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Published: unpublished

License
Authors of papers retain copyright and release the work under a Creative Commons Attribution 4.0 International License (CC BY 4.0).

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 high-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 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 the Methods and Performance section, nimCSO reaches the physical limits of the hardware (L1 cache latency) and can outperform an efficient native Python implementation over 100 times in terms of speed and 50 times in terms of memory usage (*not* counting interpreter), while also outperforming NumPy implementation 37 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 economics where factor selection affects performance of analytical (Fan et al., 2013) and ML (Peng et al., 2021) models, through medicine where drug interactions can have a significant impact on the treatment (Maher et al., 2014) (an escalating problem (Guthrie et al., 2015)) and understanding of microbial interactions can help fight gastrointenstant problems (Berg et al., 2022; Leeuwen et al., 2023), to 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

DRAFT

temperature materials for aerospace applications, through generative machine learning models (Debnath et al., 2021) driving thermodynamic modeling, alloy design, and manufacturing (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 have rapidly grown since first proposed by (Cantor et al., 2004) and (Yeh et al., 2004). Contrary to most of the traditional alloys, they contain many chemical elements (typically 4-9) in similar proportions in the hope of thermodynamically stabilizing 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), which encourages sampling from a large palette of chemical elements. At the time of writing, the ULTERA Database is the largest collection of HEA data, containing over 7,000 points manually extracted from 560 publications. It covers 37 chemical elements resulting in extremely large compositional spaces (Krajewski et al., 2024); thus, it becomes critical to answer questions like *"Which combination of how many elements will unlock the most expansive and simultaneously dense dataset?"* which has $2^{37} - 1$ or 137 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 (Evans et al., 2024).

Methods and Performance

Overview

As shown in Figure 1, `nimCSO` can be used as a user-tool based on human-readable configuration and a data file containing data "elements" which can be any strings representing problem-specific names of, e.g., market stocks, drug names, or chemical formulas. A single command is then used to recompile (`nim c -f`) and run (`-r`) problem (`-d:configPath=config.yaml`) with `nimCSO` (`src/nimcso`) using one of several methods. Advanced users can also quickly customize the provided methods with brief scripts using the `nimCSO` as a data-centric library.

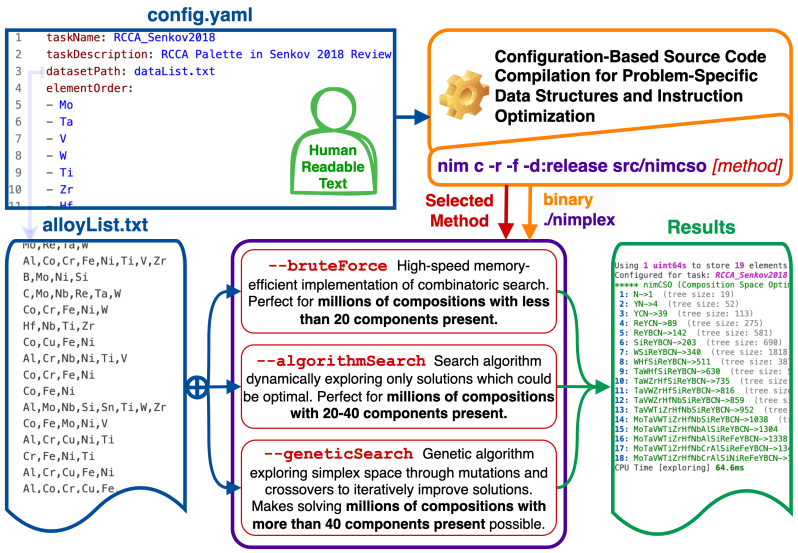

Figure 1: Schematic of core nimCSO data flow with a description of key methods. Metaprogramming is used to compile the software optimized to the human-readable data and configuration files at hand.

Internally, `nimCSO` is built around storing the data and solutions in one of two ways. The first

DRAFT

is as bits inside an integer (`uint64`), which allows for the 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 a particular `bruteForceInt` routine. The second one, used in `bruteForce`, `algorithmSearch`, and `geneticSearch`, implements a custom easily extensible `ElSolution` type containing heuristic value and `BitArray` payload, which is defined at compile time based on the configuration file to minimize necessary overheads. Both encodings outperform typical native Python and NumPy implementations, as shown in Table 1.

Table 1: Benchmarks of (1) the 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. Pre-processing time is excluded, as it has negligible impact on larger, realistic problems.

Tool	Object	Time per Dataset	Time per Entry *(Relative)*	Database Size *(Relative)*
`Python`$^{3.11}$	`set`	107.5 µs	50.0 ns *(x1)*	871.5 kB *(x1)*
`NumPy`$^{1.26}$	`array`	36.4 µs	16.9 ns *(x3.0)*	79.7 kB *(x10.9)*
`nimCSO`$^{0.6}$	`BitArray`	6.9 µs	3.2 ns *(x15.6)*	50.4 kB *(x17.3)*
`nimCSO`$^{0.6}$	`uint64`	0.98 µs	0.456 ns *(x110)*	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 N considered elements are represented as a range of integers from 0 to $2^N - 1$, and used to initialize `uint64`/`BitArrays` on the fly. 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 beyond approximately 30 elements; however, the one based on `BitArray` could be easily extended if needed.

Algorithm-Based Search

The algorithm implemented in the `algorithmSearch` routine, targeting high dimensional problems (20-50), 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 to (2) avoid exhaustive search by inhibiting 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, making the algorithm computationally feasible in higher-dimensional problems at the cost of potentially suboptimal results. In the authors' tests over HEAs, this algorithm has generated correct results (agreeing with an exhaustive `bruteForce` exploration), except for occasional differences in the last explored solution, where almost all data was discarded, and a few remaining points were highly independent.

Genetic Search

Beyond 50 components, the algorithm-based method will likely run out of memory on most personal systems. The `geneticSearch` routine resolves this issue through an evolution strategy

DRAFT

to iteratively improve solutions based on custom `mutate` and `crossover` procedures. Both are of uniform type (Goldberg, 1989) with additional constraint of Hamming weight (Knuth, 2009) preservation in order to preserve number of considered elements in 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.

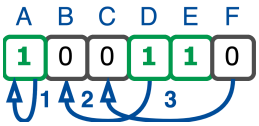

Figure 2: Schematic of `mutate` procedure where bits are swapping randomly, so that (1) bit can swap itself, (2) bits can swap causing a flip, or (3) bits can swap with no effect.

Meanwhile, in `crossover`, this constraint is satisfied by passing overlapping bits directly, while non-overlapping bits are shuffled and distributed at positions present in one of the parents, as shown in Figure 3.

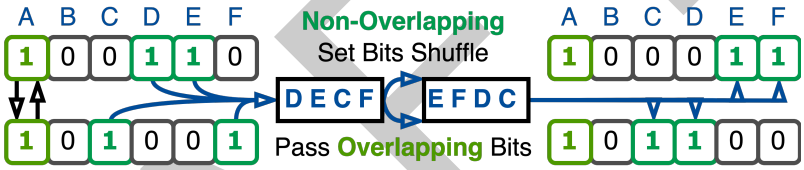

Figure 3: Schematic of uniform `crossover` procedure preserving Hamming weight implemented in `nimCSO`.

The above are applied iteratively, with best solutions carried to next generation, until the solution converges or the maximum number of iterations is reached. Unlike the other methods, the present method is not limited by the number of components and lets user control both time and memory requirements, either to make big problems feasible or to get a good-enough solution quickly in small problems. However, it comes with no optimality guarantees.

Use Examples

The tool comes with two pre-defined example problems to demonstrate its use. The first one is defined in the default `config.yaml` file and goes through the complete dataset of 2,150 data points spanning 37 components in `dataList.txt` based on the ULTERA Dataset (Debnath et al., 2021). It is intended to showcase `algorithmSearch`/`-as` and `geneticSearch`/`-gs` methods, as brute-forcing would take around one day. The second one is defined in `config_rhea.yaml` and uses the same dataset but a limited scope of components critical to RHEAs (Senkov et al., 2018) and is intended to showcase `bruteForce`/`-bf` and `bruteForceInt`/`-bfi` methods. With four simple commands (see Table 2), the user can compare the methods' performance and the solutions' quality.

Table 2: Four example tasks alongside typical CPU time and memory usage on Apple M2 Max.

Task Definition (`nim c -r -f -d:release ...`)	Time (s)	Memory (MB)
`-d:configPath=config.yaml src/nimcso -as`	308s	488 MB
`-d:configPath=config.yaml src/nimcso -gs`	5.10s	3.2 MB
`-d:configPath=config_rhea.yaml src/nimcso -as`	0.073s	2.2 MB

DRAFT

Task Definition (`nim c -r -f -d:release ...`)	Time (s)	Memory (MB)
`-d:configPath=config_rhea.yaml src/nimcso -gs`	0.426s	2.1 MB
`-d:configPath=config_rhea.yaml src/nimcso -bf`	3.726s	2.0 MB
`-d:configPath=config_rhea.yaml src/nimcso -bfi`	0.495s	2.0 MB

In case of issues, the help message can be accessed by running the tool with `-h` flag or by refering to documentation at amkrajewski.github.io/nimCSO.

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 several valuable discussions and feedback on the project, and Dr. Bradley Dice for his code contributions to the project during the review process, such as optimization of the native Python implementation.

References

Berg, N. I. van den, Machado, D., Santos, S., Rocha, I., Chacón, J., Harcombe, W., Mitri, S., & Patil, K. R. (2022). Ecological modelling approaches for predicting emergent properties in microbial communities. *Nature Ecology & Evolution*, *6*, 855–865. https://doi.org/10.1038/s41559-022-01746-7

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

Evans, M. L., Bergsma, J., Merkys, A., Andersen, C. W., Andersson, O. B., Beltrán, D., Blokhin, E., Boland, T. M., Balderas, R. C., Choudhary, K., Díaz, A. D., García, R. D., Eckert, H., Eimre, K., Montero, M. E. F., Krajewski, A. M., Mortensen, J. J., Duarte, J. M. N., Pietryga, J., … Armiento, R. (2024). *Developments and applications of the OPTIMADE API for materials discovery, design, and data exchange*. https://doi.org/10.48550/arXiv.2402.00572

Fan, J., Liao, Y., & Mincheva, M. (2013). Large covariance estimation by thresholding principal orthogonal complements. *Journal of the Royal Statistical Society Series B: Statistical Methodology*, *75*, 603–680. https://doi.org/10.1111/rssb.12016

Goldberg, D. E. (1989). *Genetic algorithms in search, optimization and machine learning* (1st ed.). Addison-Wesley Longman Publishing Co., Inc. https://doi.org/10.5860/choice.27-0936

Guthrie, B., Makubate, B., Hernandez-Santiago, V., & Dreischulte, T. (2015). The rising tide of polypharmacy and drug-drug interactions: Population database analysis 1995–2010.

DRAFT

BMC Medicine, *13*, 74. https://doi.org/10.1186/s12916-015-0322-7

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*. https://doi.org/10.48550/arXiv.2402.03528

Leeuwen, P. T. van, Brul, S., Zhang, J., & Wortel, M. T. (2023). Synthetic microbial communities (SynComs) of the human gut: Design, assembly, and applications. *FEMS Microbiology Reviews*, *47*. https://doi.org/10.1093/femsre/fuad012

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. *International Journal of Refractory Metals and Hard Materials*, *121*, 106673. https://doi.org/10.1016/j.ijrmhm.2024.106673

Maher, R. L., Hanlon, J., & Hajjar, E. R. (2014). Clinical consequences of polypharmacy in elderly. *Expert Opinion on Drug Safety*, *13*, 57–65. https://doi.org/10.1517/14740338.2013.827660

Peng, Y., Albuquerque, P. H. M., Kimura, H., & Saavedra, C. A. P. B. (2021). Feature selection and deep neural networks for stock price direction forecasting using technical analysis indicators. *Machine Learning with Applications*, *5*, 100060. https://doi.org/10.1016/j.mlwa.2021.100060

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

DRAFT