Sketch4J: Execution-Driven Sketching for Java

Jinru Hua and Sarfraz Khurshid
The University of Texas at Austin
lisahua,khurshid@utexas.edu

ABSTRACT
Sketching is a relatively recent approach to program synthesis, which has shown much promise. The key idea in sketching is to allow users to write partial programs that have “holes” and provide test harnesses or reference implementations, and let synthesis tools create program fragments that fill the holes such that the resulting complete program has the desired functionality. Traditional solutions to the sketching problem perform a translation to SAT and employ CEGIS. While effective for a range of programs, when applied to real applications, such translation-based approaches have a key limitation: they require either translating all relevant libraries that are invoked directly or indirectly by the given sketch – which can lead to impractical SAT problems – or creating models of those libraries – which can require much manual effort.

This paper introduces execution-driven sketching, a novel approach for synthesis of Java programs using a backtracking search that is commonly employed in software model checkers. The key novelty of our work is to introduce effective pruning strategies to efficiently explore the actual program behaviors in presence of libraries and to provide a practical solution to sketching small parts of real-world applications, which may use complex constructs of modern languages, such as reflection or native calls. Our tool Sketch4J embodies our approach in two forms: a stateful search based on the Java PathFinder model checker; and a stateless search based on re-execution inspired by the VeriSoft model checker. Experimental results show that Sketch4J’s performance compares well with the well-known SAT-based Sketch system for a range of small but complex programs, and moreover, that Sketch4J can complete some sketches that require handling complex constructs.

1 INTRODUCTION
Program sketching [23] is an approach to program synthesis [1, 6, 17, 24], which has led to exciting advances in the application space for synthesis. It allows users to write partial programs that have “holes” and let synthesis tools fill the holes such that the completed program satisfies given test harnesses or reference implementations. Existing sketching approaches [21] translate the partial program to propositional satisfiability formulas and leverage counter-example-guided inductive synthesis (CEGIS) to generate program with desired functionality based on off-the-shelf solvers.

While these translation-based approaches have shown their effectiveness on a range of programs, they have a key limitation: when applying to real applications with complex libraries, these translation-based approaches require either translating all libraries that are invoked directly or indirectly by the given sketch or creating models of those libraries, which can lead to impractical SAT problems.

This paper introduces Sketch4J, a novel approach that performs execution-driven sketching for synthesizing Java programs using a backtracking depth-first search that is commonly employed in software model checkers [9]. The key novelty of our work is to introduce effective pruning strategies to reduce the search space for possible solutions and efficiently explore the actual program behaviors in presence of libraries. Our approach provides a practical solution to sketching small parts of real-world applications, which may use complex constructs of imperative languages, such as reflection or native calls.

As inputs, Sketch4J takes a sketch (partial program) with holes written using Java syntax, and a test suite that characterizes the correctness specification. Sketch4J basically supports three kinds of holes: boolean conditions (e.g., for a while loop), expressions (e.g., field dereferencing), and blocks of assignment statements. Sketch4J instruments the given sketch to introduce non-determinism in the program, which allows a backtracking search to explore the space of candidate programs. Sketch4J executes the test suite against the instrumented program and backtracks the search when it encounters a failure (runtime failure or test assertion failure). Sketch4J terminates when the space of candidate programs is exhausted or a complete program that satisfies all test cases is found.

To initialize the search, Sketch4J generates candidate expressions for the expression holes based on the target type given by the user. The candidates are computed based on the visible variables or variables provided by the user. For instance, using up to two field dereferences, the expressions of the type Entry derived from a variable e that represents an entry in a singly linked list should be {e, e.next, e.next.next}, where the field next represents the next entry in the linked list.

Sketch4J introduces two key pruning strategies to optimize sketch completion. These strategies determine whether some candidates cannot lead to a correct solution and prune them. For stitching assignment statement blocks, we define a set of pruning rules based on Java semantics. For condition sketching, we introduce a value grouping strategy that splits all condition candidates into two sets based on their values (true and false) at the current point in execution. The condition candidates are generated by combining expression candidates with condition operators. We define two condition operators and = for non-primitive expressions, and six operators [==, !=, >, <, >=, <=] for primitive types. With the value grouping strategy, Sketch4J only considers two choices when it evaluates a condition, i.e., all condition candidates evaluated to be true will be regarded as a single candidate. A set of condition candidates may be split into smaller sets, e.g., when the
same condition is evaluated before executing the next iteration of the loop.

Our tool SketchJJ embodies our approach in two forms: SketchJJ-
JPF, a stateful search based on the Java PathFinder model checker [25];
and SketchJJ-JVM, a stateless search based on re-execution in-
spired by the VeriSoft model checker [9]. We evaluate SketchJJ us-
ing a two-fold controlled experiment. One, we compare SketchJJ’s performance
with the SAT-based Sketch system [23]. We create a
dataset of small yet complex data structures and execute SketchJJ
on these subjects with bounded exhaustive test cases generated by
Korat [3]. We manually transform these subjects and test suites to
the Sketch language, which is the input language for the Sketch
system. Two, we demonstrate how SketchJJ handles sketching
larger applications that use complex operations.

Experimental results show that SketchJJ’s performance com-
pares well with the Sketch system. In particular, out of 43 sketching
tasks, SketchJJ outperforms the Sketch system on 40 tasks. The
experiments also show that our pruning strategies are able to prune
an average of 35% of candidates before evaluating them against the
tests. Moreover, SketchJJ completes some sketches that require
handling reflection, I/O, native calls, and external libraries.

This paper makes the following contributions: 1) Execution-
driven pruning. We present a practical approach for program
sketching using tests based on execution-driven pruning; 2) Prun-
ing strategies. We introduce pruning strategies that reduce the
choices to explore for candidate expressions, conditions, and state-
ments and enhance the efficiency of our approach; 3) Embodi-
ment. We embody SketchJJ into two prototypes: one based on
the stateful model checker JPF [25], and the other based on a dedi-
cated stateless search using re-execution in the spirit of the VeriSoft
model checker [9]; and 4) Evaluation. We present an experimental
evaluation that compares SketchJJ with the SAT-based Sketch syn-
thesizer and also demonstrates that SketchJJ can handle complex
con structs.

2 MOTIVATING EXAMPLE

This section presents a motivating example to illustrate sketching
a small but intricate method using our approach and highlights
some of its key steps. Assume that the user wants to implement
a reverse() method for the singly linked list as shown as Figure 1
(A). Each list has a head entry (which can be null), and each entry
has a next entry and a value integer as fields. The user wants to
synthesize an implementation that uses three local variables with
the type Entry and a while loop to traverse the list. The users
leaves the condition of the while loop and as well as its body to be
synthesized by the sketching system.

More specifically, the user writes the sketch shown in Figure 1 (A),
which requires synthesis of the while condition at line 8 and a block
of assignments at line 10 with a parameter of the target type. To
specify the correctness criteria for the synthesized program, the
user writes four test cases that have between zero to three entries.
Figure 1 (B) shows one example JUnit test with three entries. Even if
we only consider candidates with one field dereference, i.e., expres-
sions like ln1.next, the space of possible complete programs is very
large; for this sketch the space has over 4.4 billion candidates with
up to 4 statements – the minimum number of statements required
for this sketch – in the while loop body.

<table>
<thead>
<tr>
<th>(A) A program sketch written by users</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. class LinkedList</td>
</tr>
<tr>
<td>2. Entry head;</td>
</tr>
<tr>
<td>3. public void reverse() {</td>
</tr>
<tr>
<td>4. if (head==null) return;</td>
</tr>
<tr>
<td>5. Entry ln1 = head;</td>
</tr>
<tr>
<td>6. Entry ln2 = null;</td>
</tr>
<tr>
<td>7. Entry ln3 = null;</td>
</tr>
<tr>
<td>8. boolean cond = SketchJJ.JPF(Entry.class);</td>
</tr>
<tr>
<td>9. while (cond) {</td>
</tr>
<tr>
<td>10. SketchJJ.JPF(Entry.class);</td>
</tr>
<tr>
<td>11. }</td>
</tr>
<tr>
<td>12. class Entry</td>
</tr>
<tr>
<td>13. Entry next;</td>
</tr>
<tr>
<td>14. int value;</td>
</tr>
<tr>
<td>15. }</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>(B) A JUnit test case provided by users</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. public void testThreeEntries() {</td>
</tr>
<tr>
<td>2. LinkedList list = new LinkedList(new int[]{1,2,3});</td>
</tr>
<tr>
<td>3. list.reverse();</td>
</tr>
<tr>
<td>4. assertEquals(&quot;[1,2,3]&quot;, list.toString());</td>
</tr>
<tr>
<td>5. }</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>(C) A solution generated by SketchJJ</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. SketchJJ.JPF(Entry.class);</td>
</tr>
<tr>
<td>2. SketchJJ.JPF(Entry.class);</td>
</tr>
<tr>
<td>3. head = ln1;</td>
</tr>
<tr>
<td>4. ln1 = head.next;</td>
</tr>
<tr>
<td>5. ln2 = head;</td>
</tr>
</tbody>
</table>

Figure 1: A sketch example for singly linked list reversal

SketchJJ dynamically selects candidates for the sketch invoca-
tions (line 8 and line 10) when it executes given test cases. When
SketchJJ first reaches the while condition sketch, SketchJJ groups
all condition candidates based on their evaluated value (true and
false), and non-deterministically considers two boolean possibilities
for the execution. When the value false is considered, the invo-
cation of the assignment sketch is ignored and does not created
additional search space because the body of the while loop will not
be executed.

When SketchJJ first reaches the assignment sketch at line 10, it
non-deterministically selects expressions that form the right-hand-
side and the left-hand-side of the assignment statement. A simple
search may explore many candidates that are subsumed by other
candidates which are already being explored. SketchJJ prunes
a number of such candidates based on the Java semantics (Section 3.3). SketchJJ backtracks its search if the current sequence of choices
fails due to a runtime error or a test failure. The user may specify
the number of statements in the while block to sketch; by default,
SketchJJ incrementally adds one assignment at a time until it finds
the first solution or reaches the preset upper bound on the number
of assignments, which we have currently set as 5.

Figure 1 (C) presents a solution generated by SketchJJ. This
solution does not use the variable ln3, but another solution with
five statements does use ln3. In this example, SketchJJ finds the
first solution in 9 seconds after exploring over 490K candidates.

3 APPROACH

We describe our execution-driven sketching in this section. As
shown in Figure 2, SketchJJ first constructs all candidates at the
expression level based on the program sketch $P$ and the default bound of field dereference $b$ (Section 3.1). Sketch4J then instruments the program so that it can dynamically select candidates for the "holes" (Section 3.2). Section 3.3 describes the pruning strategies we apply to sketch assignments and conditions when evaluating candidates via test execution ($T$). And Section 3.4 discusses our backtrack engines for stateful prototype (Sketch4J-JPF) and stateless prototype (Sketch4J-JVM).

### 3.1 Expression Candidate Generation

This section describes how we generate expression candidates. Sketch4J leverages a breadth-first iteration to generate field dereferences for all provided variables within a pre-defined bound of $b$. If no candidate variables are provided by users, Sketch4J will use all heap-allocated variables to generate expression candidates. When the number of field dereferences is within the given bound, Sketch4J creates all field dereferences using reflection and iteratively adds generated field dereferences to the list of candidates. After generating all possible candidates, Sketch4J selects expression candidates based on the target type of the "hole". We set the default bound of field dereference as one and make it configurable via a list of overloading invocations of Sketch4J.

For the primitive types int, double, and their corresponding wrapper types such as Integer and Boolean, null for non-primitive types, and an empty string for the type String. We make these default values as configuration options. Specifically, the object this is also regarded as a variable yet we do not generate its field references because these fields are already included as heap-allocated variables. The implicit length field for the array type is not reflected by the getFields() method [12] thus we manually insert this field if there exist candidates with the array type.

For each non-deterministic "hole", we put all its expression candidates in a vector called candidate vector. We assign each expression candidate a unique identifier, which is its index in the candidate vector. When Sketch4J performs sketching, it dynamically selects a candidate identifier for each "hole" using non-deterministic choose() operator and executes the program based on the candidate it selects. All candidate identifiers for sketch "holes" are initialized as -1, indicating that Sketch4J has not selected a candidate for this "hole". Once Sketch4J selects a candidate identifier for this "hole", this candidate will be used consistently across all test cases.

Current prototypes of Sketch4J can sketch assignments, conditions, and expressions involving field access, variables, and infix expressions such as $a + b$. For infix expressions, we support five operators: $[+, -, \times, /, \%]$. Sketch4J transforms an infix sketch to two expression sketches at left-hand-side and right-hand-side, and creates another "hole" for infix operators. Similar to expression candidates, we assign a unique identifier for each infix operator based on their index in the operator vector. For instance, the index for the operator "+" is 0 and the index for "\times" is 2.

### 3.2 Program Instrumentation

To introduce non-determinism in the program and allow a backtracking search to explore the space of candidate programs, we
To distinguish different sketches, we assign a unique identifier for each sketch and append it to the parameter list of the invocations. Starting from zero, \textsc{Sketch4j} assigns a unique identifier for each type of sketch. In our motivating example, \textsc{Sketch4j} assigns an identifier 0 to the first while condition sketch at line 12 in Figure 3 and assigns an identifier 0 to the first assignment sketch at line 14. For the while condition sketch, we reassign the condition with the same invocation at the end of this loop (line 15), because the loop body may have changed the condition value.

We next described how we instrument assignment sketches. \textsc{Sketch4j} first collects all visible local variables for the assignment sketch, and creates new fields to represent these variables. If these variables are used in the later part of the method after \textsc{Sketch4j} invocation, the use of these local variables is replaced with corresponding fields. For each invocation of \textsc{Sketch4j}.\_BLOCK, \textsc{Sketch4j} generates a new method \texttt{_{BLOCK}}, and passes the return value of \textsc{Sketch4j}.\_BLOCK as its parameter. \textsc{Sketch4j} generates a for loop in the \texttt{_{BLOCK}} method to assign values to the left-hand-side expressions for each assignment in the sketch block. This process is also defined as the function getBlock in Algorithm 1. Inside the for loop, \textsc{Sketch4j} creates a switch statement to select the left-hand-side expression based on the candidate identifier for the "hole", and assigns right-hand-side expression to the selected left-hand-side expression. \textsc{Sketch4j} will only generate case statements for variables and field accesses that can be assigned, but will not generate case statements for this object, unmodifiable fields like array.length, and default candidates such as \texttt{null}, \texttt{-1}, or \texttt{0}.

If the expression candidates are the same for two invocations of \textsc{Sketch4j}.\_BLOCK, we reuse the same \texttt{_{BLOCK}} method for both invocations. However, if the expression candidates are different, \textsc{Sketch4j} will generate two \texttt{_{BLOCK}} methods (\texttt{_{BLOCK}} and \texttt{_{BLOCK2}}) to handle "holes" with different candidates.

### 3.3 Pruning Strategies

In this section, we discuss our pruning strategies for sketching assignments and conditions.

#### 3.3.1 Assignment Pruning Rules

We define four pruning rules based on the Java syntax and simple program isomorphism analysis. These rules may prune the program based on one assignment (rule 1) or two consecutive assignments in the sketching block (rule 2-4).

For the rules below, we use \texttt{e1} to represent an expression of field access, variable, or infix expression, and use \texttt{v1}, \texttt{v2}, \texttt{v3} to represent variables. The method \texttt{id()} returns the candidate identifier of this candidate.

1. If the left-hand-side expression is equal to the right-hand-side expression, the candidate is ignored as the assignment has no effect on the current program state. E.g., any candidates

```
function getBlock(exprList, trueSet, falseSet)
  if first access
    /* Initialize condition candidates */
    condCands ← ∅;
    if is integer, float, or double then
      condCands ← construct(exprList, primOp);
    else
      condCands ← construct(exprList, [==, !=]);
    foreach c ∈ condCands do
      if eval(c) then
        trueSet ← trueSet ∪ c;
      else
        falseSet ← falseSet ∪ c;
    else if select=0 then
      /* Split trueSet */
      falseSet ← ∅;
      foreach c ∈ trueSet do
        if eval(c)==false then
          falseSet ← falseSet ∪ c;
        trueSet ← trueSet \ c;
      else
        trueSet ← ∅;
        foreach c ∈ falseSet do
          if eval(c)==true then
            trueSet ← trueSet ∪ c;
          falseSet ← falseSet \ c;
        if trueSet is empty then
          select = 1;
        else if falseSet is empty then
          select = 0;
        else
          select = choose(0, 1);
      return (select, trueSet, falseSet);
```
We consider the consecutive assignments (4)
and make this value configurable.

To get rid of infinite loops, we set the default bound for while loops as 10
and make this value configurable.

that have the assignment \( \text{ln1.next} = \text{ln1.next} \) will be pruned in the
motivating example shown as Figure 1.

(2) \( v_1 = v_2, v_2 = v_1 \). If the left-hand-side variable of the first as-
signment is the same as the right-hand-side variable of the second assignment,
and the right-hand-side variable of the first assignment is the same as the left-hand-side variable of the second assignment,
this candidate is ignored because the two assignments are equiva-
lent to one assignment \( v_1 = v_2 \). E.g., in Figure 1, the candidate will be
pruned if it has two consecutive statements \( \text{ln1} = \text{ln2} \) and \( \text{ln2} = \text{ln1} \).

(3) \( v_1 = v_2, v_2 = v_3 \). If both left-hand-side variables in two con-
secutive assignments are the same and both right-hand-sides are
variables, we do not need to execute this program because the second
assignment overrides the first assignment. For instance, the candidate in
Figure 1 with consecutive assignments \( \text{ln2} = \text{ln4} \) and \( \text{ln2} = \text{ln3} \) will be ignored,
because the two assignments are equivalent to a single assignment \( \text{ln2} = \text{ln3} \), which has been evaluated by
Sketch4J.

(4) \( v_3 = v_1, v_2 = v_1 \) while \( id(v_3) > id(v_2) \). If two consecutive
assignments have the same variable at the right-hand-side, we only
execute the program if the identifier of the first assignment’s left-
hand-side variable is smaller than that of the second assignment.
We consider the consecutive assignments \( v_2 = v_1, v_3 = v_1 \) and
\( v_3 = v_1, v_2 = v_1 \) as isomorphic solutions and we only execute
isomorphic solutions once with given test suite.

Sketch4J will backtrack if there is a test failure or the program
throws an unexpected exception like NullPointerException. To get
rid of infinite loops, we set the default bound for while loops as 10
and make this value configurable.

3.3.2 Condition Pruning Strategy. To sketch condition expres-
sions including if condition and while loop condition, Sketch4J
first generates all condition candidates and splits these candidates
into two groups based on their evaluated values (true and false).
Algorithm 2 presents the pseudocode of how Sketch4J sketches a
condition expression.

Condition Candidate Generation. During the first access of the
condition sketch, Sketch4J generates all condition candidates by
combining expression candidates with condition operators. We
define two condition operators \{==, ! =\} for non-primitive types
and six condition operators for primitive types \{==, ! =, >, <, >=, <=\}. The primitive type
operators are also applied to corresponding wrapper classes such as
Integer. We only need to consider the combination \( e_1 \ op e_2 \) where \( id(e_1) < id(e_2) \) based on the program
symmetry. Assume that \( e_1 \) and \( e_2 \) are two non-primitive expres-
sion candidates, and the \( e_1 \)'s candidate identifier is smaller than
\( e_2 \), we only need to consider the condition candidates \( e_1 == e_2 \)
and \( e_1! = e_2 \) because \( e_1 == e_2 \) and \( e_1! = e_2 \) are equivalent to
the previous two candidates. For instance, assume that we have five
expression candidates with the primitive type int, Sketch4J will
generate \( 60 \times (6 + 3 + 2 + 1) \) condition candidates by combining
each candidate with the ones that have bigger identifiers using six
condition operators. We also include two constant boolean value
true and false for completeness (\( e_1 == e_1 \) and \( e_1! = e_1 \)).

Condition Value Grouping. The generated condition candidates
are further split to two sets based on their evaluated values, shown
as line 9 to 13 in Algorithm 2. If it is not the first access, Sketch4J
will re-evaluate each candidate in the set and split the candidate set
based on the evaluated value of each condition candidate in the new

3.3.2 Condition Pruning Strategy. To sketch condition expres-
sions including if condition and while loop condition, Sketch4J
first generates all condition candidates and splits these candidates
into two groups based on their evaluated values (true and false).
Algorithm 2 presents the pseudocode of how Sketch4J sketches a
condition expression.

**Condition Candidate Generation.** During the first access of the
condition sketch, Sketch4J generates all condition candidates by
combining expression candidates with condition operators. We
define two condition operators \{==, ! =\} for non-primitive types
and six condition operators for primitive types \{==, ! =, >, <, >=, <=\}. The primitive type
operators are also applied to corresponding wrapper classes such as
**Condition Value Grouping.** The generated condition candidates
are further split to two sets based on their evaluated values, shown
as line 9 to 13 in Algorithm 2. If it is not the first access, Sketch4J
will re-evaluate each candidate in the set and split the candidate set
based on the evaluated value of each condition candidate in the new

**Figure 4: Sketch4J pruning example**
execution. If Sketch4J selects boolean value true in the previous execution, the trueSet will be split in this execution. For example, the condition candidate ln1 != null in the motivating example may be true in an iteration, and its value may change to false in the next iteration. Therefore, the condition candidate ln1 != null will be put in trueSet in the first iteration, and will be moved to falseSet in the next iteration.

Sketch4J chooses a boolean value at the end of each invocation based on the size of two candidate sets, shown as line 25 to 32 in Algorithm 2. If there is no candidate that is evaluated to be true, Sketch4J will select false represented as 1 at line 27 in Algorithm 2. And if the set of candidates which are evaluated to be false is empty, Sketch4J will select true represented as 0. The selected boolean value is returned from the getCondition method together with two candidate sets. If the chosen value does not satisfy test assertions, Sketch4J will backtrack to the previous choice, and select a different value based on the non-deterministic choose() operator.

Figure 4 illustrates the exploration space with and without value grouping strategy using a small example program with a while-loop that has in its body an if statement. Assume that the while condition has eight candidates (wc ∈ {w1, w2, ..., w8}) and the if condition also has eight candidates (ic ∈ {i1, i2, ..., i8}). We illustrate three cases that show the differences in the size of the exploration space in terms of the number of transitions.

We consider three cases and report the number of non-deterministic transitions with and without value group strategy. In the first case, we assume that there is only one candidate for which the condition evaluates to true for the while loop (wc) and false for the if condition (ic) when it is executed the first time and hence no further choices will be created in the state-space during future executions of the same condition; we term this case as the “best-case” shown as Figure 4 (A). The “best case” requires 4 transitions with value grouping and 16 transitions without value grouping.

Similarly, we assume that there is only one candidate for which the condition evaluates to false for the while loop (wc) and true for the if condition (ic) when it is executed the first time. Sketch4J will keep on create non-deterministic choices until it exhausts all choices; we term this case as the “worst case”. Shown as Figure 4 (B), Sketch4J has 28 transitions with value grouping strategy, while the traditional approach without value grouping requires 64 transitions. Lastly, we consider a case where candidates are equally split in each transition. Shown as Figure 4 (C), Sketch4J with value grouping strategy requires 12 transitions while traditional approach requires 40 transitions. Our example illustrates that our value grouping strategy effectively reduces the number of transitions in three cases.

3.4 Backtrack Execution

We build two prototypes based on two different backtrack engines: a stateful prototype based on Java PathFinder [25] and a stateless prototype based on re-execution.

3.4.1 Stateful prototype with Java PathFinder. Java PathFinder (JPF) [25] is a mature model checker that implements a customized JVM. JPF is a general purpose stateful model checker that provides all the common operations supported by modern software model checkers, including checking of multi-threaded programs. It provides a default depth-first search, which efficiently stores and restores program states. It is straightforward to use JPF as a backend to implement a backtracking search. While JPF handles all of Java bytecode, it only handles a limited number of native calls, which limits its applicability for synthesis in the context of real-world applications.

3.4.2 Stateless prototype using re-execution. Our second prototype is based on a dedicated stateless [9] search using re-execution [16]. Since this prototype executes on the standard JVM, it can be used for synthesis in the context of open-source projects with advanced features, such as reflection, I/O, and native calls.

4 EVALUATION

We evaluate Sketch4J on a benchmark of small but complex data structures and illustrate Sketch4J’s ability to sketch real-world Java code with advanced features.

We address the following research questions in the evaluation:

• How effective is Sketch4J to sketch small but complex subjects compared to the SAT-based synthesizer?
• How do the pruning strategies affect the search space of sketching?
• Can Sketch4J sketch real-world Java programs with advanced language features?

4.1 Evaluation Subjects

To study Sketch4J’s efficacy of sketching small but complex data structures, we select ten subjects from java.util source code and algorithm book [4]. As shown in Table 1, the subjects are: Binary Search Tree Insertion (BSTAS and BSTCD), Finding Median (MEDAS and MEDCD), Red-Black Tree Insertion (RBSTAS and RBSTCD), Singly Linked List Reversal (LLREV), Doubly Linked List Add First (DLYLAF), Doubly Linked List Add Last (DLRLAL), and Red-Black Tree Removal (RBTTRM). The dataset is presented at [22] for cross-validation. We evaluate the performance and pruning strategies of sketching assignments and conditions separately because we apply different pruning strategies to these two types of sketching.

To reach full branch coverage, we use seven test cases from [15] for Find Median subjects, and we use Korat [3] to generate bounded exhaustive test suites for the rest subjects. Korat is a test generation tool that uses given constraints to guide the generation of bounded suites. We use bounded exhaustive test suite up to three nodes for Binary Search Tree, Singly Linked List, and Doubly Linked List, and test suite up to four nodes for Red-Black Tree. We sort the test cases based on the number of nodes and execute Sketch4J with test cases in ascending order.

Table 1 lists the search space for the first five assignments or conditions. The subjects can have more than five statements (RBTCDD has seven statements) or only four statements, whose search space of the fifth statement is marked as N/A. For example, to sketch one assignment for the findMedian method with three numbers, there are four candidates at the left-hand-side of the assignment (three numbers and one temporary variable), and the right-hand-side expression has four candidates as well. Thus there are 16 possibilities for one assignment. The search space for two assignments will be the product of two assignments’ search space. Similarly, the search space size for sketching a condition in the findMedian method will
Table 1: Evaluation subjects

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Tests</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>BSTAS</td>
<td>8</td>
<td>196</td>
<td>38.4 K</td>
<td>7.5 M</td>
<td>1.5 B</td>
<td>289 B</td>
</tr>
<tr>
<td>A</td>
<td>MEDAS</td>
<td>7</td>
<td>16</td>
<td>256</td>
<td>4.1 K</td>
<td>66 K</td>
<td>N/A</td>
</tr>
<tr>
<td>A</td>
<td>LLREV</td>
<td>4</td>
<td>64</td>
<td>4.1 K</td>
<td>262.4 M</td>
<td>16.7 M</td>
<td>1.1 B</td>
</tr>
<tr>
<td>A</td>
<td>RBTAS</td>
<td>15</td>
<td>676</td>
<td>457 K</td>
<td>309 M</td>
<td>209 B</td>
<td>N/A</td>
</tr>
<tr>
<td>A</td>
<td>DLLAF</td>
<td>4</td>
<td>196</td>
<td>38 K</td>
<td>7.5 M</td>
<td>1.5 B</td>
<td>N/A</td>
</tr>
<tr>
<td>A</td>
<td>DLLAL</td>
<td>4</td>
<td>196</td>
<td>38 K</td>
<td>7.5 M</td>
<td>1.5 B</td>
<td>N/A</td>
</tr>
<tr>
<td>C</td>
<td>BSTCD</td>
<td>8</td>
<td>392</td>
<td>190.5 K</td>
<td>74.7 M</td>
<td>29.3 B</td>
<td>14.2 T</td>
</tr>
<tr>
<td>C</td>
<td>RBTRM</td>
<td>15</td>
<td>74</td>
<td>5.5 K</td>
<td>7.4 M</td>
<td>10.0 B</td>
<td>N/A</td>
</tr>
<tr>
<td>C</td>
<td>MEDCD</td>
<td>7</td>
<td>96</td>
<td>9.2 K</td>
<td>884.7 K</td>
<td>84.9 M</td>
<td>8.2 B</td>
</tr>
<tr>
<td>C</td>
<td>RBTCD</td>
<td>15</td>
<td>74</td>
<td>5.5 K</td>
<td>7.4 M</td>
<td>10.0 B</td>
<td>13.5 T</td>
</tr>
</tbody>
</table>

Column Type represents the sketching type: A represents assignments and C represents conditions. If the subject has only four statements, the search space for the fifth statement is marked as N/A.

be 96 considering six operators (==, !=, >, <, >=, <=) and four candidates at both sides of the condition. Note that not all candidates in the search space will be executed due to our pruning strategies.

4.2 Performance Comparison with Sketch Synthesizer

We compare the Sketch4J with Sketch synthesizer [23], a state-of-the-art SAT-based inductive synthesizer that has had successes on sketching code in small well-defined domains like data structures [23]. We manually transform the Java subjects and test suites to Sketch language, which is a type-based language similar to Java. Sketch language only contains a subset of the Java syntax which does not support overriding, reflection, and native call. We execute Sketch4J-JVM, Sketch4J-JPF, and Sketch synthesizer on the subjects using the same test suites and report the time when they find the first solution that satisfies all test cases.

All performance experiments are conducted on a MacBook Pro with 2.2 GHz Intel Core i7 processor and 16 GB of 1600 MHz DDR3 memory running OS X version 10.12.1.

Figure 5 represents the sketching performance time of Sketch4J-JVM, Sketch4J-JPF, and Sketch synthesizer for sketching different numbers of assignments. The x-axis shows the number of statements under sketching, and the y-axis represents the average performance time for sketching tasks with specific number of assignments. For instance, if we want to sketch three assignments in a four-assignment block for DLLAF subject, we may choose to sketch assignments (1,2,3), (1,3,4), (1,2,4), and (2,3,4). The y-axis for DLLAF with three statements represents the average performance time of these four sketching tasks. The y-axis for the last four subjects are transformed with log2 scale for better display. The green line with triangle represents the performance of Sketch4J-JVM, the red line with circle represents Sketch4J-JPF, and the blue line with square represents Sketch synthesizer. Figure 5 illustrates that Sketch4J is able to sketch small but complex data structures with a better performance compared to SAT-based inductive synthesizer in most sketch tasks. E.g., for the subject LLREV, Sketch4J-JVM sketches the first correct solution with five assignments in 3.1 seconds while Sketch synthesizer takes 11.7 seconds for the same task. Out of 22 experiments in 6 subjects, Sketch4J-JVM is faster than Sketch synthesizer in only 20 experiments except the subject DLLAF with four assignments (22.4 vs 1.9) and the subject DLLAL with four assignments (25.8 vs. 2.8). The average performance time for Sketch4J-JVM is 16.2 seconds while the Sketch synthesizer is 25.1 seconds.

Figure 6 presents the performance of three tools for sketching conditions, including if conditions and while conditions. For instance, for the subject RBTCD, Sketch4J-JVM sketches the first correct solution with six conditions in 1.6 seconds while Sketch synthesizer takes 15.2 seconds for the same task. Out of 22 experiments in 6 subjects, Sketch4J-JVM is faster than Sketch synthesizer in only 16 experiments except the subject DLLAF with four conditions (22.4 vs 1.9) and the subject DLLAL with four conditions (25.8 vs. 2.8). The average performance time for Sketch4J-JVM is 16.3 seconds while the Sketch synthesizer is 25.1 seconds.
spends 58.9 seconds. It might because the transformation of Red-Black Tree implementation to boolean formulas is not trivial and it takes a long time for the SAT solver to find a solution. Out of 21 experiments in four subjects, SketchJ-JVM is faster than Sketch synthesizer in 19 experiments except the experiment BSTCD with three and four conditions (1.7 vs. 0.8 and 10.5 vs. 1.7).

We perform the Wilcoxon test to measure if SketchJ-JVM is significantly faster than Sketch synthesizer and use Cliff’s delta effect size to measure how large this difference is. The result of $p < 0.01$ and effect size value $-0.57$ (large) indicates that SketchJ-JVM is significantly faster than SAT-based Sketch synthesizer on our sketching subjects.

Previous work [2] conjectured that backtracking solver purely based on concrete execution will be much slower than SAT-based solver in exploring large state space due to the highly optimized heuristics used by modern SAT engines. However, our experiments show that the efficacy of SketchJ with our pruning strategies for sketching assignments and conditions is comparable or even at times faster than the SAT-based synthesizer.

Comparison of two prototypes. In our experiments, stateful search using JPF is always slower than the dedicated stateless search. This is not surprising. JPF is a general purpose model checker that implements a custom JVM to handle all of Java bytecode, including multi-threaded programs. SketchJ-JPF shows how JPF provides a very convenient way to implement a solution for the sketching problem, albeit with sub-optimal performance. Note, however that using JPF as the backend for sketching opens the future work possibility for a powerful approach for sketching multi-threaded programs.

4.3 Efficacy of Pruning Strategies.

To evaluate if our pruning strategies can effectively reduce the number of candidates before executing given test suites, we report the number of executed programs with and without pruning rules for sketching assignment subjects. The x-axis represents the number of assignments and the y-axis represents the average number of executed programs for sketching a certain number of assignments. The last four subjects are transformed to square-foot scale for better display. The black bars represent the number of executed programs with pruning rules and the grey ones represent that without pruning rules. As shown in Figure 7, our pruning rules can effectively prune 7% to 70% with an average of 21% candidates before executing the test suite. For instance, the pruning rules discard 68.4% of candidates for sketching four assignments in the subject LLREV, that is, only around one third of the candidates are actually executed with the test suite and more than two third of them are pruned before being executed. The subjects BSTAS and MEDAS have a lower pruning rate (the percentage of candidates without execution out of all executed candidates) compared to other subjects, because the sketched assignments in these two subjects are scattered in different if-else conditions and only the rule 1 of the pruning rules can apply to these two subjects.

Figure 8 presents the number of executed programs with and without pruning rules for sketching condition subjects. The y-axis for all subjects are transformed to log2 scale for better display. Our value grouping strategy can effectively prune an average of 56% of candidates before executing test suites. Note that for sketching one condition in the subject RBTRM, 35 programs are executed on average with value grouping strategy, while only 15 programs are evaluated without value grouping strategy. This result indicates that the value grouping strategy might not always bring in saving when the search space is small, which is different from the pruning rules for assignments.
At line 11, we try to sketch the object whose "left" field will be used for the if condition. We provide a bounded exhaustive test suite for up to 3 nodes and execute Sketch4J on the given test suite. Sketch4J completes this task with the variable expression cur.

Figure 9 (B) shows a getSum() method that reads two variables from a file and outputs their sum. We try to sketch the infix expression $a+b$ for the return statement, and Sketch4J completes this task with three test cases (0,0), (0,1), and (2,1). Figure 9 (C) presents a sketch task with native calls. Sketch4J sketches an incomplete swap() method for two integers and returns the string concatenation for the swapped integers in a native method nativeToString() using three test cases (0,0), (0,1), and (2,1).

Figure 9 (D) presents a sketching task derived from the open source project Apache commons-lang. The version we use has 234 files, 55 thousand lines of Java code, and more than 10 external libraries including maven plugins. Derived from a human-written commit of RandomStringUtilsTest class, the sketch task involves an if condition and an assignment with a state space of 96 possibilities. Within 149 test files from the original project, only the unit test file RandomStringUtilsTest reaches the random() method that we try to sketch. Each invocation of the random() method executes more than 20 methods along the execution trace using JUnit. We first execute Sketch4J on the unit test file RandomStringUtilsTest which has 10 test cases, and Sketch4J completes the sketch task as expected in 0.06 seconds. We then perform sketching based on the entire test suite from the open source project consisting of 3274 test cases. Sketch4J completes the sketch task in 8.8 seconds as expected.

4.5 Discussion
Following the standard behavior of synthesis approaches, we compare performance time to find the first solution with SAT-based synthesizer. To further investigate the efficacy of our pruning strategies, we execute Sketch4J to find all solutions that satisfy the test suite, and compare the pruning rate of all data structure subjects, i.e., the percentage of candidates without execution out of all executed candidates. The result shows that the pruning rate for the first solution is similar to that of all solutions (35% vs 37%, Wilcoxon test $p > 0.05$), i.e., the efficacy of our pruning strategies is consistent in finding the first solution and all solutions based on our subjects.

We further discuss two configuration options that might influence the performance of Sketch4J.

It is conjectured that the order of the test cases might influence the Sketch4J’s sketching efficacy. Therefore, we sort the bounded exhaustive test cases based on the number of nodes, execute Sketch4J with test cases in ascendent order and descendent order, and compare the performance time of finding the first solution for all subjects. The result illustrates that the performance is almost the same with test cases reordered (5.12 seconds vs. 5.7 seconds, Wilcoxon test $p > 0.05$). It might because the subjects we select are relatively small with a small number of test cases and the time to evaluate test cases is negligible, thus the prioritization of test cases has little influence on the total performance.

Lastly, we investigate the order of selecting left-hand-side and right-hand-side expressions for sketching assignments. We have two options to sketch an assignment: select left-hand-side expression first and then select right-hand-side expression, or vice versa. We compare the sketching performance with these two options on sketching assignment subjects. Based on our subjects, we find that sketching right-hand-side first performs faster than the other option, especially for the experiments with large search space. For
instance, Sketch4J spends 22.4 seconds sketching four assignments for the subject DLLAF by selecting right-hand-side first, while it takes 112.5 seconds with the other option. Yet this difference is not significant for the experiments with small search space (sketching less than four assignments) based on Wilcoxon tests ($p > 0.05$). We report the performance time by selecting right-hand-side first in our performance evaluation.

5 RELATED WORK

Program synthesis. Program synthesis has had numerous successes on synthesizing code in small well-defined domains such as bit-vector logic [14] and data structures [21]. They transform partial programs [23], input-output examples [6] or oracles [14] to decision procedures and SMT solvers. These techniques are very efficient in certain domains that have been fully modeled [10]. In particular, sketch-based synthesis [23] asks programmers to write a draft program containing missing expressions, and uses counter-example-guided inductive synthesis to complete the holes. Jsketch brings sketch-based synthesis to Java [13]. Given a partial Java program written in the sketch syntax, Jsketch translates the Java program to sketch synthesizer and transfers the synthesizer result back to executable Java code. Therefore, it is confined to the limitations of translation-based sketching approaches. Sketch4J does not translate program sketches and test suites to SAT solver, nor does it apply counter-example-guided inductive synthesis (CEGIS). It directly executes test cases and can sketch real-world Java applications that involve advanced language features like reflection.

Prospector [18] and SyPet [5] try to synthesize “jungloid code snippets” for imperative languages. A jungloid is a composition of API calls, where each method takes some arguments and returns a non-void value. Instead of sketching a chain of API calls, Sketch4J targets expressions, conditions, and blocks of assignments in the program. Similar to Sketch4J, CodeHind [8] can also handle real-world Java APIs and utilizes user-provided test cases. It synthesizes and evaluates code at runtime and uses an empirical probabilistic model to guide the search towards expressions that are more often used in practice. Their probabilistic model is trained from offline data corpus for API chain completion, yet this model is not effective on sketching condition expressions and assignment blocks, which vary from case to case.


Angelic Programming. Similar to our approach, angelic programming [2] leverages the non-deterministic backtracking algorithm [7], Barman et al. [2] embed the angelic choice construct into the Scala programming language and build a parallel backtracking solver to explore the scalability of their backtracking solver. Without any pruning strategies, this approach scales up much faster compared to Sketch synthesizer. We illustrate that our pruning strategy is efficient and the performance of Sketch4J on our dataset is comparable or even faster than a SAT-based synthesizer.

6 CONCLUSION

This paper presents a novel execution-driven sketching that synthesizes Java programs using a backtracking search. Our key insight is to introduce effective pruning strategies to reduce the search space for solutions and explore the actual program behaviors by executing the given test suite. Sketch4J evaluates code at runtime and hence can sketch real-world Java code that may use complex constructs of imperative languages, such as reflection and native calls. Our experiments show that our approach is comparable or even faster than a SAT-based synthesizer on our dataset and can be applied to open source projects with advanced language features.

REFERENCES