Snakefinder: Simple Python Source Search

Vaibhav Mallya

April 24, 2009

1 Abstract

As programs have grown in complexity, code bases have grown in complexity as well. Managing and searching these complex source trees is important for improving developer productivity. Although there do exist a number of code search tools, all of them have shortcomings for the simple case of indexing all projects on a single system. We present a Python-based tool, Snakefinder, that is fast, simple, and allows hierarchy-aware querying for scattered Python source code on a single computer.

2 Introduction

Most developers trying to develop or improve an application will need to frequently search the codebase to comprehend flow and structure. A good search tool should therefore allow for rapid querying and accurate, up-to-date retrieval, and simple, fast indexing. Additionally, since good developers reuse components from other applications, a good search tool should easily allow a developer to index as many projects as their system holds. We briefly review three popular code search systems to understand some popular approaches, and where they may be improved upon.

3 Brief Reviews

3.1 Google Code Search

+ Fairly powerful regex-based search mechanism
+ Allows basic querying for classes, functions, and modules
− Minimal knowledge of hierarchy
− Not local
− Code has to be uploaded publicly

– Obvious issues with security, privacy, and so forth

– Developers have no control over index updates

3.2 grep/awk/ack

+ Local system search

+ Works well for small trees

+ Widely-distributed, widely-known

– No indexing

– Slow for non-trivial code bases or sparsely distributed files

– No knowledge of hierarchy

– Does not scale

3.3 OpenGrok

+ Very powerful search and analysis capabilities

+ Can search large, heterogenous code bases rapidly

+ Locally installed

+ Built on proven technologies - Java and Lucene

– Non-trivial to deploy and run system-wide across projects

– Hierarchy-aware querying, but only at the file/directory tree level

One property common to all these systems is that they attempt to index or search through all terms in all source files. While clearly beneficial for thoroughness and advanced analysis, this leads to significantly larger index sizes, longer index construction times, and usually, longer query times. Another problem is that basic queries can return too many useless results with too little context. Therefore, total awareness does not seem like it is an ideal solution, especially if certain types of information are significantly over-utilized or under-utilized compared to the others.

An informal study conducted by the authors of this paper found that when developers search for existing code snippets, they usually do not query at the level of individual variables and instantiations. Instead, they usually query at the block level - that is, at the level of functions, classes, files, and modules. It seems intuitive that we can conquer the common case here by ignoring most low-level code.

Additionally, when trying to find a block-level element, developers usually have some rough knowledge of hierarchy. For example, a developer looking for the a function called "startDatabase" may know that it exists somewhere inside a class or module called "Database", but may not know exactly where it is. Unfortunately, most existing code search engines do not have this level of awareness - they do not index the full block hierarchy of a source tree. OpenGrok does feature hierarchy-awareness at the level of files and directories, but has little knowledge of hierarchy within the files themselves. Snakefinder attempts to solve these majority-case problems for Python source code.

4 Snakefinder's Approach

4.1 Querying

Query languages themselves are a fairly well-studied problem, and there already exist a number of useful query languages. For the purposes of this paper, we implemented our own query language, roughly modeled after SQL. The BNF for the language is as follows:

⟨regex⟩ = [any valid python regex without >>, >, or ∼.]
⟨blocktype⟩ ::= file | class | def
⟨part⟩ ::= ⟨blocktype⟩ = ⟨regex⟩ | ⟨regex⟩
⟨base⟩ ::= ⟨part⟩ | ⟨part⟩ ∼ ⟨base⟩ | ⟨part⟩ >> ⟨base⟩ | ⟨part⟩ > ⟨base⟩

There are three important properties of note:

- The possible ∼ after a part denotes an inclusive OR. Parts can be OR'ed together to form a component.
- There is no recursion necessary for parsing; the query is simply scanned left to right.
- It is possible to provide just a raw regex without a type; this will search all supported block types.
- In this iteration, full module indexing and querying support is not included due to time constraints and the complexity of python's module construction rules; currently, files are considered the only type of module. Full support for class and function querying is included, however.

4.2 Indexing

The indexer generates the following:

- A directed, acyclic network graph of block-level elements. This graph is recursively defined in terms of nodes. Every node contains

  - The URL (file path, lineno, content snippet)
  - A set for all "class" child nodes, a set for all "function" child nodes, etc
  - We can obtain the set of all child nodes for a node by taking the union of these sets

- Disjoint sets for each block type, - a set of all files, a set of all classes, etc
  - This allows for simple scanning to find of all elements of a type that match some regex
  - The universal set of all elements - all indexed blocks- is the union of these sets

5 Results

We ran our indexer over the large, densely populated home directory tree of the author of this paper. We feel this was an accurate simulation of a real-world use-case - we had several whole python installations, one-off scripts scattered haphazardly, and several large, irrelevant, source trees of non-python projects.

- Total files scanned: 200,770
- Python files matched: 6,571
- Net size of Python source files: 60.0 megabytes
- Time required to index: 118.175 seconds
- Index size: 11.9 megabytes

Upon a quick glance at the index size, it seems like the minimalist approach has not been as successful as desired. However:

- There has been no compression utilized whatsoever - the index is simply a serialized Python data structure
- The data structures themselves have more optimal substitutes in the Python standard library

All told, the index is two or three times larger than it could be with some more invested effort and care. In fact, manually compressing the archive post-mortem with Bz yield an index of less than 3 megabytes! Queries were straightforward to test; this approach yields valid queries. We include some sample queries and outputs as a separate attachment. Our hierarchical querying mechanism takes no more than a second to return the full set of valid results and is essentially negligible in terms of time.

6 Limitations

6.1 Dynamism

Python is a dynamic language, and has a number of powerful constructs such as metaclasses, dynamic module reloading, and monkeypatching that interfere with static analysis. Unfortunately, there is no simple way (or in some cases, no way at all), to analyze source files for certain properties. Snakefinder simply sidesteps all of these issues by ignoring them; in most cases, this should not affect indexing of well-written code. A notable exception is the popular Django web application framework which heavily utilizes the aforementioned techniques. Snakefinder will not be able to fully index the block hierarchies of Django applications.

6.2 Lexical Analysis

Python is bundled with a number of powerful tools for introspection and source analysis. Python's lexical analyzer in particular would have come in quite useful for this project, but suffered from lack of good code samples. As a result, we wrote our own simple indentation-based analyzer for the purposes of this paper. Unfortunately, our analyzer is imperfect, so if it encounters, for example, a function whose default arguments are split across multiple lines, it will only index the content of the first line. For a future version of the indexer, this is an obvious area for improvement.

6.3 Parallelization

Finally, with the increasing ubiquitness of multi-processor and multi-core computers, it seems natural that we could parallelize the processing aspect of an embarassingly parallel problem such as this one. I/O boundedness would come into play, but multi-programming should introduce a noticable improvement to the algorithm.

7 Conclusion

Based on both the positive and negatives aspects of some popular source code search tools, we investigated a new approach for hierarchy-aware indexing and retrieval of source code that eschews total awareness in favor of handling the majority case well. Despite some shortcomings in our current implementation, we feel we have demonstrated that the approach itself shows promise.