# EvoSuite

**Test Generation:** EvoSuite begins by instrumenting the bytecode of the target class to track coverage and identify all branch goals. It then initializes a population of whole‐test‐suite candidates, where each individual represents a set of JUnit test methods. A genetic algorithm evolves these suites: crossover swaps entire test methods between parents, and mutation can add, remove, or alter statements within tests. Fitness is computed as the sum of normalized branch distances for every branch not yet covered—lower values indicate suites closer to covering branches—so the search optimizes for high structural coverage. To prevent uncontrolled growth, EvoSuite applies bloat control: it penalizes overly long tests and, after the search, removes any statements or methods that do not contribute to coverage. Hybrid strategies enhance this process: local search fine‐tunes parameter values within tests, and EvoSuite occasionally invokes dynamic symbolic execution to solve hard‐to‐cover branches.

**Assertion Preparation:** After generating tests that cover branches, EvoSuite synthesizes assertions to serve as oracles. For each test method, it collects potential observation points—return values, object states, or results of “observer” methods—then creates a pool of assertion candidates. Next, EvoSuite applies mutation testing: it generates mutants of the class under test (small injected faults) and runs each candidate assertion against both the original and each mutant. By building a detection matrix that records which assertions “kill” (fail on) which mutants, EvoSuite selects a minimal subset of assertions that together detect all covered mutants. This mutation‐based minimization ensures that only meaningful, nonredundant assertions remain, capturing deviations from intended behavior without overwhelming the developer with spurious checks.

**Remaining Steps & Output:** Throughout the process, EvoSuite continuously monitors resource and time budgets, terminating the evolutionary search once limits are reached. It then performs a final reduction pass to prune any test methods or statements that became redundant (i.e., no longer improve coverage). The assertion synthesis step follows, using the minimized detection matrix to yield concise oracles. Finally, EvoSuite outputs a JUnit‐compliant test suite composed of high‐coverage test methods paired with a carefully chosen set of assertions that both reflect current behavior and flag real faults. This end‐to‐end pipeline produces ready‐to‐run tests that can be directly integrated into developers’ build systems.

**~ Post search pruning:**

EvoSuite’s post‐search pruning is a **coverage‐preserving minimization** carried out in two phases:

1. **Test‐Method Elimination**
   * Iterate over each generated test method in the suite.
   * Temporarily remove the method and re‐execute the instrumented class to recompute total branch coverage.
   * **If coverage remains unchanged**, permanently discard that method; otherwise, keep it.
   * Repeat until no further whole test can be dropped without losing coverage.
2. **Intra‐Test Statement Pruning**
   * For each remaining test method, walk through its statements (e.g., setup calls, method invocations, assertions).
   * Temporarily delete a statement and rerun coverage.
   * **If coverage is not impacted**, permanently remove that statement; otherwise, restore it.
   * Continue until every statement in every test is “necessary” for maintaining the same coverage.

By greedily deleting any code whose removal doesn’t degrade branch coverage, EvoSuite ensures the final suite is as small as possible while preserving the exact coverage achieved during the search.

**~ Questions:**   
How is such a pruning scalable? in big software using such a method of deleting each test and seeing the coverage again and again will cost too much

Pruning in EvoSuite operates at the granularity of **individual classes**, with each generated suite typically comprising only a few dozen test methods. Because:

1. **Test suites are class‐local and small**
   * EvoSuite produces tests for one Java class at a time. Even though it evolves whole suites, the number of test methods per class rarely exceeds a few dozen (thanks to bloat control).
   * Each test method is usually only a handful of statements invoking methods on that class, so executing it to measure coverage is very fast—often in milliseconds.
2. **Coverage measurement is lightweight**
   * EvoSuite instruments bytecode so that each branch maps to a simple counter or bitmap. Checking whether a removed test still covers a branch means merely re‐running that small set of methods under instrumentation and recomputing which branch bits flip.
   * Because coverage data is collected at runtime in a single pass (via in‐memory counters), recomputing coverage for “suite minus one test” does not involve heavyweight analysis—it’s just running the remaining tests and aggregating their bitmaps.
3. **Greedy, one‐pass strategy with early exit**
   * EvoSuite’s post‐search pruning uses a **greedy** algorithm: for each test method, it temporarily drops it and re‐runs coverage; if no branch coverage is lost, the method is discarded permanently. It then moves on to the next method.
   * As soon as dropping a method causes the coverage bitmap to change (i.e., a previously covered branch is no longer hit), the method is retained and pruning proceeds to the next candidate. This early‐exit check avoids unnecessary full‐suite reruns once any difference is detected.
   * Once no further whole‐method removals are possible, EvoSuite repeats the same process **within each remaining test**, deleting one statement at a time. Since most statements in a fully pruned suite either immediately affect coverage or are quickly identified as irrelevant, many deletions terminate after a handful of branch checks.
4. **Per‐class budget and parallelization**
   * EvoSuite enforces a fixed **time/resource budget per class**. Once the search budget is exhausted, pruning begins immediately and stops as soon as all candidates are evaluated—there’s no open‐ended loop.
   * In practice, EvoSuite can run pruning in parallel across multiple classes or even across multiple JVMs, since each class’s pruning is independent. This parallelism amortizes the cost when targeting large codebases.

Because of these factors—**small suite sizes**, **fast instrumentation‐based coverage checks**, and a **greedy, early‐exit removal procedure**—the post‐search pruning step remains tractable even for sizeable projects. In other words, EvoSuite trades repeated quick coverage‐only test executions (milliseconds each) against the cost of maintaining any single test, rather than re-searching or doing heavy symbolic analysis for each removal. This ensures that, although pruning does re-run coverage after each candidate deletion, the overall runtime remains proportional to the number of tests/statements (not cubic or exponential) and is acceptable in large codebases.

**~ DynaMOSA**

DynaMOSA (Dynamic Many‐Objective Sorting Algorithm) is the core search engine EvoSuite uses to drive test‐suite evolution toward high branch coverage. Below is an expanded explanation of how DynaMOSA works and why it’s critical for efficient, scalable test generation.

1. Background: Why Many Objectives?

* Each branch = an individual objective  
  In branch‐coverage testing, hitting each distinct branch in the code is a separate goal. A naïve approach (one objective at a time) wastes effort: once a test covers one branch, evolving further tests might inadvertently lose that coverage while trying to reach a new branch. MOSA treats *all* branches as simultaneous objectives, guiding the search toward covering as many as possible without sacrificing already‐covered ones.
* Pareto‐based selection  
  Standard MOSA uses Pareto nondomination: a test suite that covers more branches (or gets closer, via “branch distance,” on uncovered ones) dominates another. A “front” of nondominated candidates is preserved generation after generation. However, when there are *hundreds* of branches, classic MOSA can be overwhelmed by too many objectives, making selection and sorting expensive and often diluting search pressure.

2. Dynamic Objective Selection

* Key insight: Not all uncovered branches are equally “reachable” at any given moment. Some branches lie deep behind complex conditions; others are just one or two method calls away.
* Fitness distance heuristic: For each uncovered branch, EvoSuite computes a *branch distance*—a numeric measure of how “close” a test suite’s execution came to satisfying that branch’s condition. Smaller distance means “easier” to flip.
* Dynamic prioritization: DynaMOSA selects only those branches whose branch distance is minimal (i.e., currently closest to being covered) and treats only those as active objectives in the next generation. As tests begin to cover those “nearby” branches, DynaMOSA updates the set of active objectives to include the next‐closest branches.
  + This avoids wasting search effort on objectives whose branch distance is plateaued or unreachable from the current population.
  + By dynamically focusing on a smaller subset of “reachable” targets, the algorithm converges faster.

3. Fitness Assignment & Pareto Sorting

1. Compute Branch Distances
   * For each test suite and each currently active branch objective, DynaMOSA records the normalized branch distance.
   * If a suite already covers a branch, that objective’s distance is zero.
2. Objective Vector Construction
   * Each candidate test suite is represented by a vector of distances to all *active* objectives.
   * For example, if there are three active branches, a suite’s fitness vector might be (0.0, 0.5, 0.2) meaning it covers the first branch, is halfway to the second, and close on the third.
3. Pareto Triage
   * Candidates are sorted into Pareto “fronts”: suites that are not dominated by any other in terms of this multi‐dimensional distance vector.
   * Within each front, DynaMOSA uses a crowding distance metric to maintain diversity favoring suites that explore different combinations of branch distances.
4. Selection & Variation
   * EvoSuite picks survivors from the top Pareto fronts (plus some diversity from less‐fit fronts) to form the next generation’s parents.
   * Crossover and mutation operators then produce offspring.

By limiting the vector to only *active* objectives (often just a handful at a time), Pareto sorting is efficient and keeps search pressure high on branches that can be covered soon.

4. Lifecycle of Active Objectives

* Initialization: All uncovered branches whose distance is *finite* (i.e., can be reached by executing some input path) are potential candidates.
* Iteration:
  1. Identify the minimum branch distance across all current suites for each uncovered branch.
  2. Pick the k branches with the lowest distances (where k is a small, tunable parameter—often 3–5).
  3. Set these as the active objectives for the next evolutionary cycle.
* Branch Coverage: When a suite finally covers one of these active branches (distance drops to zero), DynaMOSA removes it from the “uncovered” pool and possibly brings in another branch whose branch distance is now the next smallest.
* Termination: Once all branches are covered or the search budget is exhausted, no new objectives remain. Any remaining uncovered branches are deemed too hard, and EvoSuite moves on to pruning and assertion synthesis.

5. Benefits of DynaMOSA

* Scalability: By dynamically restricting the number of objectives, DynaMOSA avoids the combinatorial explosion of sorting against hundreds of branch distances.
* Focused Search: The algorithm always prioritizes those branches that are “reachable next,” preventing wasted effort on unrealistic coverage goals until prerequisites are met.
* Diversity Maintenance: Pareto sorting with crowding distance ensures that multiple coverage paths are explored in parallel, reducing the chance of getting stuck in local optima.

In practice, this means EvoSuite spends most of its time exploring paths that it has a realistic chance of hitting, only gradually expanding the frontier to deeper or more complex branches. The result is higher coverage in less time compared to static many‐objective approaches.