Software Reuse Detection Using an Integrated Space-Logic Domain Model

Wei Qu Michael Jiang
Motorola Labs
Motorola Inc., Schaumburg, IL
wei.qu@motorola.com, michael.jiang@motorola.com

Yuanyuan Jia
Bioengineering Department
University of Illinois, Chicago, IL
yjia2@uic.edu

Abstract

Software reuse detection is a challenging task due to various modifications and the large size of software code. Most existing approaches adopt a token-based software representation and use sequential analysis for software reuse detection. Due to the intrinsic limitations of such a space-domain analysis, these methods have difficulties to handle statement reordering, insertion and control replacement. Recently, logic-domain models such as program dependent graph have been exploited to solve these issues. Although they can improve the performance in terms of accuracy, they introduce additional problems. Their computational complexity is very high and dramatically increases with the software size, limiting their applications in practice. In this paper, we propose a novel software reuse detection framework using an integrated space-logic domain model. It embeds a logic-domain software model into a space-domain analysis and thoroughly exploits the software's information from both space domain and logic domain. The preliminary experimental results have demonstrated the superior performance of the proposed approach compared with other methods.

1. Introduction

Software reuse detection has received a significant amount of attention in recent years due to its numerous applications such as source code plagiarism detection, open source localization, intellectual property infringement uncovering, software debugging, etc [1], [3], [11]. Detection of the exactly reused software code is much easier since it can be solved reasonably well by using regular text search techniques. However, reuse detection for software with modifications is a more difficult task [1], [3]. In addition to all of the challenging problems inherent in text searching, software reuse detection must deal with alterations from the simple modifications such as reformatting, comment changes, identifier renaming to more complicated changes such as statement reordering, insertion, and control logic changes, some of which may be very hard for human beings to identify because of the tricky variations and the large software size.

Most early efforts for software reuse detection relied on the use of tokenization and space-domain sequential analysis. Tokens are the basic units in a programming language such as keywords, operators, parentheses, etc. Such methods usually consist of two stages: firstly, a parser or a lexical analyzer filters a program into a sequence of tokens; then a sequential analysis method is used to compare the token sequences and detect the similar counterparts. A string-based approach was proposed by Baker in [1]. Wise [14] proposed a YAP algorithm, which relies on the "sdiff" function in UNIX to compare lists of tokens for the longest common sequence of tokens. Gitchell and Tran presented a SIM plagiarism detection system comparing token sequences using a dynamic programming string alignment technique in [7]. JPlag [11] and MOSS [12] are two widely used token-based tools for programming plagiarism detection especially in academic area. Recently, Kamiya et al. [8] proposed CCFinder, a clone detecting technique with transformation rules and a token-based comparison. Chen et al. designed a token-based system called SID in [3]. It uses a metric based on Kolmogorov complexity to measure the shared information between two programs.

Although the token-based sequential analysis methods can handle format changes and identifier renaming since the blanks and comments are ignored by the parser and variables of the same type are filtered into the same tokens, they have intrinsic limitations for software reuse detection due to only using "space-domain" sequential analysis. For example, reordered or inserted statements can break a token sequence which may otherwise be regarded as a duplicate to another sequence. These limitations have recently inspired researchers to exploit other domain information. Baxter et al. [2] proposed a tool that transforms source code into abstract-syntax trees (AST) and detects code reuse by finding identical subtrees. However, it may introduce many false positives because two code segments with same syntax

```
int function1()
{
    int index = 1;//Max is 100
    int timer = 0;//count the times
    while (index < 100)
    {
        timer = function2(index, timer);
        index = function3(index, 1);
    }
    printf("Timer is %d\n", timer);
    return index;
}
int function2(int p1, int p2)
{
    ...
}
int function3(int p1, int p2)
{
    ...
}
```

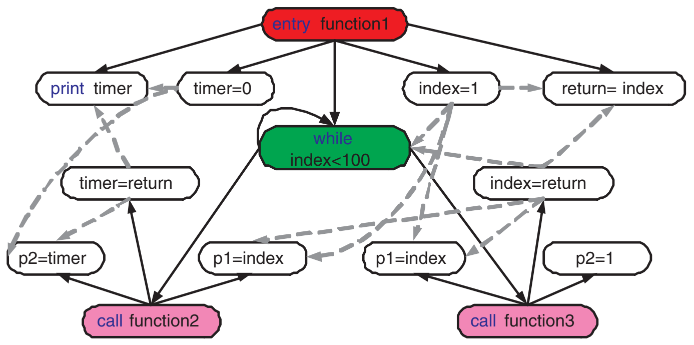


Figure 1. An example of the Program Dependence Graph (PDG).

subtrees may not be necessarily reused code. Komondoor et al. proposed to use program dependence graph (PDG) and program slicing to find isomorphic subgraphs and code duplication. Another PDG-based approach was proposed by Krinke in [9]. This notion is carried further in the work of Liu et al. [10], which improves the plagiarism search efficiency by a PDG-based GPlag algorithm. This work provides a very promising direction to resolve the problems of software reuse detection in logic-domain analysis. However, since general subgraph isomorphism is NP-complete [6], these logic-domain approaches suffer from the exponentially increased computational complexity with the size of software code, and thus limit their use in practice. Although a lossy filter may partially remedy this issue, it does not fundamentally save the computational complexity of the graph-based algorithms and may introduce much more false negatives and false positives.

In this paper, we present a novel software reuse detection framework using both space-domain and logic-domain analysis. In particular, it discards the software tokenization, which sacrifices too much information to tolerate identifier renaming. Instead, it uses a logic-domain model, program independence graph, as software representation. Efficient space-domain fingerprinting is exploited to generate "seed matches". Graph matching is also used to recover the lost information and enhance the detection accuracy. The rest of the paper is organized as follows: In Section 2, we briefly review the related work. In Section 3, we present the proposed joint space-logic domain software reuse detection framework. Experimental results are presented in Section 4. Finally, we provide a summary in Section 5.

2. Related work

Since we will use local fingerprinting algorithm and program dependence graph in our model, we give a brief summary of these techniques in this section.

2.1. Local fingerprinting algorithm: winnowing

Winnowing [12] is an efficient local fingerprinting algorithm designed to guarantee that matches of a certain length are detected. Given a set of documents, the purpose is to find the substring matches between them that satisfy two properties: (1) if there is a substring match at least as long as the guarantee threshold t, then this match is detected; (2) we do not detect any matches shorter than the noise threshold k. For a sequence of hashes $h_1 \ldots h_n$, if $n > t-k$, then at least one of the h_i must be chosen to guarantee detection of all matches of length at least t. This suggests an approach as follows. Let the window size be $\omega = t-k+1$. Consider the sequence of hashes $h_1 h_2 \ldots h_n$ that represents a document. Each position $1 \le i \le n-\omega+1$ in this sequence defines a window of hashes $h_i \ldots h_{i+\omega-1}$. To maintain the guarantee, it is necessary to select one hash value from every window to be a fingerprint of the document. Winnowing is a strategy which satisfies the above requirements. It has been demonstrated that this algorithm can work well in practice. Specifically, in each window, Winnowing selects the minimum hash value. If there is more than one hash with the minimum value, select the rightmost occurrence. All selected hashes then construct the fingerprint list of the document. For the implementation details of Winnowing algorithm, we refer readers to [12].

2.2. Program dependence graph

The Program Dependence Graph (PDG) represents a program as a directed and labeled graph in which the nodes are statements and predicate expressions such as variable declarations, assignments, etc., and the edges incident to a node represent both data values and control conditions [5]. Following the notation in [5] [10], we define a PDG as follows:

The program dependence graph G for a subroutine is a 4-tuple element $G = (V, E, \mu, \delta)$, where V is the set of *program nodes, $E \subseteq V \times V$ is the set of edges, μ is the set of nodes' types, and δ is the edges' types.*

Fig. 1 illustrates an example of PDG. The nodes represent individual statements and predicates of a subroutine. The edges represent the data and control dependence among statements and predicates where the solid lines are control flow and the dash lines are data flow.

Similar to [10], we use subgraph isomorphism to analyze the PDGs. A subgraph isomorphism can be defined as follows [10]:

A bijective function g: $V \rightarrow V'$ is a subgraph isomorphism from G to G' if there exists a subgraph $S \subset G'$ such that f is a graph isomorphism from G to S, where the graph isomorphism is defined as a bijective function f from a graph $G = (V, E, \mu, \delta)$ to a graph $G' = (V', E', \mu', \delta')$ if $\mu(v) = \mu'(f(v))$, $\forall e = (v_1, v_2) \in E$, $\exists e' = (f(v_1), f(v_2)) \in E'$, such that $\delta(e) = \delta(e')$, $\forall e' = (v_1', v_2') \in E'$, $\exists e = (f^{-1}(v_1'), f^{-1}(v_2')) \in E$, such that $\delta(e') = \delta(e)$.

γ-isomorphic is a specific subgraph isomorphism. It can be further defined as:

A graph G is γ-isomorphic to G' if there exists a subgraph $S \subseteq G$ such that S is subgraph isomorphic to G', and $|S| \geq \gamma|G|$, $\gamma \in (0, 1]$.

For the details of program dependence graph and its implementation, we refer readers to [5], [10].

3. An integrated space-logic domain model

In this section, we present our integrated space-logic domain model for software reuse detection. Fig. 2 shows the system structure. It has three modules: (1) joint space-logic analysis; (2) space-domain matching; and (3) logic-domain matching. For a better understanding, we also present an example below the diagram in Fig. 2.

3.1. Joint space-logic domain analysis

Suppose we have two programs: the original program and the target program. Each of them can be a large software including many files. For simplicity, we assume that all the files within a software are concatenated together to construct a single sequence. The purpose of software reuse detection is to find the duplication between these two sequences. For each sequence, we first generate the corresponding program dependence graph. Then instead of directly searching the identical subgraphs between two PDGs, we reorder the graphs according to the source code location, namely, assigning nodes to lines. For example, since nodes n_1, n_2, n_3 are generated from line number 101, they are put back into line number 101 after reordering. For simplicity, we only show the rearranged nodes and ignore the edges in the example of Fig. 2. Finally we hash the reordered PDGs line by line.

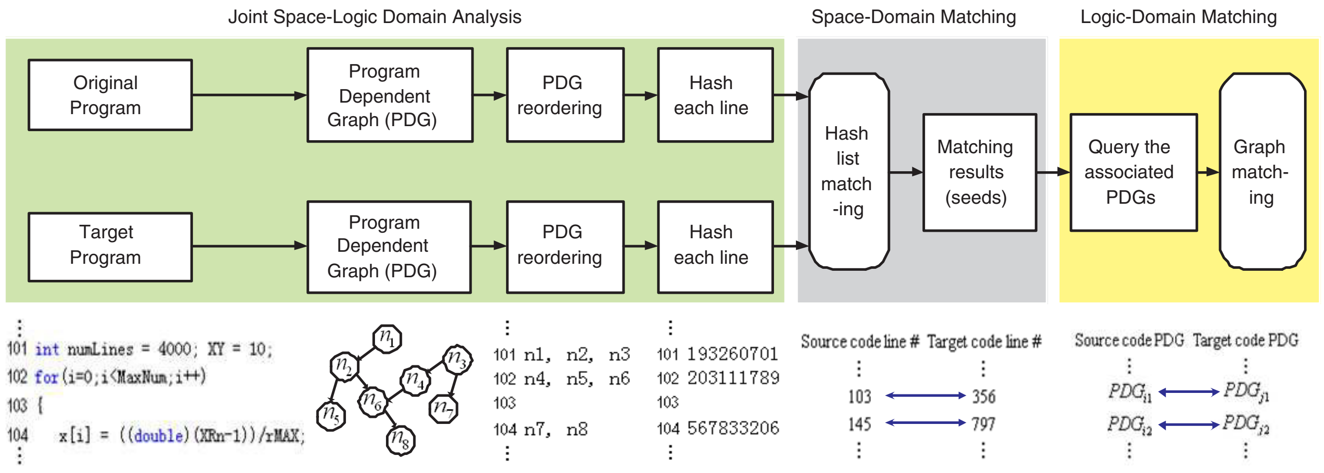
Compared with the token-based approaches [14], [12], [11], the proposed model substitutes the tokenization process by a logic-domain software model. As an efficient representation of software logic structure, PDG captures a program's inherent logic information. It has the ability to tolerate not only the simple modifications such as format alteration, and identifier renaming, but also more complicated changes such as statement reordering, insertion and control replacement.

3.2. Space-domain matching

Space-domain matching algorithms share an advantage that the computational complexity is much lower than that of graph matching. In order to make software reuse detection efficient enough for practical applications, we firstly apply a space-domain matching between the two hash lists. Different search techniques have been studied in the literature [7], [12], [11]. In this paper, we adopt Winnowing algorithm [12], which was briefly introduced in Section 2. It can efficiently find most duplicated code pairs between two programs. For instance, in the example of Fig. 2, line number 103 in the original program has been found identical to line number 356 in the target program. Similarly, line number 145 of the original program is equal to the line number 797 of target program. We use these line pairs as seeds for the next step logic-domain matching.

3.3. Logic-domain matching

Winnowing algorithm achieves efficient software reuse detection by compressing the data and sacrificing part of the information. Only the selected lines called fingerprints are compared between the two sequences. All other lines are ignored to trade for speed. Obviously, such kind of methods can not avoid missing lots of matches and thus increase the false negative rate. Moreover, the detected lines are often very scattered. It is desirable to merge them together to get bigger and more meaningful blocks. In the existing literature, this is usually achieved by using a simple spatial threshold [11]. For example, if the distance between two detected lines in a sequence is less than the threshold, these two lines and the lines between them are regarded to be one pattern and merged together. It is easy to see that this scheme is problematic. First of all, how to determine the threshold is challenging. It is usually very sensitive to different software. Even for one particular software, different thresholds may be needed for different parts. Secondly, the neighboring fingerprints may not have any relationship between each other at all. Neither are the lines between them. Simple concatenating fingerprint lines by spatial in-



```
101 int numLines = 4000; XY = 10;
102 for(i=0;i<MaxNum;i++)
103 {
104     x[i] = ((double)(XRn-1))/rMAX;
```

```
101 n1,  n2,  n3
102 n4,  n5,  n6
103
104 n7,  n8
```

```
101 193260701
102 203111789
103
104 567833206
```

Figure 2. System structure

formation may generate additional errors. Finally, such a scheme is not robust to software reordering and insertion. Since merging is made based on only spatial relationship, it may generate quite different results on reordered software even no other modifications are made.

To handle the above problems, we propose to further apply a logic-domain matching. For each "seed" line pair, we retrieve the associated PDGs. Then we use graph matching algorithm on each PDG pair to further detect if there is any matched subgraph inside the PDG pair. In order to reduce computational cost, it only outputs the first match in case there are several. Different graph matching algorithms may be used [4], [13]. In our experiments, we applied VF algorithm [4] to detect the isomorphic subgraphs. Although general subgraph isomorphism is NP-complete, this algorithm is computationally efficient for graphs with hundreds and thousands of nodes. To allow more trivial changes, Liu et al. applied γ-isomorphic testing instead of full subgraph isomorphism in [10]. For a better comparison, we also use γ-isomorphic here. Specifically, if h, $h \subset H$ is γ-isomorphic ($0 < \gamma \leq 1$) to h', $h' \subset H'$, the subroutine associated with h' can be regarded as software reuse of that of h.

As we can see, during logic-domain matching, each seed "pulls" out a node net (PDG). Therefore, although space-domain matching loses some information due to using fingerprints, logic-domain analysis provides the ability to recover the lost information as long as one node from a PDG is found as a seed in the space-domain matching. Moreover, the problem of merging neighboring detected lines is also solved in an innovative way. Specifically, there is no distance threshold needed anymore since we don't merge the detected lines based on the spatial information at all. Instead, the detected reused source codes are merged in logic-domain through the associated PDGs.

Table 1. Statistics of the test C programs

Name	Tracker	Inter	Driver	Commu
LOC	895	7,651	11,986	23,156
Project Function	video processer	interface program	device driver	cellphone tester

4. Experimental results

To demonstrate the effectiveness and efficiency of the proposed approach, we performed experiments on different programs. Due to the limited space available, we will only present a few examples in this paper. Statistical data of the test programs used in our experiment is provided in Table 1. The detection performance of the proposed method was compared with JPlag [11], MOSS [12], and GPlag [10], respectively. All experiments were performed on a 3.2GHz Pentium IV PC. We implemented our algorithm and GPlag using Matlab and C/C++ without code optimization.

4.1. Robustness analysis

To evaluate the robustness of the proposed approach against the state of the art [11], [12], [10], we set up two different experiments.

In the first experiment, we manually applied the following six kinds of modifications on the program `Tracker`: (1) changing comments; (2) adding blank space lines; (3) renaming identifiers; (4) reordering statements; (5) inserting statements; (6) control replacement (for example, substitute *while* with *for*). In Table 2, we show the number of different changes (ground truth). Then, we tested the four

Table 2. Results of robustness experiment 1

Changes	GT	JPlag	MOSS	GPlag	Ours
Adding Comments	11	0	0	0	0
Adding Space	9	0	0	0	0
Renaming	63	0	0	1	0
Reordering	24	21	16	3	0
Insertion	26	10	14	5	0
Control Replacement	29	29	28	1	0

Notes: GT is the ground truth.

Table 3. Results of robustness experiment 2

Method	JPlag	MOSS	GPlag	Ours
FNR	77.2%	53%	11%	3.5%
FP	164	93	0	0

Notes: FNR is the false negative rate; FP is the false positive.

approaches on the generated target program. Table 2 also provides the experimental results. Instead of using a successful rate, it lists the number of failures to show what kind of modification can cheat the individual reuse detection tools. As we can see, all four algorithms could successfully handle format alteration such as adding comments or blank space lines. Identifier renaming was also solved very well. However, due to only using token-based space-domain analysis, JPlag and MOSS could not deal with statement reordering, insertion and control replacement. By exploiting a more effective logic-domain software representation, GPlag achieved much better results. However, compared with our approach, GPlag still made several mistakes because it excluded the PDGs smaller than a certain size. Moreover, it also used a lossy filter in order to improve the efficiency.

In the second experiment, we first applied totally 976 different changes on `Inter` and dispersedly inserted its subroutines into `Driver` to get the target program. Although the number of changes looks like very big, most of them were achieved very easily. For example, we ran a small program to automatically insert an unrelated statement after every line for a certain part of codes. By comparing the original `Inter` and the much larger and sufficiently

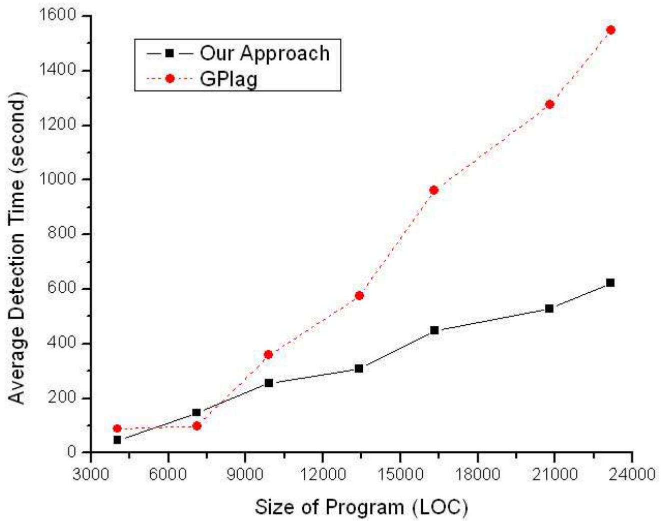


Figure 3. Speed comparison of GPlag and our approach on program `Commu` with different lines of code (LOC)

altered target program, we want to evaluate the detection ability of the four approaches. In Table 3, we show the experimental results. As we can see, due to the intrinsic limitations of token-based methods, both JPlag and MOSS got very high false negative rate (FNR) and generated a lot of false positives (FP). GPlag greatly improved the performance in terms of much lower false negative rate and less false positives. Compared to GPlag, our approach achieved even better results because it did not discriminate the small size PDGs and avoided using a lossy filter.

4.2. Speed analysis

In this section, we present a quantitative speed analysis of the proposed approach compared with GPlag. The program `Commu` totally has 291 different subroutines. Each time we took a part of the whole subroutines as the original program and then randomly made some modifications to generate a target program. Fig. 3 shows the average detection time of GPlag and our approach in different cases. Considering that GPlag has to set several parameters of the lossy filter to get a tradeoff between accuracy and efficiency, we experientially selected these parameters under a condition that GPlag could achieve comparable detection results with our approach. As we can see, when the program's size is small, GPlag may achieve slightly fast performance than our approach because our method pays additional computational cost for space-domain analysis. However, when the program's size increases, the speed of our approach becomes much faster than GPlag. This is because the computational complexity of fingerprinting based space-domain analysis is much lower than that of graph matching in logic-domain. Although our approach also exploits PDG-based graph matching, it is only used for the seed graphs. Thus the computational cost is much less than that of GPlag.

5. Conclusions

In this paper, we have presented a software reuse detection framework based on a novel integrated space-logic domain model. Compared with the common practice of using a token-based space-domain model, the proposed approach exploits logic-domain information and solves the problems of complicated modifications such as statement reordering, insertion and control replacement. Compared with the methods using logic-domain models, our approach reduces the high computational complexity and thus greatly improves the algorithm's efficiency. Preliminary experimental results have shown the superior performances of the proposed approach compared with existing methods for software reuse detection.

Several important issues of the proposed joint space-logic domain model remain unsolved and will be the subjects of future research: (1) The current framework compares the similarity between the original program and the target program directly without doing duplication analysis within each program. However, in our experiments, we found that many programs themselves have a lot of "redundancy", where codes are repeated again and again. Therefore, if we can do software reuse analysis for each program at first, it will greatly decrease the inter-program searching complexity and thus improve the running speed of the software reuse detection; (2) In the current implementation of the proposed approach, PDG nodes are reordered only based on the line location without considering file information, a hierarchical structure including such information may expediate the searching speed and improve the detection accuracy; (3) The joint space-logic model is not limited to software reuse detection but can be applied for other software analysis applications. For example, we plan to integrate the proposed approach with pattern recognition techniques for software pattern extraction.

References

[1] B. S. Baker. On finding duplication and near duplication in large software systems. In *Proc. 2nd Working Conf. on Reverse Engineering*, 1995.

[2] I. D. Baxter, A. Yahin, L. Moura, Santapos, M. Anna, and L. Bier. Clone detection using abstract syntax trees. In *Proc. International Conference on Software Maintenance*, pages 368–377, 1998.

[3] X. Chen, B. Francia, M. Li, B. Mckinnon, and A. Seker. Shared information and program plagiarism detection. *IEEE Trans. on Information Theory*, 50(7):1545 – 1551, July 2004.

[4] L. P. Cordella, P. Foggia, C. Sansone, and M. Vento. Performance evaluation of the VF graph matching algorithm. In *Proc. 10th ICIAP IEEE Computer Society Press*, pages 1172–1177, 1999.

[5] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 9(3):319–349, Oct. 1987.

[6] M. Garey and D. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman Co., 1979.

[7] D. Gitchell and N. Tran. A utility for detecting similarity in computer programs. In *Proc. 30th ACM Special Interest Group on Computer Science Education Tech. Symp.*, pages 266–270, New Orleans, LA, 1998.

[8] T. Kamiya, S. Kusumoto, and K. Inoue. Ccfinder: a multi-linguistic token-based code clone detection system for large scale source code. *IEEE Trans. on Software Engnieering*, 28(7):654–670, July 2002.

[9] J. Krinke. Identifying similar code with program dependence graphs. In *Proc. Eigth Working Conference on Reverse Engineering*, 2001.

[10] C. Liu, C. Chen, J. Han, and P. S. Yu. GPLAG: Detection of software plagiarism by program dependence graph analysis. In *Proc. 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 872–881, Philadelphia, PA, Aug. 2006.

[11] L. Prechelt, G. Malpohl, and M. Philippsen. Finding plagiarisms among a set of program with jplag. *Journal of Universal Computer Sciences*, 8(11), 2002.

[12] S. Schleimer, D. Wilkerson, and A. Aiken. Winnowing: Local algorithms for document fingerprinting. In *Proc. the ACM SIGMOD International Conference on Management of Data*, pages 76–85, June 2003.

[13] J. R. Ullmann. An algorithm for subgraph isomorphism. *Journal of the Association for Computing Machinery*, 23:31–42, 1976.

[14] M. J. Wise. YAP3: Improved detection of similarities in computer program and other texts. In *Proc. 27th SIGCSE technical symposium on Computer science education*, pages 130–134, Philadelphia, PA, 1996.