Open Source Evolutionary Computation with Chips-n-Salsa

Vincent A. Cicirello [a]

Computer Science, School of Business, Stockton University, 101 Vera King Farris Dr, Galloway, NJ, USA
vincent.cicirello@stockton.edu

Abstract: When it was first introduced, the Chips-n-Salsa Java library provided stochastic local search and related algorithms, with a focus on self-adaptation and parallel execution. For the past four years, we expanded its scope to include evolutionary computation. This paper concerns the evolutionary algorithms that Chips-n-Salsa now provides, which includes multiple evolutionary models, common problem representations, a wide range of mutation and crossover operators, and a variety of benchmark problems. Well-defined Java interfaces enable easily integrating custom representations and evolutionary operators, as well as defining optimization problems. Chips-n-Salsa's evolutionary algorithms include implementations with adaptive mutation and crossover rates, as well as both sequential and parallel execution. Source code is maintained on GitHub, and immutable artifacts are regularly published to the Maven Central Repository to enable easily importing into projects for reproducible builds. Effective development processes such as test-driven development, as well as a variety of static analysis tools help ensure code quality.

1 INTRODUCTION

Evolutionary computation refers to the family of problem solving frameworks that are inspired by models of natural evolution and genetics. Most forms of evolutionary computation are population-based, maintaining a population of many candidate solutions to a problem, which evolve over many generations using operators that mimic evolutionary processes such as mutation and recombination. Evolutionary computation is often used to solve a wide variety of problems, including in software engineering (Sobania et al., 2023; Arcuri et al., 2021; Petke et al., 2018), computer vision and image processing (Wan et al., 2023; Bi et al., 2023), neural network construction (Zhou et al., 2021), engineering design (Tayarani-N. et al., 2015), production scheduling (Branke et al., 2016), finance (Ponsich et al., 2013), graph theory (Pizzuti, 2018), feature selection (Xue et al., 2016), data mining (Mukhopadhyay et al., 2014), multiobjective optimization (Liang et al., 2023), dynamic optimization problems (Yazdani et al., 2021), among many others.

Previously, we introduced Chips-n-Salsa (Cicirello, 2020), an open source Java library for stochastic local search and related algorithms. At

the time, Chips-n-Salsa version 1.3.0 did not include evolutionary computation. Instead it focused on a variety of other metaheuristics, such as hill climbing (Hoos and Stützle, 2018; Selman and Gomes, 2006; Prügel-Bennett, 2004), simulated annealing (Delahaye et al., 2019), and stochastic sampling search algorithms (Grasas et al., 2017; Cicirello and Smith, 2005; Gomes et al., 1998; Bresina, 1996; Langley, 1992). Chips-n-Salsa supports self-adaptive search, such as adaptive annealing schedules (Cicirello, 2021; Hubin, 2019; Štefankovič et al., 2009) for simulated annealing, and adaptive restart schedules (Cicirello, 2017; Luby et al., 1993) for multi-start search. Chips-n-Salsa is also designed to easily support parallel search, as well as to enable defining hybrids of multiple metaheuristics. The algorithm implementations are highly customizable such as in choice of search operators, including providing well-defined interfaces to enable the option to integrate custom components with library components. Chips-n-Salsa currently requires Java 17 at minimum. Source code is maintained on GitHub, and API documentation and other information available on the web. Immutable artifacts of every release, including precompiled jar of the library and jars of the source and documentation, are regularly published to the Maven Central Repository to enable easily importing into

---

[a] https://orcid.org/0000-0003-1072-8559

Table 1: URLs for Chips-n-Salsa.

Source	https://github.com/cicirello/Chips-n-Salsa
Website	https://chips-n-salsa.cicirello.org/
Maven	https://central.sonatype.com/artifact/org.cicirello/chips-n-salsa/
Docs	https://chips-n-salsa.cicirello.org/api/
Examples	https://github.com/cicirello/chips-n-salsa-examples

software development projects, as well as to ensure reproducible builds. See Table 1 for library URLs.

During the four years since that previous publication, there have been six major releases of Chips-n-Salsa, whose scope we expanded to include evolutionary computation. This paper focuses on version 7.0.0, and more specifically on the genetic algorithms (GA) and other evolutionary algorithms (EA) that the library now provides. Chips-n-Salsa supports multiple forms of EA, such as the classic generational EA as well as the $(\mu + \lambda)$-EA, and also includes many crossover and mutation operators for different representations, such as for optimizing bit-vectors, permutations, and vectors of reals and integers. Additionally, to complement Chips-n-Salsa's emphasis on self-adaptation, it includes an adaptive EA where the crossover and mutation rates evolve during the search. We have also added many common benchmarking problems to the library. The objectives of doing so include serving as a research testbed, and supporting reproducible empirical evolutionary computation research. Reproducible research (National Academies, 2019) is crucial in all fields of science and engineering.

In the remainder of this paper, we cover the EA functionality of Chips-n-Salsa as of version 7.0.0. We begin with a brief discussion of related evolutionary computation libraries in Section 2. Then, in Section 3, we present the EAs of Chips-n-Salsa, along with a discussion of key features of the implementations, and a discussion of the crossover and mutation operators available for each supported representation. We also summarize the benchmark problems implemented within the library. Our objective isn't only to enable reproducible research, but also to provide a production ready open source toolkit for use by practitioners. Thus, we employ high-quality development practices, which we summarize in Section 4. We wrap up in Section 5.

2 RELATED WORK

There are several other open source libraries available for evolutionary computation (Jenetics, 2024; Tarkowski, 2023; de Dios and Mezura-Montes, 2022; Izzo and Biscani, 2020; Scott and Luke, 2019; Bell, 2019; Gijsbers and Vanschoren, 2019; Detorakis and Burton, 2019; Simson, 2019).

Many focus on a specific form of evolutionary computation. For example, LGP (Simson, 2019) implements genetic programming (GP) in Kotlin. DCGP (Izzo and Biscani, 2020) is also a GP library, but in C++ with a Python interface. Metaheuristics (de Dios and Mezura-Montes, 2022) is a Julia package consisting of several metaheuristics for both single-objective and multi-objective optimization. CEGO (Bell, 2019) is a differential evolution C++ library with a Python wrapper. Quilë (Tarkowski, 2023) and GAIM (Detorakis and Burton, 2019) are both C++ libraries for GAs, with GAIM focused specifically on multi-population island models. ECJ (Scott and Luke, 2019) and Jenetics (Jenetics, 2024) are both Java libraries supporting multiple forms of evolutionary computation.

Chips-n-Salsa differs from the existing libraries in a few ways. First, it supports both evolutionary computation as well as other metaheuristics. Due to this, it is straightforward to create hybrids of multiple techniques. Second, observing that much of the runtime of an EA is spent generating random numbers, we have highly optimized the randomness of Chips-n-Salsa utilizing the $\rho\mu$ library (Cicirello, 2022b). Third, Chips-n-Salsa includes more built-in support for evolving permutations compared to other libraries, with a comprehensive collection of evolutionary permutation operators (Cicirello, 2023). Additionally, Chips-n-Salsa is designed to enable parallel execution.

3 CORE FUNCTIONALITY

This Section provides an overview of the core evolutionary computation functionality of the Chips-n-Salsa library. It is organized into subsections covering the EA features and characteristics (Section 3.1), the evolutionary operators provided by the library (Section 3.2), and the available benchmark problems (Section 3.3).

3.1 Evolutionary Algorithms

Evolutionary Models: Chips-n-Salsa provides implementations of both generational EAs, where each

```java
public interface Copyable <T extends
 Copyable <T>> {
 |  T copy(T c);
}
```
Listing 1: Interface to define custom representation.

generation involves replacing the current population with an offspring population produced via application of the crossover and mutation operators, as well as steady-state EAs, where a small number of children are generated at a time, replacing a small number of members of the population. For example, the $(\mu + \lambda)$-EA maintains a population of size μ, generates a small number of children λ, and keeps the μ best of the combination. The simplest form of steady-state EA is the $(\mu + 1)$-EA that generates a single offspring at a time. The library also includes the special case of the $(1 + 1)$-EA. Chips-n-Salsa additionally supports the special case of a mutation-only EA. The generational EAs include an option for elitism, where a small number of the best population members survive without undergoing crossover or mutation.

Representations: Chips-n-Salsa provides several built-in representations, including efficient bit-vectors suitable for a GA, as well as vectors of integers and reals, such as for real-valued function optimization. Since many combinatorial optimization problems concern searching for an optimal ordering, Chips-n-Salsa supports evolving permutations utilizing an efficient permutation representation (Cicirello, 2018). Section 3.2 focuses on the evolutionary operators available for each of these representations.

Customizable: If the built-in representations are unsuitable for a problem, Chips-n-Salsa supports custom representations via Java generic types and by implementing a Java interface `Copyable` as seen in Listing 1, which enables the library to create identical copies of population members as needed in a representation-independent manner. Defining evolutionary operators for the new representation is accomplished by implementing the `MutationOperator` and `CrossoverOperator` interfaces shown in Listing 2. Those same interfaces also enable implementing custom crossover and mutation operators for the built-in representations.

Adaptive: In addition to the more common case of control parameters that remain constant throughout a run, Chips-n-Salsa includes an implementation of an adaptive EA that encodes the crossover and mutation rates as part of each population member, using Gaussian mutation to evolve these during the search (Hinterding, 1995).

Parallel: All metaheuristics in the library implement a set of Java interfaces, enabling the EA im-

```java
public interface MutationOperator <T>
 extends Splittable <MutationOperator
 <T>> {
 |  void mutate(T c);
}


public interface CrossoverOperator <T>
 extends Splittable <CrossoverOperator
 <T>> {
 |  void cross(T c1, T c2);
}


public interface Splittable <T extends
 Splittable <T>> {
 |  T split();
}
```
Listing 2: Interfaces for evolutionary operators.

plementations to utilize the library's existing parallel architecture for multi-populations to accelerate runtime on multicore systems. One of these interfaces is the `Splittable` interface shown earlier in Listing 2, which enables the library's parallel architecture to replicate the functionality of operators, etc when spawning threads.

Hybridization: Hybrids of EA with other search algorithms are common. For example, a memetic algorithm combines an EA with local search (Neri and Cotta, 2012). Creating such hybrids in Chips-n-Salsa is straightforward, facilitated by its plug-and-play design.

Configurable and optimized randomness: The runtime of an EA or GA can be significantly impacted by the choice of pseudorandom number generator (PRNG) (Nesmachnow et al., 2015), as well as the choice of algorithm for generating values from specific distributions (e.g., uniform subject to bounds, Gaussian, Cauchy, etc). For this reason, we carefully optimized all random behavior within Chips-n-Salsa utilizing the enhanced random functionality of the $\rho\mu$ library (Cicirello, 2022b). Additionally, by default, Chips-n-Salsa uses a carefully chosen, efficient PRNG, but also provides the option to configure the PRNG through its `Configurator` class whose interface is shown in Listing 3.

3.2 Evolutionary Operators

Selection Operators: The selection operators (Mitchell, 1998) include fitness proportionate selection (i.e., weighted roulette wheel), truncation selection, tournament selection, stochastic universal sampling (SUS), linear rank selection, exponential rank selection, Boltzmann selection, and random

```
public class Configurator {
    public static void
      configureRandomGenerator(long seed);
    public static void
      configureRandomGenerator(
      RandomGenerator.SplittableGenerator
      r);
}
```

Listing 3: Random generator configuration.

```
public interface SelectionOperator extends
  Splittable <SelectionOperator> {
    default void init(int generations);
    void select(
      PopulationFitnessVector.Double fitnesses,
      int[] selected);
    void select(
      PopulationFitnessVector.Integer fitnesses,
      int[] selected);
}
```

Listing 4: Interface for defining selection operators.

selection. There are two forms of each of linear rank, exponential rank, and Boltzmann selection: one that operates like roulette wheel and one that operates like SUS. There are also options to transform fitness values relative to the population fitness during selection, such as with sigma scaling or by shifting the fitness scale. An interface, SelectionOperator, is provided to enable defining custom selection operators (see Listing 4).

Evolutionary Operators for Bit Vectors: Chips-n-Salsa supports the common bit-flip mutation for mutating bit vectors, and all of the common crossover operators, such as single-point, two-point, k-point, and uniform crossover operators.

Evolutionary Operators for Integer Vectors and Real Vectors: For vectors of integers and reals, the library includes the obvious extensions of single-point, two-point, k-point, and uniform crossover. For mutating real vectors, the library includes Gaussian mutation (Hinterding, 1995), Cauchy mutation (Szu and Hartley, 1987), and uniform mutation. For integer vectors, it includes a uniform mutation (e.g., that adds a uniform random value from $[-w, w]$ to components of the vector), as well as a random value change mutation that replaces a value with a different random value from its domain.

Evolutionary Operators for Permutations: Many evolutionary operators exist for evolving permutations (Cicirello, 2023). Chips-n-Salsa provides an extensive collection of crossover and mutation operators for permutations. Mutation operators include all of the common permutation mutation op-

erators (Cicirello, 2023; Serpell and Smith, 2010; Eiben and Smith, 2003; Valenzuela, 2001) as well as a few less common ones. The mutation operators for permutations include: swap, adjacent swap, insertion, reversal, scramble, uniform scramble, cycle mutation (Cicirello, 2022a), 3opt (Lin, 1965), block move, block swap, as well as window-limited mutation operators (Cicirello, 2014). Crossover operators include: cycle crossover (Oliver et al., 1987), position based crossover (Barecke and De-tyniecki, 2007), order crossover (Davis, 1985), order crossover 2 (Syswerda, 1991), non-wrapping order crossover (Cicirello, 2006), uniform order based crossover (Syswerda, 1991), partially matched crossover (Goldberg and Lingle, 1985), uniform partially matched crossover (Cicirello and Smith, 2000), edge recombination (Whitley et al., 1989), enhanced edge recombination (Starkweather et al., 1991), precedence preservative crossover (Bierwirth et al., 1996), and uniform precedence preservative crossover (Bierwirth et al., 1996).

Hybrid Operators: The library includes support for utilizing hybrid crossover operators and hybrid mutation operators. Specifically, it includes classes to integrate multiple crossover operators or mutation operators, such that an operator is chosen randomly from a specified set during each application. The random choice can be weighted (e.g., to choose one operator with higher probability than another) or it can be a uniform random choice.

3.3 Benchmark Problems

To solve optimization problems using the EAs of the library, you either implement the fitness function directly via a Java interface, or if it is a minimization problem, you can implement its cost function via a Java interface, and then use one of the library's classes for mapping a cost function (e.g., minimization) to a fitness function (e.g., maximization). Listing 5 shows the interfaces for defining fitness functions; and Listing 6 shows the interfaces for defining optimization problems (OptimizationProblem for real-valued costs and IntegerCostOptimizationProblem for integer-valued costs). The library treats optimization functions and fitness functions as distinct concepts.

Chips-n-Salsa includes many common benchmarking problems. For example, it includes implementations of all of Ackley's (Ackley, 1985; Ackley, 1987) classic problems for bit vectors, such as one-max, two-max, trap, porcupine, plateaus, and mix, as well as various royal roads problems (Mitchell et al., 1992; Holland, 1993; Jones, 1994). It includes some real-valued function optimization problems (Forrester

```java
public interface FitnessFunction.Double <T
  extends Copyable <T>> extends
  FitnessFunction <T> {
    double fitness(T candidate);
}
```

```java
public interface FitnessFunction.Integer
  <T extends Copyable <T>> extends
  FitnessFunction <T> {
    int fitness(T candidate);
}
```

```java
public interface FitnessFunction <T extends
  Copyable <T>> {
    Problem <T> getProblem();
}
```

Listing 5: Interfaces to define fitness functions.

et al., 2008; Gramacy and Lee, 2012). It also includes the permutation in a haystack (Cicirello, 2016), a benchmarking problem for permutations, as well as many NP-Hard combinatorial optimization problems (Garey and Johnson, 1979), such as the traveling salesperson, bin packing, largest common subgraph, quadratic assignment, and many scheduling problems.

4 DEVELOPMENT PRACTICES

In developing Chips-n-Salsa, we utilize best practices in software engineering, such as test-driven development, integrating static analysis tools into the build pipeline, and continuous integration and continuous deployment (CI/CD). Frequent public releases are deployed to artifact repositories (e.g., Maven Central and GitHub Packages) to ease potential integration into other open source projects. The immutable nature of the artifacts published to the Maven Central Repository help ensure builds of projects that depend upon Chips-n-Salsa are reproducible.

A few of the quality control methods used in developing Chips-n-Salsa are as follows:

Unit testing: Following test-driven development practices, unit tests are written for all components in the JUnit framework.

Regression testing: All new and existing test cases must pass before changes are accepted, to ensure that new functionality does not introduce bugs into existing components.

Test coverage: We use the test coverage tool, JaCoCo (Hoffmann et al., 2024), to compute both C0 coverage (instructions coverage) and C1 coverage (branches coverage) for all builds.

```java
public interface OptimizationProblem <T
  extends Copyable <T>> extends Problem
  <T> {
    double cost(T candidate);
    double value(T candidate);
    default double costAsDouble(T
      candidate);
    default SolutionCostPair<T>
      getSolutionCostPair(T candidate);
    default boolean isMinCost(double cost);
    default double minCost();
}
```

```java
public interface
  IntegerCostOptimizationProblem <T
  extends Copyable <T>> extends Problem
  <T> {
    int cost(T candidate);
    int value(T candidate);
    default double costAsDouble(T
      candidate);
    default SolutionCostPair<T>
      getSolutionCostPair(T candidate);
    default boolean isMinCost(int cost);
    default int minCost();
}
```

```java
public interface Problem <T extends Copyable
  <T>> {
    double costAsDouble(T candidate);
    SolutionCostPair<T>
      getSolutionCostPair(T candidate);
}
```

Listing 6: Interfaces to define problems.

Pull-request checks: Chips-n-Salsa is developed openly on GitHub. GitHub's built-in CI/CD framework, GitHub Actions (GitHub, 2024c), is used during pull-requests to verify that all test cases pass, and to run the test coverage analysis.

Static analysis: The build pipeline runs several static analysis tools to automatically detect error prone code, including: CodeQL (GitHub, 2024a), RefactorFirst (Bethancourt, 2024), SpotBugs (SpotBugs, 2024), FindSecBugs (Arteau, 2024), and Snyk (Snyk, 2024).

Code style: For consistency, we use Google Java Style (Google, 2024), and automatically reformat to this style using a Maven plugin (Spotify, 2024).

Dependency management: We keep dependencies up to date with dependabot (GitHub, 2024b).

Documentation site: The project website, among other things, contains the Javadoc formatted documentation of the library (URL in Table 1), which is

automatically updated during the release process.

Example code: In addition to the repository of the library itself, an additional GitHub repository (URL in Table 1) provides a collection of detailed examples demonstrating how to use Chips-n-Salsa in projects.

Chips-n-Salsa is licensed via the GNU General Public License v3.0 (Free Software Foundation, 2007), and it welcomes contributions from the open source community, with well-defined guidelines for contributors.

5 CONCLUSION

Evolutionary computation is widely used in many fields. Chips-n-Salsa provides a well-engineered open source framework for EA and related metaheuristics, including a comprehensive set of evolutionary operators for common representations like bit vectors, permutations, and vectors of integers or reals, as well as implementations of many common benchmarking problems. It thus can serve as an ideal framework for empirical research. For example, if a researcher implements their new EA using the library, they benefit from an easy way to compare their approach to many others as well as on many problems. Deploying immutable software artifacts of each version of the library to Maven Central better enables reproducible research (National Academies, 2019), as future runs of experiments can use the exact versions of components as the original runs.

REFERENCES

Ackley, D. H. (1985). A connectionist algorithm for genetic search. In *ICGA*, pages 121–135.

Ackley, D. H. (1987). An empirical study of bit vector function optimization. In *Genetic Algorithms and Simulated Annealing*, pages 170–204. Morgan Kaufmann.

Arcuri, A., Galeotti, J. P., Marculescu, B., and Zhang, M. (2021). Evomaster: A search-based system test generation tool. *JOSS*, 6(57):2153.

Arteau, P. (2024). Find security bugs: The spotbugs plugin for security audits of java web applications. https://find-sec-bugs.github.io/.

Barecke, T. and Detyniecki, M. (2007). Memetic algorithms for inexact graph matching. In *IEEE CEC*, pages 4238–4245.

Bell, I. H. (2019). CEGO: C++11 evolutionary global optimization. *JOSS*, 4(36):1147.

Bethancourt, J. (2024). RefactorFirst. https://github.com/refactorfirst/RefactorFirst.

Bi, Y., Xue, B., Mesejo, P., Cagnoni, S., and Zhang, M. (2023). A survey on evolutionary computation for computer vision and image analysis: Past, present, and future trends. *IEEE TEVC*, 27(1):5–25.

Bierwirth, C., Mattfeld, D. C., and Kopfer, H. (1996). On permutation representations for scheduling problems. In *PPSN*, pages 310–318.

Branke, J., Nguyen, S., Pickardt, C. W., and Zhang, M. (2016). Automated design of production scheduling heuristics: A review. *IEEE TEVC*, 20(1):110–124.

Bresina, J. L. (1996). Heuristic-biased stochastic sampling. In *AAAI*, pages 271–278.

Cicirello, V. A. (2006). Non-wrapping order crossover: An order preserving crossover operator that respects absolute position. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 1125–1131.

Cicirello, V. A. (2014). On the effects of window-limits on the distance profiles of permutation neighborhood operators. In *International Conference on Bio-inspired Information and Communication Technologies*, pages 28–35.

Cicirello, V. A. (2016). The permutation in a haystack problem and the calculus of search landscapes. *IEEE Transactions on Evolutionary Computation*, 20(3):434–446.

Cicirello, V. A. (2017). Variable annealing length and parallelism in simulated annealing. In *International Symposium on Combinatorial Search*, pages 2–10.

Cicirello, V. A. (2018). JavaPermutationTools: A java library of permutation distance metrics. *Journal of Open Source Software*, 3(31):950.

Cicirello, V. A. (2020). Chips-n-Salsa: A java library of customizable, hybridizable, iterative, parallel, stochastic, and self-adaptive local search algorithms. *Journal of Open Source Software*, 5(52):2448.

Cicirello, V. A. (2021). Self-tuning lam annealing: Learning hyperparameters while problem solving. *Applied Sciences*, 11(21):9828.

Cicirello, V. A. (2022a). Cycle mutation: Evolving permutations via cycle induction. *Applied Sciences*, 12(11):5506.

Cicirello, V. A. (2022b). $\rho\mu$: A java library of randomization enhancements and other math utilities. *Journal of Open Source Software*, 7(76):4663.

Cicirello, V. A. (2023). A survey and analysis of evolutionary operators for permutations. In *Proceedings of the 15th International Joint Conference on Computational Intelligence*, pages 288–299.

Cicirello, V. A. and Smith, S. F. (2000). Modeling ga performance for control parameter optimization. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 235–242.

Cicirello, V. A. and Smith, S. F. (2005). Enhancing stochastic search performance by value-biased randomization of heuristics. *Journal of Heuristics*, 11(1):5–34.

Davis, L. (1985). Applying adaptive algorithms to epistatic domains. In *IJCAI*, pages 162–164.

de Dios, J.-A. M. and Mezura-Montes, E. (2022). Metaheuristics: A julia package for single- and multi-objective optimization. *JOSS*, 7(78):4723.

Delahaye, D., Chaimatanan, S., and Mongeau, M. (2019). Simulated annealing: From basics to applications. In *Handbook of Metaheuristics*, pages 1–35. Springer.

Detorakis, G. and Burton, A. (2019). Gaim: A c++ library for genetic algorithms and island models. *JOSS*, 4(44):1839.

Eiben, A. E. and Smith, J. E. (2003). *Introduction to Evolutionary Computing*. Springer, Berlin, Germany.

Forrester, A. I. J., Sóbester, A., and Keane, A. J. (2008). Appendix: Example problems. In *Engineering Design via Surrogate Modelling: A Practical Guide*, pages 195–203. Wiley.

Free Software Foundation (2007). Gnu general public license. https://www.gnu.org/licenses/gpl-3.0.en.html.

Garey, M. R. and Johnson, D. S. (1979). *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., USA.

Gijsbers, P. and Vanschoren, J. (2019). Gama: Genetic automated machine learning assistant. *JOSS*, 4(33):1132.

GitHub (2024a). CodeQL. https://codeql.github.com/.

GitHub (2024b). Dependabot. https://github.com/dependabot/dependabot-core.

GitHub (2024c). Github actions. https://github.com/features/actions.

Goldberg, D. E. and Lingle, R. (1985). Alleles, loci, and the traveling salesman problem. In *ICGA*, pages 154–159.

Gomes, C. P., Selman, B., and Kautz, H. (1998). Boosting combinatorial search through randomization. In *AAAI*, pages 431–437.

Google (2024). Google java style guide. https://google.github.io/styleguide/.

Gramacy, R. B. and Lee, H. K. H. (2012). Cases for the nugget in modeling computer experiments. *Statistics and Computing*, 22(3):713–722.

Grasas, A., Juan, A. A., Faulin, J., de Armas, J., and Ramalhinho, H. (2017). Biased randomization of heuristics using skewed probability distributions: A survey and some applications. *CAIE*, 110:216–228.

Hinterding, R. (1995). Gaussian mutation and self-adaption for numeric genetic algorithms. In *CEC*, pages 384–389.

Hoffmann, M. R., Janiczak, B., and Mandrikov, E. (2024). JaCoCo java code coverage library. https://www.jacoco.org/jacoco/.

Holland, J. (1993). Royal road functions. *Internet Genetic Algorithms Digest*, 7(22).

Hoos, H. H. and Stützle, T. (2018). Stochastic local search. In *Handbook of Approximation Algorithms and Metaheuristics Methologies and Traditional Applications*, chapter 17. Chapman and Hall/CRC, 2nd edition.

Hubin, A. (2019). An adaptive simulated annealing em algorithm for inference on non-homogeneous hidden markov models. In *AIIPCC*, pages 1–9.

Izzo, D. and Biscani, F. (2020). dcgp: Differentiable cartesian genetic programming made easy. *JOSS*, 5(51):2290.

Jenetics (2024). Jenetics: Genetic algorithm, genetic programming, evolutionary algorithm, and multi-objective optimization. https://jenetics.io/.

Jones, T. (1994). A description of holland's royal road function. *Evolutionary Computation*, 2(4):409–415.

Langley, P. (1992). Systematic and nonsystematic search strategies. In *AIPS*, pages 145–152.

Liang, J., Ban, X., Yu, K., Qu, B., Qiao, K., Yue, C., Chen, K., and Tan, K. C. (2023). A survey on evolutionary constrained multiobjective optimization. *IEEE TEVC*, 27(2):201–221.

Lin, S. (1965). Computer solutions of the traveling salesman problem. *The Bell System Technical Journal*, 44(10):2245–2269.

Luby, M., Sinclair, A., and Zuckerman, D. (1993). Optimal speedup of las vegas algorithms. *Inf Process Lett*, 47(4):173–180.

Mitchell, M. (1998). *An Introduction to Genetic Algorithms*. MIT Press, Cambridge, MA.

Mitchell, M., Forrest, S., and Holland, J. (1992). The royal road for genetic algorithms: Fitness landscapes and ga performance. In *ECAL*.

Mukhopadhyay, A., Maulik, U., Bandyopadhyay, S., and Coello, C. A. C. (2014). A survey of multiobjective evolutionary algorithms for data mining: Part i. *IEEE TEVC*, 18(1):4–19.

National Academies (2019). *Reproducibility and Replicability in Science*. National Academies Press, Washington, DC.

Neri, F. and Cotta, C. (2012). Memetic algorithms and memetic computing optimization: A literature review. *Swarm and Evolutionary Computation*, 2:1–14.

Nesmachnow, S., Luna, F., and Alba, E. (2015). An empirical time analysis of evolutionary algorithms as c programs. *Softw Pract Exp*, 45(1):111–142.

Oliver, I. M., Smith, D. J., and Holland, J. R. C. (1987). A study of permutation crossover operators on the traveling salesman problem. In *Int Conf on Genetic Algorithms and Their Application*, pages 224–230.

Petke, J., Haraldsson, S. O., Harman, M., Langdon, W. B., White, D. R., and Woodward, J. R. (2018). Genetic improvement of software: A comprehensive survey. *IEEE TEVC*, 22(3):415–432.

Pizzuti, C. (2018). Evolutionary computation for community detection in networks: A review. *IEEE TEVC*, 22(3):464–483.

Ponsich, A., Jaimes, A. L., and Coello, C. A. C. (2013). A survey on multiobjective evolutionary algorithms for the solution of the portfolio optimization problem and other finance and economics applications. *IEEE TEVC*, 17(3):321–344.

Prügel-Bennett, A. (2004). When a genetic algorithm outperforms hill-climbing. *TCS*, 320(1):135–153.

Scott, E. O. and Luke, S. (2019). Ecj at 20: Toward a general metaheuristics toolkit. In *GECCO*, pages 1391–1398.

Selman, B. and Gomes, C. P. (2006). Hill-climbing search. In *Encyclopedia of Cognitive Science*, pages 333–336. Wiley.

Serpell, M. and Smith, J. E. (2010). Self-adaptation of mutation operator and probability for permutation representations in genetic algorithms. *Evolutionary Computation*, 18(3):491–514.

Simson, J. (2019). Lgp: A robust linear genetic programming implementation on the jvm using kotlin. *JOSS*, 4(42):1337.

Snyk (2024). Snyk. https://snyk.io/.

Sobania, D., Schweim, D., and Rothlauf, F. (2023). A comprehensive survey on program synthesis with evolutionary algorithms. *IEEE TEVC*, 27(1):82–97.

SpotBugs (2024). SpotBugs: Find bugs in java programs. https://spotbugs.github.io/.

Spotify (2024). fmt-maven-plugin: Opinionated maven plugin that formats your java code. https://github.com/spotify/fmt-maven-plugin.

Starkweather, T., McDaniel, S., Mathias, K., Whitley, D., and Whitley, C. (1991). A comparison of genetic sequencing operators. In *ICGA*, pages 69–76.

Syswerda, G. (1991). Schedule optimization using genetic algorithms. In *Handbook of Genetic Algorithms*. Van Nostrand Reinhold.

Szu, H. and Hartley, R. (1987). Nonconvex optimization by fast simulated annealing. *Proceedings of the IEEE*, 75(11):1538–1540.

Tarkowski, T. (2023). Quilë: C++ genetic algorithms scientific library. *JOSS*, 8(82):4902.

Tayarani-N., M.-H., Yao, X., and Xu, H. (2015). Metaheuristic algorithms in car engine design: A literature survey. *IEEE TEVC*, 19(5):609–629.

Valenzuela, C. L. (2001). A study of permutation operators for minimum span frequency assignment using an order based representation. *J Heuristics*, 7(1):5–21.

Štefankovič, D., Vempala, S., and Vigoda, E. (2009). Adaptive simulated annealing: A near-optimal connection between sampling and counting. *JACM*, 56(3):18:1–18:36.

Wan, Y., Ma, A., He, W., and Zhong, Y. (2023). Accurate multiobjective low-rank and sparse model for hyperspectral image denoising method. *IEEE TEVC*, 27(1):37–51.

Whitley, L. D., Starkweather, T., and Fuquay, D. (1989). Scheduling problems and traveling salesmen: The genetic edge recombination operator. In *ICGA*, pages 133–140.

Xue, B., Zhang, M., Browne, W. N., and Yao, X. (2016). A survey on evolutionary computation approaches to feature selection. *IEEE TEVC*, 20(4):606–626.

Yazdani, D., Cheng, R., Yazdani, D., Branke, J., Jin, Y., and Yao, X. (2021). A survey of evolutionary continuous dynamic optimization over two decades—part a. *IEEE TEVC*, 25(4):609–629.

Zhou, X., Qin, A. K., Gong, M., and Tan, K. C. (2021). A survey on evolutionary construction of deep neural networks. *IEEE TEVC*, 25(5):894–912.