Checksims User Guide

Matthew Heon Dolan Murvihill

September 10, 2015

Contents

1 User Guide

Checksims is a tool for detecting source code similarities in an arbitrary number of user-provided programming projects. Its primary purpose is to flag potential cases of academic dishonesty in programming assignments. *Checksims* is not intended to detect academic dishonesty on its own, but rather to act as a tool to identify suspicious assignments for review by course staff.

Checksims accepts a number of submissions (programming assignments) as input, applies a tokenizer to transform each submission into a series of tokens, and then applies a pairwise similarity detection algorithm to all possible pairs of submissions. The results of the algorithm are then printed via an output strategy.

1.1 Installing Checksims

Checksims is distributed as an executable Java package (`.jar` file). As a Java application, *Checksims* is cross-platform and should run on any system capable of running a Java virtual machine (JVM). The provided Jar file is completely self-contained and requires no installation, and should be named as follows:

```
checksims-1.1.1-jar-with-dependencies.jar
```

Note that 1.1.1 represents the current version of *Checksims* at the time of this writing, and may be different for the version you receive.

Note that *Checksims* requires a Java 8 virtual machine. The latest version of the Oracle JVM is recommended, and can be found at the following URL:

```
https://www.java.com/en/download/index.jsp
```

A 64-bit processor and JVM are strongly recommended. Some *Checksims* detectors can consume a substantial amount of memory, potentially more than the 4GB maximum available to a 32-bit JVM. A 64-bit JVM can prevent a number of memory-related program crashes.

1.2 Running Checksims

Checksims is a command-line application, and is typically invoked from the operating system's shell or command prompt. The `.jar` file given can be run using Java as follows:

```
java -jar PATH/TO/CHECKSIMS_JAR.jar <ARGUMENTS>
```

It may be desirable to rename the provided `.jar` file or write a wrapping shell script to reduce the amount of typing required for this basic invocation.

1.2.1 Arguments

Checksims has two mandatory arguments: a single *glob* match pattern, and at least one directory to scan for submissions.

The *glob* match pattern is a shell-style match pattern used to identify files to include in submissions. Wildcard characters accepted by a shell are permitted; for example, providing a `*` does match every file in a submission, while `*.c` includes all C files. To ensure that these are not parsed by your shell, it is recommended to escape this pattern (typically using double quotes — `"*.c"` for example).

After the glob match pattern, one or more directories to search for submissions must be provided. *Checksims* assumes each subdirectory of these search directories is a submission.

It identifies any files matching the given glob pattern within a submission directory, append all matching files together, and tokenize the collection. By default, it is not recursive and will only identify files in the submission directory, but not in any subdirectories. An argument is provided to enable recursion through subdirectories to generate submissions (see section 1.2.1.1 for details). File names are discarded during this process, but the contents of all matching files will be present. Each submission is named for its *root* directory (that is, the subdirectory of a submissions directory); if a directory containing two subdirectories named "A" and "B" is provided as a search directory for submissions, two submissions named "A" and "B" will be created.

After creation, any empty submissions (no files found matching given pattern, or only empty files found) are removed prior to running the detection algorithm.

At present, there is no way of differentiating submissions beyond placing them within separate directories.

1.2.1.1 Optional Arguments Before the glob matcher, you may place a number of arguments to control the operation of *Checksims*. These are detailed below:

- `-a`, `-algorithm`: Specify algorithm to use for similarity detection. Available options are `linecompare` and `smithwaterman` at present, and can be listed with the `-h` option. If no algorithm is given, the default is used.

- `-c`, `-common`: Perform common code removal. Specify a directory containing common code (files within this directory will be identified using the same glob matcher as normal submissions).

- `-f`, `-file`: Output to a file. Must provide filename of output file as argument. The name of the output strategy used will be appended to the given filename as an extension. If more than one output strategy is given, more than one output file will be produced, each with the given filename but with differing extensions.

- `-h`, `-help`: Print usage information and available algorithms, preprocessors, and output strategies.

- `-j`, `-jobs`: Specify number of threads to use. Defaults to number of CPUs available on your system.

- `-o`, `-output`: Specify output format(s) to use. More than one can be provided; if so, separate them with commas. Available options are `html`, `csv`, and `threshold` at present, and can be listed with the `-h` option. If no output format is given, the default is used.

- `-p`, `-preprocess`: Specify preprocessors to apply. More than one can be provided; if so, separate them with commas. At present, the only available option is `lowercase`. Available options can be listed with the `-h` option. If this argument is not provided, no preprocessors are applied.

- `-r`, `-recursive`: Recursively traverse subdirectories when generating submissions.

- `-t`, `-token`: Specify tokenization to use. Available options are `line`, `whitespace`, and `character` at present, and can be listed with the `-h` option. If the `-t` option is not given, the default tokenization for the algorithm is used.
- `-v`, `-verbose`: Verbose debugging output.
- `-vv`, `-veryverbose`: Very verbose debugging output. Overrides `-v` if both are specified.
- `-version`: Print current version of *Checksims*.

Checksims contains built-in usage information and descriptions of its arguments, which can be printed by supplying the `-h` or `-help` flag. The output mirrors the information provided above, though it may be more up-to-date.

1.2.1.2 JVM Arguments

A number of arguments can also be passed to the Java virtual machine. These are usually placed in the command line as follows:

```
java <JVM_ARGS> -jar PATH/TO/CHECKSIMS_JAR.jar <ARGS>
```

These arguments are well-documented and can be used on all Java virtual machines. Several commonly-used flags are detailed below.

- `-d64`, `-d32`: Specify a 64 or 32 bit JVM, respectively. Some JVMs will only support 32 or 64 bit, but not both. Using a 64-bit JVM where available is preferred to enable the VM to use more than 4GB of memory.
- `-Xmx`: Specify a maximum amount of memory for the JVM to use. The number can be formatted as [Amount][Unit] where [Unit] is M for megabyte or G for gigabyte. Note that the number is specified immediately after the flag, with no `=` character. For example, to set the JVM to use at most 4GB of ram, specify `-Xmx4G` at the command line.

1.2.1.3 Sample Command Line

A typical command line invocation of *Checksims* is shown below.

```
java -d64 -jar PATH/TO/CHECKSIMS_JAR.jar -a smithwaterman -o html,csv -v -r
          -f ./out "*.c"  SUBMISSION_DIR_ONE   SUBMISSION_DIR_TWO
```

This instructs *Checksims* to do the following:

- Use a 64-bit JVM to prevent memory issues (`-d64`).
- Use an algorithm named `smithwaterman` to perform similarity detection (`-a`).
- Generate output using two strategies, `html` and `csv` (`-o`), and save this output in two files names `out.html` and `out.csv` (`-f`).
- Perform similarity detection on all files with extension `".c"` in directories `SUBMISSION_DIR_ONE` and `SUBMISSION_DIR_TWO`.

1.2.2 Common Errors and Solutions

This section contains a number of common errors that can occur while using *Checksims*, and suggests potential fixes.

1.2.2.1 Out of Memory Errors A Java Out of Memory exception occurs when *Checksims* uses all the memory available to the Java virtual machine. This is usually caused by running a complex comparison algorithm (for example, *Smith-Waterman*) on large submissions.

The first potential fix is to increase the amount of memory available to the JVM. This can be done by installing 64-bit Java and passing the `-d64` flag to use a 64-bit JVM (enabling the use of more than 4GB of memory). If a 64-bit JVM is already installed, a larger amount of memory can be provided using the `-Xmx` flag.

If more memory cannot be allocated to the JVM, it is also possible to reduce the amount of memory used by *Checksims*. This can be done in a number of ways. Firstly, reducing the number of threads used with the `-j` flag will cause a substantial decrease in the amount of memory used. Each thread uses roughly the same amount of memory, so a reduction from 4 to 2 threads should cause *Checksims* to use half as much memory. Furthermore, changing the tokenization used can impact the number of tokens stored, which has substantial implications for algorithm memory use. Changing from Character to Whitespace tokenization for *Smith-Waterman*, for example, will usually result in a 4-fold reduction in memory use.

1.2.2.2 No Submissions Detected In the case that *Checksims* cannot build any student submissions to compare, the first step is usually to check the glob match pattern used. Ensure that any characters that might be interpreted by your shell (for example, `*`) are properly escaped (single or double quoted on Linux or OS X, double quoted on Windows). Furthermore, check that the glob match pattern is syntactically valid for your platform.

Verify that you are passing *Checksims* a directory containing a number of student submissions, each of which is contained in a single subdirectory of the directory passed to *Checksims*. Even if each student submission is a single file, it must be contained in a subdirectory. Student submission directories may contain subdirectories themselves without issue.

1.3 Description of Tokenizations

Checksims breaks submissions into a sequence of tokens as they are read in. Several options are provided, each providing a tradeoff of speed for performance. Each similarity detection algorithm provides a default tokenization that has been chosen to optimize its performance for typical usage, but this default can be overridden at runtime if desired. This may be desirable, as tokenization has strong implications for algorithm accuracy and performance.

Only one tokenization is supported at any given time; it is impossible to request that *Checksims* tokenize one submission using character tokenization, and another using whitespace tokenizations. This is done to ensure a uniform basis for token comparison.

Three tokenization options are provided by default: `Character`, `Whitespace`, and `Line`. Their advantages and disadvantages are listed below.

1.3.1 Character Tokenization

The simplest tokenization method, `Character` tokenization, breaks a submission into the characters that compose it and builds a token for each character. Whitespace characters (spaces and newlines) are treated as tokens. No deduplication of whitespace is done — if a submission contains three consecutive spaces, all will be treated as independent tokens.

`Character` tokenization has the slowest performance of all the tokenization schemes as it generates far more tokens for the algorithm to process. However, for most algorithms, `Character` tokenization will be the most conducive to accuracy, as it can identify largely similar words and lines that would otherwise be ignored. `Character` tokenization also uses slightly more memory to store compared to the other tokenization schemes — usually not enough more to cause problems. However, `Character` tokenization may have a more serious impact on the amount of memory used by certain algorithms (such as Smith-Waterman).

1.3.2 Whitespace Tokenization

`Whitespace` tokenization breaks a submission apart at whitespace characters (spaces, tabs, newlines) to create tokens. Whitespace characters are removed as part of the splitting process, and are not included as tokens.

`Whitespace` tokenization represents a balance between performance and accuracy. With preprocessing (lowercasing to remove case ambiguity, etc), it can retain much of the accuracy of `Character` tokenization while substantially improving performance (assuming `Whitespace` tokens are on average four characters, a fourfold reduction in token count can be expected, even ignoring the deletion of whitespace).

1.3.3 Line Tokenization

`Line` tokenization splits the submission at line boundaries, creating a token from each line in the original. Non-newline whitespace characters (spaces and tabs) are retained.

`Line` tokenization represents the fastest but least precise tokenization option. It is capable of identifying exact duplication, but even trivial attempts to obfuscate similarities will prevent detection.

1.4 Description of Preprocessors

After a submission is converted into tokens, these tokens can then be manipulated to improve detection accuracy. This is accomplished by the use of predefined preprocessors. Two preprocessors are presently available. The `lowercase` preprocessor converts all letters to lowercase. The `deduplicate` preprocessors remove duplicated whitespace (spaces, tabs, and newlines).

1.5 Description of Algorithms

Checksims provides two detection algorithms at present. The first is *Smith-Waterman*. It offers accurate detection but slow performance. The second is *Line Comparison*. It is very fast, but not very accurate and easily fooled by obfuscation.

1.5.1 Smith-Waterman

The *Smith-Waterman* algorithm for string overlaying was originally developed to find optimal local alignments between DNA sequences for bioinformatics problems. Adapted to handle arbitrary alphabets, it proves a valuable tool for identifying similar token sequences. As a local alignment algorithm, it is capable of detecting sequences even when they are not completely identical. A small number of missing or unmatched tokens are tolerated, identifying more similarities than simply finding the longest common sequence. Furthermore, *Smith-Waterman* is guaranteed to identify the optimal local alignment — if common sequences exist, they will be found.

However, *Smith-Waterman*'s accuracy comes at a substantial performance cost. The algorithm itself is $O(n*m)$ where n and m are the lengths of the two sequences being compared; assuming equal and even growth of both sequences, the algorithm scales as roughly $O(n^2)$ (both for runtime and memory). For smaller submissions, Smith-Waterman can complete an entire class in a few minutes; for larger submissions, however, hours (or even days) may be required.

Because of the performance penalty of *Smith-Waterman*, it is recommended to use it with the `Whitespace` tokenization scheme, which it defaults to. This reduces the number of tokens present, greatly improving performance.

1.5.2 Line Comparison

The *Line Comparison* algorithm identifies identical tokens in both submissions. It is a trivial algorithm unique to Checksims, and notable for its speed. *Line comparison* hashes each input token, and identifies hash collisions (identical tokens). Similarity is reported on the number of collisions detected between the two submissions.

Line comparison makes one pass through each submission, and thus is $O(m + n)$. It is thus far faster than *Smith-Waterman*.

As the name of the algorithm indicates, it is only intended to be used with (and defaults to) the `Line` tokenization scheme. `Whitespace` tokenization results in a percent of shared words contained in submissions that is almost always very high and does not mean much about the actual similarity of two submissions. `Character` tokenization tends to result in greater than 99% similarity for all submissions, given that most all will be using the same basic alphabet (capital and lowercase letters, numbers, and language-appropriate syntax such as { or []).

Given the restriction to the use of `Line` tokenization, even small changes (for example, a single missing character) can result in otherwise extremely similar lines not being recorded as similar. It is possible that preprocessors could remove some trivial differences (for example, changes to whitespace or addition of comments). However, other alterations, like reordering statements or changes to identifier names, are very difficult to catch with preprocessors. *Line Comparison* is thus very limited in terms of accuracy.

1.6 Description of Output Strategies

Once an algorithm has been applied to the submissions, the results must be printed in a usable format so they may be used and interpreted. Output strategies determine how this

is done.

Results will often be presented as a square matrix, henceforth referred to as a *Similarity Matrix*. These matrices are built from the complete results of the similarity detection algorithm, and contain the similarity of every submission to every other. As the name would imply, a similarity matrix is a $N \times N$ matrix (with N being the number of submissions), with each cell containing a number representing the degree of similarity of one submission to another.

In a similarity matrix, the submissions used in similarity detection are counted, and a square matrix of that dimension is created. Submissions are each assigned a row and column. Every cell is initialized as the degree of similarity of the submissions that define its intersection (specifically, column submission's similarity to row submission). If the row and column submissions are the same, the cell is ignored (declared as empty). An example is shown in Figure 1. Each cell shows the similarity of the submission on the X axis with the submission on the Y axis. In Figure 1, the bottom-left corner cell shows the percentage similarity of submission A to submission C — that is, the proportion of submission A's tokens that are shared with submission C.

1.6.1 CSV

The CSV output strategy records output as a similarity matrix in comma-separated value format. This output format is computer-readable, not human-readable. It can be imported into Microsoft Excel or a number of other software statistics packages to generate statistics about detected similarities

1.6.2 HTML

The HTML output strategy produces a web page that can be opened in a typical web browser, presenting a colorized version of a similarity matrix. A color range (yellow to red) shows how similar each cell is, allowing easy visual identification of similar students and clusters of similarities.

1.6.3 Threshold

The threshold output strategy produces an ordered list of submission pairs that are sufficiently similar (by default, 60% or greater). The list is ordered from most to least similar, and omits all information about similarities below the threshold. This output strategy produces quickly actionable information about the most-similar submissions.

	A	B	C
A	N/A	Similarity of B to A	Similarity of C to A
B	Similarity of A to B	N/A	Similarity of C to B
C	Similarity of A to C	Similarity of B to C	N/A

Figure 1: A sample similarity matrix