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Published: unpublished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 often encountered in materials science, where, for instance, datasets on Compositionally Complex Materials (CCMs) often span 20-45 chemical elements, 5-10 generalized processing histories, and several temperature regimes, for up to 60 total data components.

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 can outperform 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.

This tool employs a set of methods, ranging from (1) brute-force search through (2) genetic algorithms to (3) a newly designed search method. They use custom data structures and procedures written in Nim language, which are compile-time optimized for the specific problem statement and dataset pair, which allows nimCSO to run faster and use 1-2 orders of magnitude less memory than general-purpose data structures. All configuration is done with a simple human-readable config file, allowing easy modification of the search method and its parameters.

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 these applications are Compositionally Complex Materials (CCMs), and their matal-focused subset of Refractory High Entropy Alloys (RHEAs),

The Journal of Open Source Software

which are quickly growing since first proposed by (Cantor et al., 2004) and (Yeh et al., 2004). Contrary to most 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 a large palette of chemical elements. The resulting compositional spaces are both extremely vast and challenging to explore in terms of possible changes (Krajewski et al., 2024); thus, it becomes critical to answer the question like *"Which combination of 15 elements out of 60 in the dataset will result in the largest dataset?"* which has $\binom{60}{15}$ or 53 trillion possible solutions.

Methods and Performance

Overview

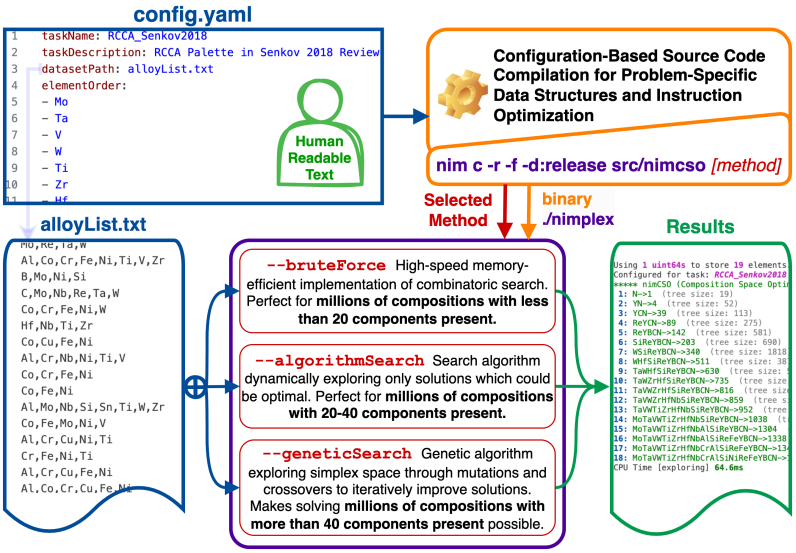

Figure 1: Schematic of core nimCSO data flow with a description of key methods. Metaprogramming is used to recompile the software optimized to the human-readable data and configuration files at hand.

The metaprogramming employed in nimCSO allows for static optimization of the code at the compile time.

The

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	Representation	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

Algorithmic Search

For highly dimensional problems (>20), the brute force search becomes suboptimal, prompting the need for a more efficient method. The algorithm implemented in `nimCSO` (see `algorithmSearch()`) 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. This method has generated the same results as combinatoric brute forcing in our tests, 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.

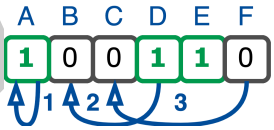

Figure 2: The schematic of `mutate` procedure where bits are swapping randomly, so that (1) bit can swap itself, (2) bits can swap causing a flip, or (3) bits can swap with no effect.

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.

Benchmarks

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

Acknowledgements

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

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/

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