Comparing Multiple Source Code Trees, version 3.1

Warren Toomey
School of IT
Bond University
April 2010

This is my 3rd version of a tool to compare source code trees to find similarities. The latest algorithm is not only elegant but extremely fast. A performance analysis of the algorithm is given.

Why Write Such a Tool?

- To detect student plagiarism.
- To determine if your codebase is ‘infected’:
  - by proprietary code from elsewhere, or
  - by open-source code covered by a license like the GPL.
- To trace code genealogy between trees separated by time (e.g. versions), useful for the new field of computing history.

Issues with Code Comparison

- Can rearrangement of code be detected?
  - per line? per sub-line?
- Can “munging of code” be detected?
  - variable/function/struct renaming?
- What if one or both codebases are proprietary?
- How can third parties verify any comparison?
- Can a timely comparison be done?
- What is the rate of false positives?
  - of missed matches?

Code Comparison Requirements

- Must permit the detection of code rearrangement to some extent.
- Must be reasonably fast.
- Any code representation must be exportable without divulging the original code.
  - this allows others to verify any code comparison.
  - however, something of the original code's structure has to be divulged.
- If possible, it should detect different coding of the same algorithm: renamed variables, constants,

Original Idea: Lexical Comparison

- Break the code in each tree into lexical tokens, then compare runs of tokens between trees.
- This removes all the code's semantics, and deals with code rearrangement (but not code "munging").
- Example tokens:
  - Single chars: [ ] { } + - * / % !
  - Multiple chars: ++ && += !=
  - Keywords: int char return if for while do break
  - Literal values: identifiers, "strings", 'x', numbers

Advantages of a Lexical Approach

- Code does not need to be compilable.
- Non-experts can “see” the similarities.
  - e.g. in a courtroom setting, once similar code has been identified
- Other approaches:
  - compare intermediate forms, e.g. bytecode
  - compare functional results
  - identify and compare algorithmic units

CTF Files: Serialised Token Streams

- Each source code tree is converted into a serialised token stream: a *CTF* file.
- Each token is represented by 1 byte.
- Literal values are hashed down to 2 bytes.
- Filenames and timestamps are also included.
- A CTF file reveals the code structure, but literal values are not revealed.
- Allows for the export of a code tree to a 3rd party without revealing the original source code.

Example of a CTF Stream

```
385: do {
386: id891 = id64003 [ id100 ] ;
387: id64003 [ id100 ] = NUM48 ;
388: if ( id891 > NUM408 )
389: id891 = NUM426 ; else
390: if ( id891 > NUM446 )
391: id891 = NUM446 ; else
392: id891 = NUM48 ;
393: id55378 -> id32607 = id891 ;
394: id55378 += NUM49 ;
395: id32068 ( id100 ++ ) ;
```

1st Comparison Approach

```
foreach (token in one CTF file)
{
  walk the other CTF file to find
  a matching run of tokens;
}
```

- O(M * N), where M,N are the number of tokens in each file. Very, very slow.
- This version could not compare a CTF file to a set of CTF files, only to one other CTF file.

2nd Comparison Approach

- Break each token stream into groups of N consecutive tokens: a token *tuple.*
- Find tuples in other code trees that match.
  - this indicates a potential run of similarity.
- Once all tuple matches are found, merge them to find the full extent of the runs of similarity.
- Much faster than v1, and allows multiple trees to be compared simultaneously.
- But the merge component is very ugly.

3rd Comparison Approach

- v2 sliced the streams up into N-token tuples, found matches in the (unordered) set, and then rebuilt the full runs of similarity.
- By having an unordered set of tuples, more work had to be done to merge partial runs.
- In v3, we walk each CTF file from one end to another, making token tuples.
- If we find a match, we know exactly which existing runs may need to be extended.

3rd Comparison Approach

```
for (all tokenised source trees) {
  for (all consecutive runs of N tokens from the source files in the tree) {
    build a token tuple T of the N tokens in the run plus their identifiers;

    for (each existing tuple T2 in the tuple list which matches T) {
      if (T and T2 would extend an existing comparison run R) {
        modify R so that T and T2 are now the end tuples of the run;
      } else {
        create a new comparison run R where T and T2 are the start and the
                                                              end tuples of the run;
        add R to the list of comparison runs;
      }
    }
    add tuple T to the tuple list;
  }
}
```

Why is This Approach Better?

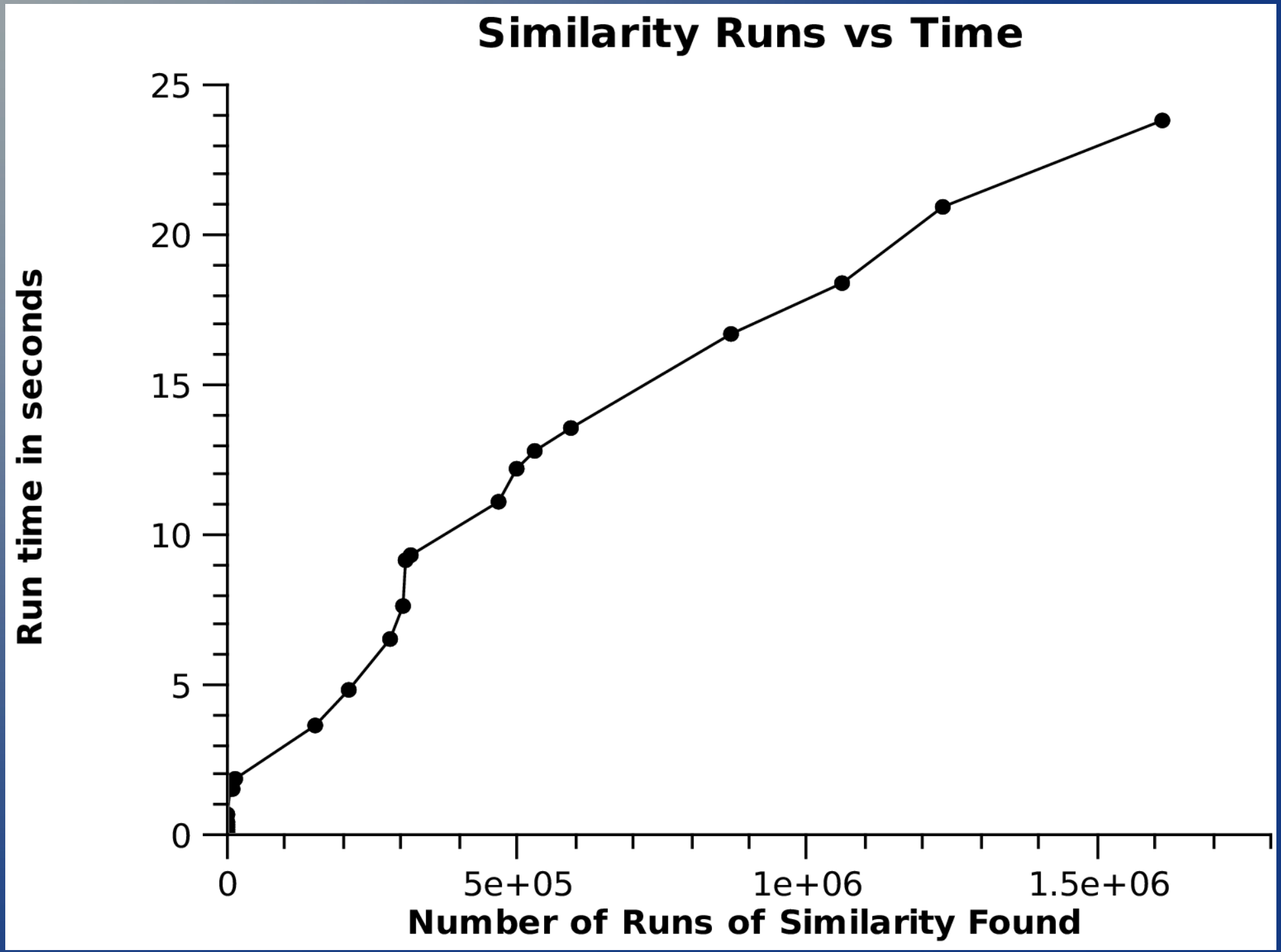
- When a tuple match is found, there are only a few incomplete runs from the last tuple, so finding the run to extend is easy.
- The algorithm is 5-10 times faster than v2.
- The algorithm's implementation is 40% smaller than v2, and it is much more elegant.
- The algorithm now seems to scale well based on the size of input. v1 was O(M*N) and v2 seemed to be $O(N^2)$, where N= total number of tokens.

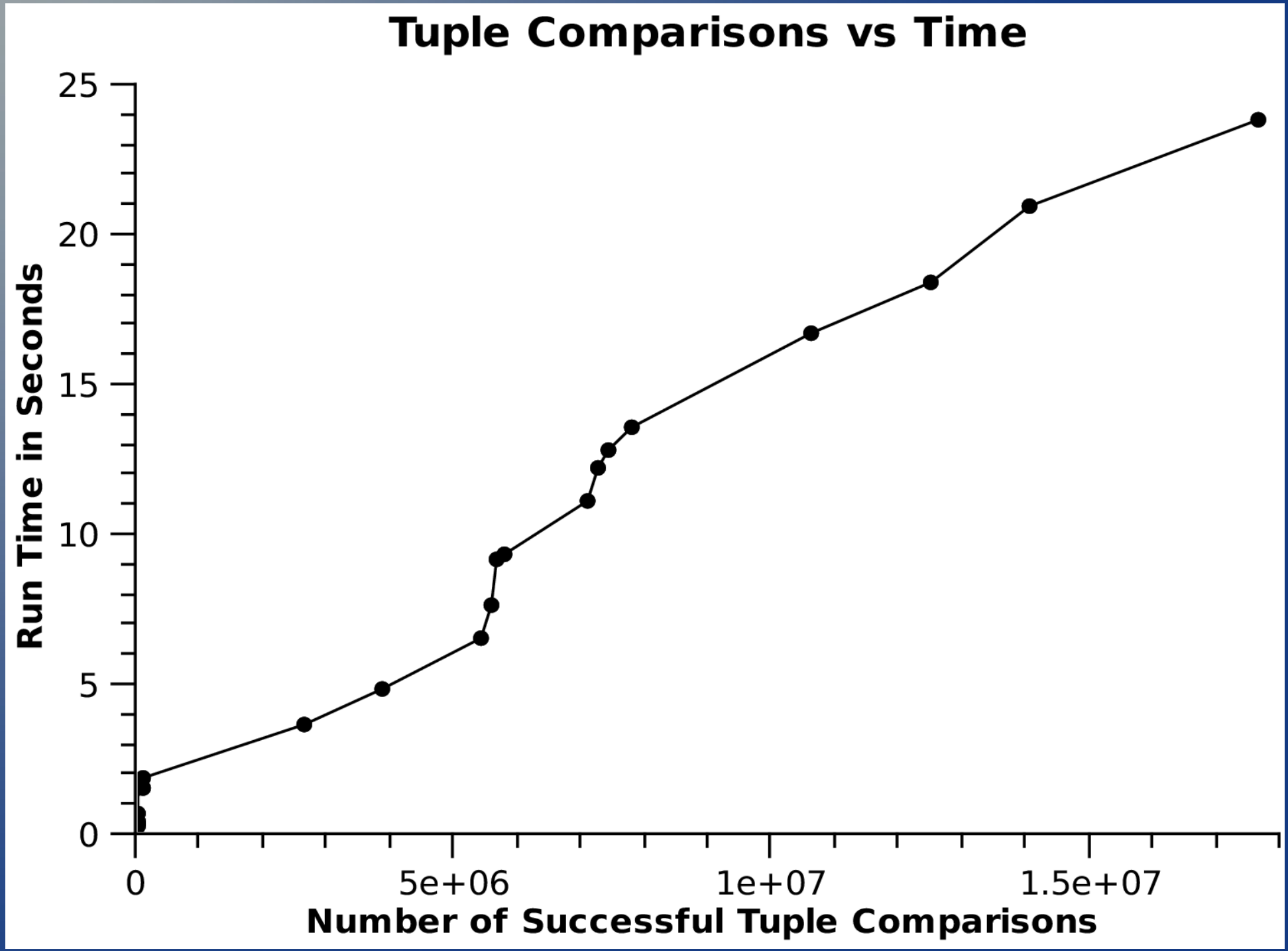
Heuristics Used

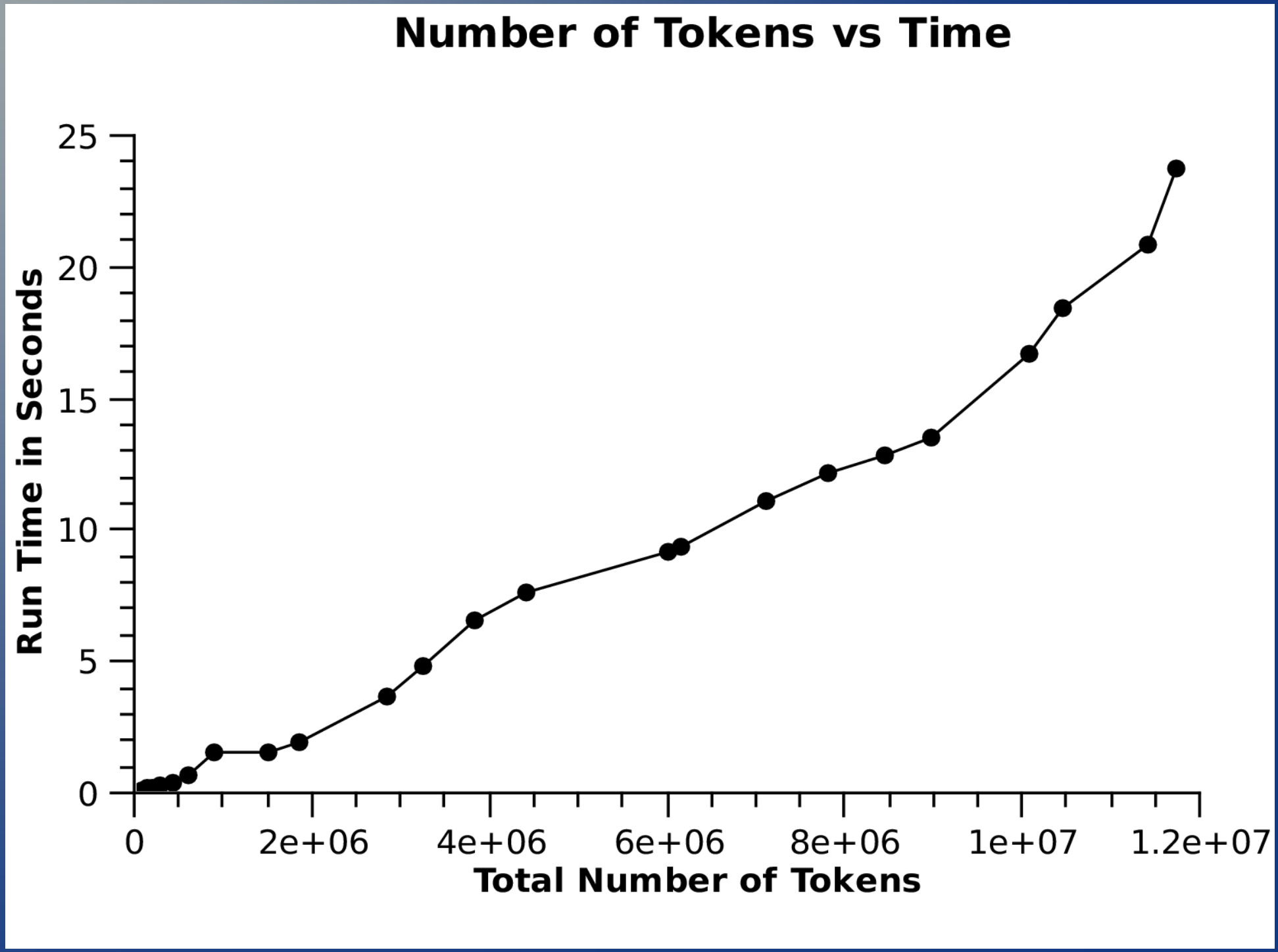
- Tuples are searched using hashes + linked lists.
  - very low probability of false positives
  - 1 in 2^{32} for runs of N tokens, near zero for larger runs of similarity
- When a tuple match is found, existing incomplete runs are searched using hashes.
  - low probability (1 in 2^{24}) that an existing run will not be extended
  - Instead, two separate runs will be reported

Performance Analysis

- A number of code trees, some related, up to 2MLOC were chosen as representative input.
  - Several UNIX kernel trees
  - Two Linux kernel trees
  - Other application code trees
- Comparisons were done cumulatively, to measure performance as input size increased.
- Several metrics:
  - run time, # of tokens, # of tuple comparisons,
  - # of complete similarity runs found.


Similarity Runs vs Time
Run time in seconds
25
20
15
10
5
0
0
5e+05
1e+06
1.5e+06
Number of Runs of Similarity Found

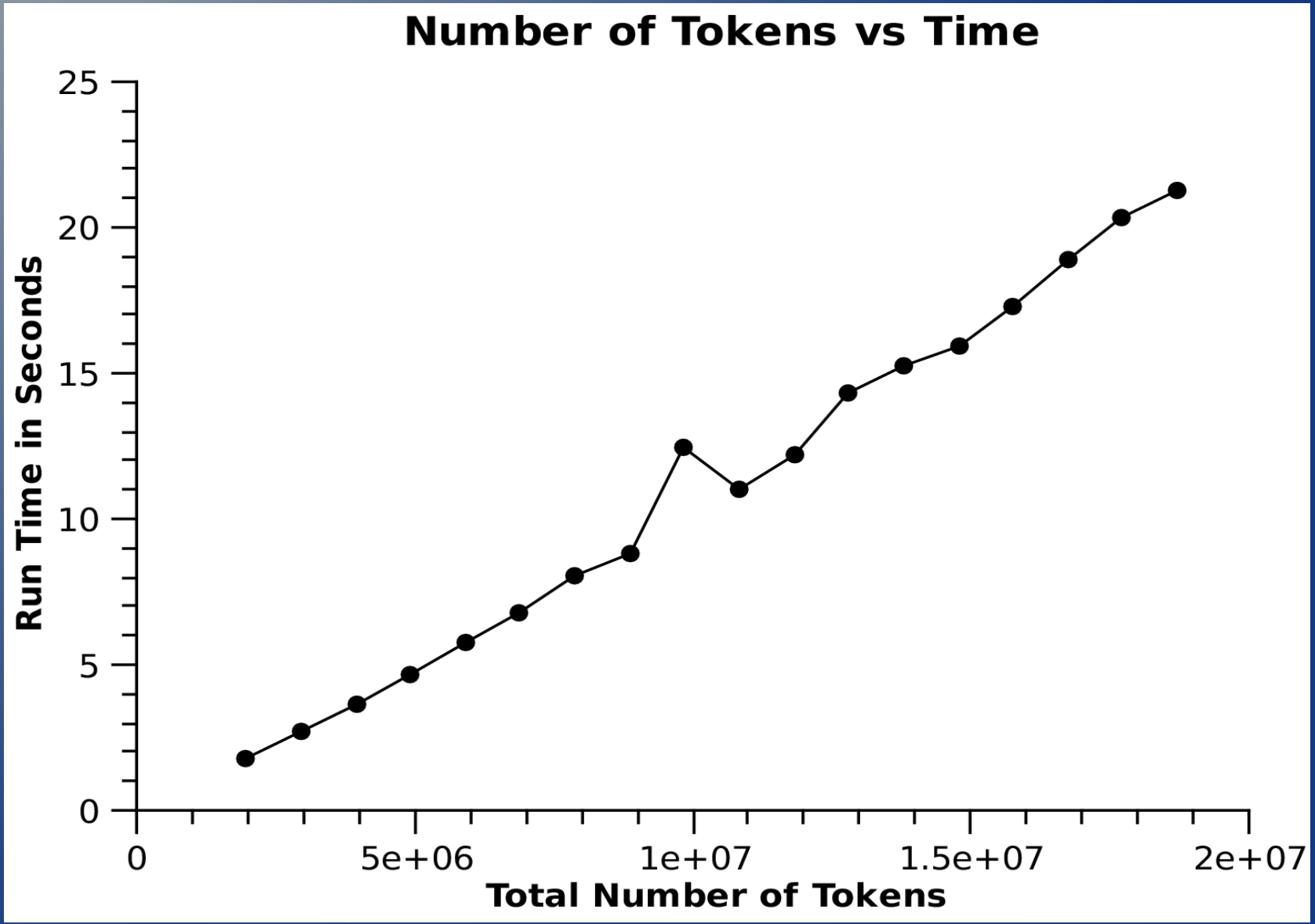

Tuple Comparisons vs Time
Run Time in Seconds
0
5
10
15
20
25
0
5e+06
1e+07
1.5e+07
Number of Successful Tuple Comparisons


Number of Tokens vs Time
Run Time in Seconds
0
5
10
15
20
25
0
2e+06
4e+06
6e+06
8e+06
1e+07
1.2e+07
Total Number of Tokens

Performance Analysis

- Both the heuristics and the specific input affects the program's performance.
  - for small input, hashing dominates the run time
  - for related trees, the cost to search large numbers of incomplete runs dominates the run time
- Run time vs. number of tuples: could be linear, could be exponential.
- I ran the tool with several trees of random tokens, to determine if non-similarity caused an exponential performance

Random Token Trees



Conclusion

- New algorithm is more elegant, more efficient than previous algorithms, and seems to scale more linearly with input size.
- On a 2GHz P4 with only 1G of RAM:
  - 15 trees, 2.87MLOC, 2.8M runs => 36 seconds
- Biggest drawback is memory usage:
  - The above requires nearly all the RAM
  - Memory usage is proportional to # tokens + # of runs found