentation (Li et al., 2024).

One of the most promising materials for such applications are the aforementioned CCMs, and their metal-focused subset of Refractory High Entropy Alloys (RHEAs) (Senkov et al., 2018), which are rapidly growing since first proposed by (Cantor et al., 2004) and (Yeh et al., 2004). Contrary to most of the traditional alloys, they contain a large number of chemical elements (typically 4-9) in similar proportions, in hope to thermodynamically stabilize the material by increasing its configurational entropy ($\Delta S_{conf} = \Sigma_i^N x_i \ln x_i$ for ideal mixing of N elements with fractions x_i), what encourages sampling from a large palette of chemical elements. At the time of writing, ULTERA Database is the largest collection of HEA data, containing over 6,300 points manually extracted from almost 550 publications. It covers 37 chemical elements resulting in extremely compositional spaces (Krajewski et al., 2024); thus, it becomes critical to answer questions like *"Which combination of 15 elements will result in the largest dataset?"* which has $\binom{37}{15}$ or 10 billion possible solutions.

Another significant example of intended use is to perform similar optimizations over large (many millions) datasets of quantum mechanics calculations spanning 93 chemical elements and accessible through OPTIMADE API (**Evans2024?**).

Methods and Performance

Overview


config.yaml
Human Readable Text
alloyList.txt
Configuration-Based Source Code Compilation for Problem-Specific Data Structures and Instruction Optimization
nim c -r -f -d:release src/nimcso [method]
Selected Method
binary ./nimplex
Results
--bruteForce High-speed memory-efficient implementation of combinatoric search. Perfect for millions of compositions with less than 20 components present.
--algorithmSearch Search algorithm dynamically exploring only solutions which could be optimal. Perfect for millions of compositions with 20-40 components present.
--geneticSearch Genetic algorithm exploring simplex space through mutations and crossovers to iteratively improve solutions. Makes solving millions of compositions with more than 40 components present possible.

```yaml
taskName: RCCA_Senkov2018
taskDescription: RCCA Palette in Senkov 2018 Review
datasetPath: dataList.txt
elementOrder:
- Mo
- Ta
- V
- W
- Ti
- Zr
- Hf
```

```
Mo,Re,Ta,W
Al,Co,Cr,Fe,Ni,Ti,V,Zr
B,Mo,Ni,Si
C,Mo,Nb,Re,Ta,W
Co,Cr,Fe,Ni,W
Hf,Nb,Ti,Zr
Co,Cu,Fe,Ni
Al,Cr,Nb,Ni,Ti,V
Co,Cr,Fe,Ni
Co,Fe,Ni
Al,Mo,Nb,Si,Sn,Ti,W,Zr
Co,Fe,Mo,Ni,V
Al,Cr,Cu,Ni,Ti
Cr,Fe,Ni,Ti
Al,Cr,Cu,Fe,Ni
Al,Co,Cr,Cu,Fe
```

```
Using 1 uint64s to store 19 elements
Configured for task: RCCA_Senkov2018
***** nimCSO (Composition Space Opti
 1: N->1  (tree size: 19)
 2: YN->4  (tree size: 52)
 3: YCN->39  (tree size: 113)
 4: ReYCN->89  (tree size: 275)
 5: ReYBCN->142  (tree size: 581)
 6: SiReYBCN->203  (tree size: 690)
 7: WSiReYBCN->340  (tree size: 1818
 8: WHfSiReYBCN->511  (tree size: 38
 9: TaWHfSiReYBCN->630  (tree size:
10: TaWZrHfSiReYBCN->735  (tree size
11: TaVWZrHfSiReYBCN->816  (tree siz
12: TaVWZrHfNbSiReYBCN->859  (tree s
13: TaVWTiZrHfNbSiReYBCN->952  (tree
14: MoTaVWTiZrHfNbSiReYBCN->1038  (t
15: MoTaVWTiZrHfNbAlSiReYBCN->1304
16: MoTaVWTiZrHfNbAlSiReFeYBCN->1338
17: MoTaVWTiZrHfNbCrAlSiReFeYBCN->134
18: MoTaVWTiZrHfNbCrAlSiNiReFeYBCN->
CPU Time [exploring] 64.6ms
```

Figure 1: Schematic of core nimCSO data flow with a description of key methods. Metaprogramming is used to compile the software optimized to the human-readable data and configuration files at hand.

Under the hood, `nimCSO` is built around storing the data and solutions in one of two ways. The first is as bits encoded in an integer (`uint64`), which allows for highest speed and lowest memory consumption possible, but is limited to 64 dimensions and does not allow for easy extension to other use cases, thus as of publication it is used only in the special `bruteForceInt` routine. The second one, used in `bruteForce`, `algorithmSearch`, and `geneticSearch` is through a custom `ElSolution` type containing heuristic value (easily extensible) and `BitArray` payload, which is defined at compile time based on the configuration file to minimize necessary overheads. Both encodings significantly outperform both typical native Python and NumPy implementations, as shown in the Table **??**.

Table 1: Benchmarks of (1) average time to evaluate how many datapoints would be lost if 5 selected components were removed from a dataset with 2,150 data points spanning 37 components, averaged over 10,000 runs, and (2) the size of the data structure representing the dataset. Values were obtained by running scripts in `benchmarks` directory on Apple M2 Max CPU.

Tool	Object	Time per Dataset	Time per Entry *(Relative)*	Database Size *(Relative)*
`Python`$^{3.11}$	`set`	327.4 µs	152.3 ns *(x1)*	871.5 kB *(x1)*
`NumPy`$^{1.26}$	`array`	40.1 µs	18.6 ns *(x8.3)*	79.7 kB *(x10.9)*
`nimCSO`$^{0.6}$	`BitArray`	9.2 µs	4.4 ns *(x34.6)*	50.4 kB *(x17.3)*
`nimCSO`$^{0.6}$	`uint64`	0.79 µs	0.37 ns *(x413)*	16.8 kB *(x52)*

Brute-Force Search

The brute force search is a naïve method of evaluating all possibilities; however, its near-zero overhead can make it the most efficient for small problems. In this implementation, all entries in the *power set* of considered elements are represented as a range of integers from 0 to $2^{elementN} - 1$ and used to initialize `uint64`/`BitArrays`. To minimize the memory footprint of solutions, the algorithm only keeps track of the best solution for a given number of elements present in the solution. Current implementations are limited to 64 elements, as it is not feasible for more than approximately 30 elements; however, the one based on `BitArray` could be easily extended if needed.

Algorithm-Based Search

For higher dimensional problems (20-50 components), the brute force search becomes suboptimal, prompting the need for a more efficient method. The algorithm implemented in the `algorithmSearch` routine iteratively expands and evaluates candidates from a priority queue (implemented through an efficient binary heap (Williams, 1964)) while leveraging the fact that *the number of data points lost when removing elements A and B from the dataset has to be at least as large as when removing either A or B alone* to delay exploration of candidates until they can contribute to the solution. Furthermore, to (1) avoid revisiting the same candidate without keeping track of visited states and (2) further inhibit the exploration of unlikely candidates, the algorithm *assumes* that while searching for a given order of solution, elements present in already expanded solutions will not improve those not yet expanded. This effectively prunes candidate branches requiring two or more levels of backtracking. In the authors' tests, this method has generated the same results as `bruteForce`, except for occasional differences in the last explored solution.

Genetic Search

The algorithm-based method is an efficient for problems with up to 40 elements with a certain level of guaranteed optimality by design, however for higher dimensionality of the problem it will likely run out of memory on most systems. The genetic search method implemented in `nimCSO` (see `geneticSearch()`) is a evolution strategy to iteratively improve solutions based on custom `mutate` and `crossover` procedures. Both procedures are of uniform type (Goldberg, 1989) with additional constraint of Hamming weight (Knuth, 2009) preservation in order to preserve order (number of considered elements) of parents and offspring. In `mutate` this is achieved by using purely random bit swapping, rather than more common flipping, as demonstrated in the Figure 2.

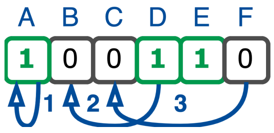


Figure 2: The schematic of `mutate` procedure where bits are swapping randomly, so that (1) bit can swap itself, (2) bits can swap causing a flip, or (3) bits can swap with no effect.

In `crossover`

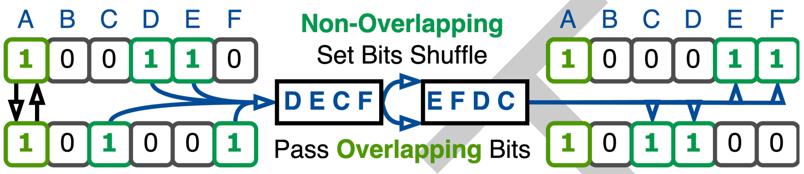


Figure 3: The schematic of uniform `crossover` procedure preserving Hamming weight implemented in `nimCSO`. Overlapping bits are passed directly, while non-overlapping bits are shuffled and distributed at positions present in one of the parents.

that iteratively improves a set of solutions by (1) mutating them and (2) crossing them over to create new solutions. The algorithm is designed to preserve the number of elements present (bits set) in their output solutions, which is a critical feature of the problem. The algorithm is primarily aimed at (1) problems with more than 40 elements, where neither `bruteForce` nor `algorithmSearch` are feasible and (2) at cases where the decent solution is needed quickly. Its implementation allows for arbitrary dimensionality of the problem and its time complexity will scale linearly with it. You may control a set of parameters to adjust the algorithm to your needs, including the number of initial randomly generated solutions `initialSolutionsN`, the number of solutions to keep carry over to the next iteration `searchWidth`, the maximum number of iterations `maxIterations`, the minimum number of iterations the solution has to fail to improve to be considered.

, but it becomes suboptimal for larger problems.

This custom genetic algotithm utilizes

procedures preserving the number of elements present (bits set) in their output solutions to iteratively improve a set of solutions.

It is primarily aimed at (1) problems with more than 40 elements, where neither `bruteForce_` nor `algorithmSearch_` are feasible and (2) at cases where the decent solution is needed quickly. Its implementation **allows for arbitrary dimensionality** of the problem and its time complexity will scale linearly with it. You may control a set of parameters to adjust the algorithm to your needs, including the number of initial randomly generated solutions `initialSolutionsN`, the number of solutions to keep carry over to the next iteration `searchWidth`, the maximum number of iterations `maxIterations`, the minimum number of iterations the solution has to fail to improve to be considered.

Use Examples

Tracking up to 11.9 million solutions.

Method	Time (s)	Memory (MB)
nimCSO (-d:release –threads:on)	302s	488 MB
nimCSO (-d:danger –threads:off)	302s	488 MB
NumPy (Python 3.11)	302s	488 MB
Dict Python 3.11	302s	488 MB

Contributions

A.M.K. was reponsible for conceptualization, methodology, software, testing and validation, writing of manuscript, and visualization; A.D. was responsible for testing software and results in training machine learning models; A.M.B., W.F.R., Z-K.L. were responsible for funding acquisition, review, and editing. Z-K.L. was also supervising the work.

Acknowledgements

This work has been funded through grants: **DOE-ARPA-E DE-AR0001435**, **NSF-POSE FAIN-2229690**, and **ONR N00014-23-2721**.

We would also like to acknowledge Dr. Jonathan Siegel at Texas A&M University for valuable discussions and feedback on the project.

References

Cantor, B., Chang, I. T. H., Knight, P., & Vincent, A. J. B. (2004). Microstructural development in equiatomic multicomponent alloys. *Materials Science and Engineering A*, *375-377*, 213–218. https://doi.org/10.1016/j.msea.2003.10.257

Debnath, A., Krajewski, A. M., Sun, H., Lin, S., Ahn, M., Li, W., Priya, S., Singh, J., Shang, S., Beese, A. M., Liu, Z.-K., & Reinhart, W. F. (2021). *Journal of Materials Informatics*, *1*. https://doi.org/10.20517/jmi.2021.05

Goldberg, D. E. (1989). *Genetic algorithms in search, optimization and machine learning* (1st ed.). Addison-Wesley Longman Publishing Co., Inc. ISBN: 0201157675

Knuth, D. E. (2009). *The art of computer programming, volume 4, fascicle 1: Bitwise tricks & techniques; binary decision diagrams* (12th ed.). Addison-Wesley Professional. ISBN: 0321580508

Krajewski, A. M., Beese, A. M., Reinhart, W. F., & Liu, Z.-K. (2024). *Efficient generation of grids and traversal graphs in compositional spaces towards exploration and path planning exemplified in materials*. http://arxiv.org/abs/2402.03528

Li, W., Raman, L., Debnath, A., Ahn, M., Lin, S., Krajewski, A. M., Shang, S., Priya, S., Reinhart, W. F., Liu, Z.-K., & Beese, A. M. (2024). *Design and validation of refractory alloys using machine learning, CALPHAD, and experiments*. https://doi.org/https://dx.doi.org/10.2139/ssrn.4689687

Rumpf, A. (2023). *Nim programming language v2.0.0*. https://nim-lang.org/

Senkov, O. N., Miracle, D. B., Chaput, K. J., & Couzinie, J.-P. (2018). Development and exploration of refractory high entropy alloys—a review. *Journal of Materials Research*, *33*, 3092–3128. https://doi.org/10.1557/jmr.2018.153

Williams, J. W. J. (1964). Algorithm 232 - heapsort. *Communications of the ACM*, *7*, 347–349. https://doi.org/10.1145/512274.512284

Yeh, J. W., Chen, S. K., Lin, S. J., Gan, J. Y., Chin, T. S., Shun, T. T., Tsau, C. H., & Chang, S. Y. (2004). Nanostructured high-entropy alloys with multiple principal elements: Novel alloy design concepts and outcomes. *Advanced Engineering Materials*, *6*, 299–303. https://doi.org/10.1002/adem.200300567