Skip Lists, Jump Lists, and Other Randomized Data Structures for Maintaining Ordered Sets

Bairong Lei, Michael Wexler

University of Waterloo, David R. Cheriton School of Computer Science

Abstract. In this project, we will discuss several probabalistic data structures for maintaining ordered sets. These data structures use probabalistic methods to keep balanced, which helps to avoid skewed distributions that can lead to excessive running times.

1 Introduction

The need for ordered sets is very common in computer science. By ordered set, we mean a typical set data structure which has the additional property of maintaining the order of its elements. Programmers often need such a structure in order to determine if a given element is in the set, and also to find the nearest predecessor for a given element. Given the functionality of an ordered set, it can be extended so that every key maps to a certain value. This would essentially be a dictionary data structure. One would hope that the most common dictionary operations would take O(log n), or better yet, O(1) time.

There has been much work in developing set data structures that provide O(log n) lookup time. Examples of these are hash-tables, AVL trees, and red-black trees. In addition to quick lookup times, AVL trees and red-black trees have the additional property of being able to provide O(log n) time for the next-predecessor operation, which, for a given key, will give the smallest key that is greater than that key.

For the afforementioned dictionary data structures, there is often the occurence where

For instance, for a given n, if we were to insert the sequence 1 2 ... n into a normal binary search tree, we would end up having a very skewed structure, which would could take O(n) time to find an element. This is the reason AVL and red-black trees were developed. They use deterministic methods to perform self-balancing, so the skewed distribution never occurs.

The afforementioned data structures in their classical implementations all use deterministic methods for organizing their data. Oftentimes, one wants to put a certain element of randomness into their dictionary data structures in order to avoid situations where their data structure becomes skewed. In this paper we will discuss various data structures that use randomness in order to maintain a level of balance, to avoid skewed distributions.

This paper is organized as follows: section 2 will give an overview on several probabalistic set data structures. Section **??** will give an overview on some experiments we performed in order to compare these data structures.

The last section draws a conclusion of our work.

All test files and program code we describe in this report can be found in our GitHub repository `https://github.com/wexlermi/cs840project`.

2 Randomized data structures

2.1 Treaps

The first randomized data structure we will talk about is the treap. Treaps are at their foundation binary-search-trees, with a certain amount of randomness that allows them to remain balanced in the average case. Every element in a

treap has the binary-search-tree property, such that all the nodes in its right subtree are greater than the root, and all the nodes in its left subtree are less than the root. However, every node in a treap contains a priority value, in which the priority of a treap node is always higher than that of its children. When we insert a new element into the treap, we randomly assign it a priority, which is a random integer from a sufficiently large range to ensure there are no collisions. Then we put the element in the treap, in the place it would go if it were considered only as a binary search tree. Once it is put in its proper place in the treap, a sequence of rotations are performed to restore the priority requirement, while still maintaining proper place to make sure its key is in the proper order.

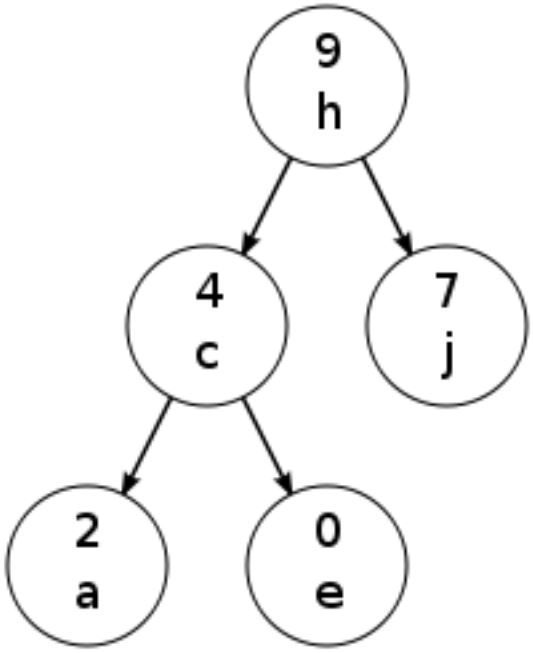

Fig. 1. Example of a treap from http://en.wikipedia.org/wiki/File:TreapAlphaKey.svg

Figure 2 is an example of a very small treap. The letters represent keys, and have the in-order property one would expect in a binary search tree. The numbers represent the priorities, in which case we have a max-heap, since each node has a priority greater than its children.

2.2 Skip lists

Skip lists are a randomized data structure that is inherently built upon ordinary linked-lists. Imagine we had a linked list of n numbers, in sorted order. If we had to search for an element x in this linked-list, it would be quite inefficient to linearly search for it. Instead, we can put pointers every other element, or every 3rd element, etc. This way, when searching for an element, we can first search through the top level, and once we find an element greater than it, we can drop down and search our original linked list.

We can repeat the process for the top linked list: put pointers to every other element. We keep repeating this process, where each higher level has half the pointers of the level below it, until we get a level which has 1 or 2

pointers. Then when we search for an element x, we start at the top list, skipping over elements that are less, then dropping down to lower and lower levels, repeating the process, until we either find the element we are looking for, or reach the bottom level and overshoot it, in which case we determine that the element does not exist in the skip list.

While the above description of a linked-list seems good at first, we reach a problem for insertion and deletion, in which case the length of the levels no longer have length 2^i. Instead, we decide to use probabalistic methods to keep our levels approximately of length 2^i. We specify a varible p which represents the probability that an element on level i will appear on level $i+1$. The typical value is $p = \frac{1}{2}$. Every time we would like to insert a key x into the skip-list, we first put it at level 0. Then, we "flip a coin" a certain amount of times, and while we are getting heads, we keep adding the element to higher and higher levels, until we get a tails, and then we stop. This will mathematically ensure that our skip-list has approximately half the number of nodes at higher and higher levels, and provides us with good performance for insert and delete operations, something that is quite hard to maintain in deterministic skip-lists.

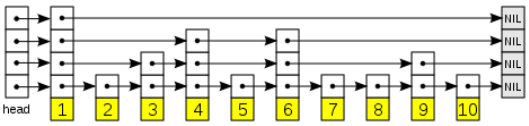

Fig. 2. Example of a skip-list from cite2

cite2: `http://en.wikipedia.org/wiki/Skip_list`

2.3 Skip lifts

In [PBM11] Bose, Doueb, Morin talk about an improvement to the skip list called the skip lift.

The main steps are:

1. Including the `crest.h` header file of the CREST project. (Line 1)
2. For all variables that appear in branching nodes of the code (in this case the variables x, y, z and w), one has to instruct CREST to collect symbolic conditions with respect to those variables during the execution. (Lines 10-13)
3. OPTIONAL: Add a `printf` to every line inside conditional branch that outputs the variable values (Line 17). Then, during the execution of CREST, whenever a solution is found that enters the desired line of code (in this case line 18), an output is provided stating the calculated values for the variables.

Remark 1. This example already contains nonlinear constraints inside the `if`-statements and therefore serves as an example of how CREST-Z3 extends the functionality of the classical CREST.

Remark 2. The datatypes CREST can handle in the current status of the project are: (unsigned) characters, (unsigned) integers, (unsigned) shorts.

Then, after compiling the CREST project according to the instructions, one runs inside the `bin` folder of the project the program `crestc` with the instructed file as input. This will generate the control flow graph and will collect initial information about the program. For this code analysis step, the developers of CREST utilize a tool called.

As a result, in the same folder as the instructed file, several other files will appear, including the compiled version of the program. Those files are provided by CIL and one can find the control flow graph and general information about the branching inside the program there.

After that, in order to obtain possible values for the observed variables for maximal code coverage, one has to run the tool `run_crest` with the following inputs:

– The executable file that was compiled by `crestc`
– A number of maximal iterations through the control flow graph we want to allow (i.e. a limitation to the paths we want to consider. Otherwise the program might run too long)
– A strategy CREST should use to traverse the control flow graph. Possible options here are in the set

{`dfs`, `cfg`, `random`, `uniform_random`, `random_input`, `cfg_baseline`, `hybrid`}

For details on the respective search strategies consider

An example of this would be:

```
$ ../bin/run_crest ./hello 10 -dfs
```

Consequently, in the same folder some new files do appear. One class of files have a name of the form `input`*, where one can find possible input values for the variables even if one has not added a `printf` line as in the code above (Line 17). Another file is called `coverage` and contains the nodes in the control flow graph indicated by numbers that are covered. Additionally, one can find a file called `szd_execution`, where the intermediate symbolic formulas that appeared during the execution of `run_crest` can be found.

Remark 3. The `input` files are only added by default in CREST-Z3, but not in the original CREST. In order to instrument CREST to do the same, one can follow the instructions given here: `https://code.google.com/p/crest/wiki/FAQ`.

For the code given in Example **??**, one of the `input`-files has the following content:

```
-1
1
1
1409925863
```

Thus $x := -1, y := 1, z := 1$ and $w := 1409925863$ would be possible inputs to reach the code lines 17 and 18.

Remark 4. That w has such a large value is due to the fact that it does not matter what we put for w as input, as it is redefined in line 15 dependent on x, y and z. Therefore, it did not play a role in any further symbolic evaluation and the standard test value that is chosen randomly for the first run is kept.

2.4 Limitations of CREST and Possible Workarounds

We have created several small test cases to check what CREST can and what it cannot – or just partially – do. In this section, we will only discuss limitations that are independent from the solving technique used. The theoretical and practical limitations of the solving routine is the subject of section 3.

Constraints in handling the unary negative operator. During the course of our evaluation of CREST, we discovered that it is unable to handle the negative unary operator. For instance, the following code sample was unable to be solved by CREST:

Example 1.

```
if ( x*x*x == -8){
    printf("GOAL %d\n", x);
    return 1;
}
```

Surprisingly, a slight alternation of the code by replacing `x*x*x == -8` by `x*x*x+8==0` makes it possible to be solved. We suspect that there might be some bug in the generation of the formula, as we checked by hand if Z3 is able to solve this formula and encountered – not surprisingly considering the reputation of Z3 – a positive answer.

Interprocedural Analysis Possible... Or Not? We discovered a reasonable flaw concerning interprocedural analysis.

Consider Example **??** again, and assume we replace line 14 by

```
if (f(x,y,z))
```

where `f` is the following function:

```
f(int x, int y, int z){return x*y+z==0;}
```

Using this instruction, CREST fails to produce possible outputs to enter the branches induced by the modified `if` statement.

However, let us replace line 14 is instead by

```
if (f_2(x,y,z)==0)
```

where `f_2` is the function

```
f_2(int x, int y, int z){return x*y+z;}
```

Then everything works well and CREST is able to state and solve the necessary equations to branch into the body of that `if`-statement.

But what went wrong? Studying the file `szd_execution`, it appears that it failed to produce the equations that need to be solved. We suspect that this is due to the fact that during the analysis process, CREST is not able to set conditions under which conditions the boolean expression `x*y+z == 0` itself makes the `if` statement evaluate to be true. If it is directly inside the `if`-statement, it is clear to CREST how to evaluate it. But in the case of `f`, it fails to reason that `f` evaluates to `1`, the C equivalence to `true`, exactly when `x*y+z == 0`. This can be seen as an issue of the analysis process, and we will post it right after this project is handed in.

Double resp. Float Variables. We mentioned above (Remark 2) the data types supported by CREST. What is missing are datatypes like `float` or `double`.

In fact, as we will see in Section 4, for a large set of implementations where nonlinear equations play a role inside `if` statements have floating point variables that determine the evaluation.

For the original CREST, there exists a patch to extend its functionality to be able to also handle floating point variables (`https://code.google.com/p/crest/`, Issue #15). Unfortunately, that patch cannot analogously be applied to CREST-Z3. We discovered that in order to obtain this functionality, we would need to refactor large parts of the core of CREST-Z3, and this would go above and beyond the scope of our project.

Arrays. Even though both Z3 as well as YICES are capable of handling array types, CREST does not support that as an observable type. This can be seen as a flaw, even though there is a workaround to this: In the original CREST paper (), the authors were testing their software on the open source program GREP with a very high coverage rate (60% of the estimated coverable branches).

In fact, one variable they were observing in the project was `keys`, which is in fact an array. In order to instrument CREST to do that, they put every entry of the array into the scope of CREST in the following way:

```
[...]
keys = argv[optind++];
  keycc = strlen(keys);
```

```
  for(iii=0; iii<keycc;iii++){
    CREST_char(tmp1);
    keys[iii]=tmp1;
  }
  printf("%s\n",keys);
[...]
```

This means that CREST adds, in every iteration of the `for`-loop, `tmp1` as a new variable. From that moment on, it knows that keys at certain position store an observed value and then it can start performing evaluations based on that.

No Automatic Instrumentation. For CREST, we need to manually instrument code in order to clarify which variables it should observe. Depending on the size of the project, this can be a very time-consuming task. Furthermore, it requires the program to be executable to work, i.e. one cannot utilize it for pure in-between test generation for units, but either has to run it after the project has reached the executable state, or one has to write in-between executable routines for the units. Both solutions are not optimal in general.

To solve this issue, we have written a script that automates this instrumentation process. It can be found in the GitHub repository of our project (`src/crest_instrument.py`).

3 Z3 And the Problem of Solving Nonlinear Equations

This section is dedicated especially to Z3 and some theory around solving nonlinear equations.

3.1 Introduction to Z3

The tool Z3 was developed by researchers from Microsoft to decide – if possible – "Satisfiability Modulo Theories" (abbreviated SMT) problems. These problems are given in first order logic and can be seen as an extension to the satisfiability problem in predicate logic (SAT), by introducing datatypes like numbers and their respective operations. A standard to represent those formulas in an automatically readable form was developed in the SMT-LIB project . The developers of Z3 are using this standard as their internal representation.

We will refrain from providing a detailed description of SMT problems and their representation, as this would exceed the scope of this report. But the following example will present the basic structure of those problems:

Example 2. Given the variables x, y, z, which we consider to be integer numbers, an SMT solver could answer the question of whether or not a solution for x, y and z exists, given the following constraints:

1. $x \neq y$
2. $xy + x^2y = z$
3. $|z| < 10$.

The solver would either state that it cannot solve this problem, or that no solution exists (provable), or it would provide values for x, y and z that satisfy the given conditions.

Z3 can be included as a library to C/++ source code, and one can call the theorem prover from there. This is very handy, as one does not have to parse any input to a tool and parse the output back, but instead can just can call the respective functions inside his own functions.

The source code of Z3 was recently (September 2012) published by Microsoft (`http://z3.codeplex.com/`), so we could examine the techniques the developers used to handle the nonlinear equations we are looking for inside their code. We will go into more detail on that after we discuss some theory behind nonlinear equations.

3.2 Theoretical Limitations

There are some theoretical limitations given considering the problem of solving nonlinear equations.

We are given the following scenario: consider integer variables $x_1, \ldots, x_n, n \in \mathbb{N}$, and a boolean expression inside an if statement that evaluates to true if and only if a polynomial given in those variables is zero for some choice of $x_1, \ldots, x_n$. Finding a solution for the x_i is the so-called "Hilbert's tenth problem"

Despite the intractability of this problem, one can always try to find solutions and cancel the computations after a certain amount of time.

Given a set of polynomial equations, the status quo to find solutions is to first calculate the Gröbner basis of this system (originated by B. Buchberger in his PhD Thesis, consider e.g. in . This can be seen as a simplification of the system, which leads in the case of the existence of only finitely many solutions directly to the solutions. Inside the source code of Z3, we could also find an implementation of an algorithm to calculate those bases.

But there is one problem to this approach: calculating a Gröbner basis has double exponential complexity, independent of the choice of the algorithm chosen to compute it . Therefore, computations can take a long time, and the output may be quite simple, but is still hard to utilize it for finding the solutions. The next subsection will provide an example to illustrate this fact.

Finding solutions to nonlinear equations is also heavily studied in current research. Almost every computer algebra system provides a functionality to solve systems of equations, but there are still plenty of examples where those solvers do fail. Thus, one is also limited by the possibilities current research provides.

3.3 Practically Hard Problems

In order to illustrate where one reaches those limitations, we considered the following code:

```
int main(int argc, char *argv[]){
  int w;
  int x;
  int y;
  int z;
  if (x*x*y*y*y*z*z == 2700){
    if (x*x*y*w + y*y*w*w == 525){
      if ( x * y*w+y*y*y*w*w==1365){
        printf("GOAL %d %d %d %d\n", x, y, z, w);
          return 1;
        }
     }
  }
  return 0;
}
```

In order to cover all branches, one could use as input $(x, y, z, w) := (2, 3, 5, 7)$. Z3 is not able to solve it. We stopped the calculation after a couple of hours.

Let us examine the reason why it fails. Z3 will try to compute a Gröbner basis of the ideal generated by

$$x^2y^3z^2 - 2700, \quad x^2yw + y^2w^2 - 525, \quad xyw + y^3w^2,$$

and based on that result, it will try to solve the equation. As said, solving for integer values is, in general, undecidable.

Let us look at the Gröbner basis for this system, which we calculated using the computer algebra system SINGULAR The output is a polynomial system that fits into a file of the size of 283KB. We will give a short snapshot of one element in the output for the sake of visualization:

```
6755988021120000yw31-1659946256789184000000yw30
+1737133233041462256000000yw29
-1010330953087128944544000000yw28
+3574401248043641406179880000000yw27
-7914047652075802157021883600000000yw26
+1087259303325678451222704893100000000000yw25
-8876099505180942301261003813440000000000000yw24
+39085314130645312603256453079810000000000000000yw23
-71160249389966711159864135184245700000000000000yw22
+[...]
```

SINGULAR is specialized in performing those computations, and it took several minutes to compute this Gröbner basis. There are several factors that play a role how fast an algorithm computes this Gröbner basis. One of them is how "smart" it is implemented. This computation can take either minutes or hours, depending on the implementation.

Of course, one can also try to solve those equations directly using SINGULAR or other state-of-the-art computer algebra systems. We observed that those will not provide the easy solution that we know about this system.

The computer algebra system MAPLE e.g. outputs the following:

```
{w = w,
x = RootOf(13*w*_Z^6-5*w*_Z^5+13*w^2*_Z^3+(-5*w^2+20475*w)
    *_Z^2-5250*w*_Z-1378125+17745*w^2),
y = -(13*RootOf(13*w*_Z^6-5*w*_Z^5+13*w^2*_Z^3
    +(-5*w^2+20475*w)*_Z^2-5250*w*_Z-1378125+17745*w^2)^2*w[...],
z = 338*RootOf(-15138703125*w^6+61992127968750*w^5
    -12932841796875*w^4-4814544287109375*w^3[...]}
```

This means that MAPLE recognizes that the value of w can be chosen arbitrarily and that the solution solely depends on the values of x, y and z. Furthermore, the solution of MAPLE depends on algebraic extensions of the ground field (usually chosen to be $\mathbb{Q}$).

The computer algebra system SAGE , which utilizes MAXIMA for solving such equations, fails to output any solution whatsoever.

The last computer algebra system that we tested for those purposes was MATHEMATICA, in the context of WOLFRAM ALPHA
(`http://www.wolframalpha.com`). It found several solutions, but not ones with integer values. One solution was, for example:

```
w<-(25 sqrt(21))/13 and
x = Root[13 #1^6 w-5 #1^5 w+13 #1^3 w^2+#1^2 (20475 w-5 w^2)-5250 #1
    w+17745 w^2-1378125&, 1]
and y = 1/2 (sqrt((x^4+2100)/w^2)-x^2/w)
and z = -30 sqrt(3) sqrt(1/(x^2 y^3))
```

As a conclusion, one can see that deriving an integer solution using external tools appears to be a very daunting task.

3.4 Alternatives

Z3 is of course not the only SMT-solver that is available, yet it belongs to the group of the most powerful ones. Other tools that are capable of solving complex SMT's and that use the same input as Z3 are, for instance:

- Alt-Ergo and
- CVC4.

In the original CREST project, YICES was used for solving. It was not capable of nonlinear constraints, but accepted input in the SMT-LIB standard. Therefore, replacing the SMT-solver in the CREST project took only the instrumentation effort, which can also be done similarly with the tools above. This would lead to an interesting comparison between those tools regarding the branches in code they are able to resolve.

Another approach that is, in our opinion, much more preferable would be to utilize computer algebra systems such as the ones mentioned in the last subsection, which in general contain the implementations of the latest research techniques to solve equations of all kinds. There is, of course, more instrumentation to be done in order to get them to work within the CREST context, but the extra effort will be paid back by lower calculation times plus more robust results.

4 Experiments And Case Study

In this section, we will discuss some experiments on existing software projects done with CREST.

We will start by summarizing evaluations done in

After that, we will discuss project types where evaluations of nonlinear equations are likely to be found in C/++ programs, and how common the use of nonlinear equations within boolean expressions is.

We will conclude this section by evaluating CREST on a set of test cases we generated, and we will examine some key metrics that give insight into the accuracy and performance of CREST.

4.1 Summary of Former Evaluations

The author of has tested the original program on three different source codes:

- Vim , $\sim 150,000$locl
- Grep (`http://www.gnu.org/software/grep/`), $\sim 15,000$loc
- Replace , ~ 600loc

He was able to cover over 90% of Replace within some minutes. Around 60% of the branches in Grep were covered within a couple of minutes, where one could partly make improvements using different search strategies. For Vim, he only instructed CREST to analyze some modes of the editor, with a sample input. Within 2-3 hours he was able to cover up to one third of all branches that were reachable.

In the paper about CREST-Z3 , one can only find simple proof of concepts that it extends the classical CREST. A large test on real world programs has not been made.

4.2 The Frequency of Nonlinear Equations Appear in Real World programs

Ad Real World Programs According to the Limitations. First, we were searching for `if`-statements in programs that contain nonlinear expressions in the supported types. Our search was not fruitful, and only brought up a few examples that contained the desired property.

Considering the CRYPTO++ library (`http://www.cryptopp.com/`, a C++ library containing all kinds of algorithms from the field of cryptography), we could only find a few, fairly simple tests of the following format:

```
<<rsa.cpp>>
[...]
[168] Integer a = modn.Exponentiate(i, r);
[169] if (a == 1)
[170]     continue;
[...]
```

Those examples are not very hard to solve.

Looking at the source code of the computer algebra systems SINGULAR and GAP , where we assumed to find examples, revealed that there is no occurrence of boolean expressions consisting of nonlinear expressions to be found in the code.

Our conclusion is that nonlinear constraints with the standard supported datatypes in CREST are very rare to be found in real life C code.

Ad Real World Programs outside the Limitations. If we assume that floating point numbers would also be supported, the range of real world programs that contain boolean expression based on nonlinear equations is much higher.

The first type of programs where those types of equations do appear are coming from the field of computer graphics. In the technique called "ray tracing" for example, they are very common. The following code snippet serves as an example in what way one finds them there (taken from `http://www.codermind.com/articles/Raytracer-in-C++-Part-I-First-rays.html`):

```
[...]
[1] double B = r.dir * dist;
[2] double D = B*B - dist * dist + s.size * s.size;
[3] if (D < 0.0f)
[4]     return false;
[5] double t0 = B - sqrt(D);
[6] double t1 = B + sqrt(D);
[7] bool retvalue = false;
[9] if ((t0 > 0.1f) && (t0 < t))
[...]
```

In line 2, the variable `D` is calculated using a nonlinear polynomial formula. In the next line it is checked whether the result is smaller than zero or not. This code snippet furthermore shows an inconvenience we would obtain if we would allow floating point numbers, namely the square root use in line 5 and 6 and the boolean expression based on it in line 9. There are publications on ways to define square root and other floating point typical operations that extend SMT-LIB, e.g. , and the basic preliminaries are included in the SMT-LIB version 2.0. Nonetheless, we could not

find the full support of the arithmetic in the solvers. If one would extend CREST-Z3 to also handle floating point numbers, this problem should also be taken into consideration.

Another area where we found boolean expressions based on nonlinear equations in floating point numbers were algorithms coming from finance. There is an interesting collection of C++ programs in. The way they appear there mostly is in converging constraints and in early termination of a program if a certain formula is fulfilled. The same holds for algorithms coming from numerics; consider for example the project LAMMPS (Mollecular Dynamics Simulator,

Summa summarum one can say that considering nonlinear constraints, in practice it is only reasonable to additionally consider them if also floating point numbers are supported. For the supported datatypes, one rarely finds code where those play a significant, non-trivial role.

4.3 Some Empirical Experiments on the Solving Ability of CREST-Z3

We evaluated CREST for both performance and accuracy. To measure performance, we measured the time it took for CREST to analyze a given program. To measure accuracy, we measured the branch coverage ratio, which is the number of branches CREST reaches, divided by the total number of program branches.

Due to the fact that CREST cannot handle the unary minus operator, which is prevalent in most real-world programs, we had to create our own test set of programs to evaluate on CREST. We wrote a test-set generator createTestFiles.py that, given several parameters, creates a random C file containing a certain amount nested if-statements, each of which contain an equation, as part of a total system of equations. For the equations in our test programs, we specified the total degree for any individual monomial to be 2, and the variables to range from 0 to 10. Future analysis could also change these variables, and see how the dependent variables change.

Here we present a sample C program that we generated, specified to have 3 variables, and a nested-depth of 2 :

```
#include <stdio.h>
int main() {
int x0;
int x1;
int x2;
    if ( 9 * x0 * x0 + 4 * x2 * x2 + 8 * x0 * x1 == 500 ) {
        printf("Solved the if at depth 1.\n");
        if ( 9 * x0 * x1 + 8 * x2 * x2 + 6 * x1 * x2 == 728 ) {
            printf("Solved the if at depth 2.\n");
        }
        else {
            printf("At the else at depth 2.\n");
        }
    }
    else {
        printf("At the else at depth 1.\n");
    }
}
```

We were interested in exploring the effect on program timing and coverage caused by changing several independent variables. The independent variables we analyzed were number-of-variables, nested-depth, and search-type. By search-type, we mean the type of search that CREST uses to traverse the program's control-flow-graph. For testing purposes, this can be one of the following three: DFS, random, and hybrid.

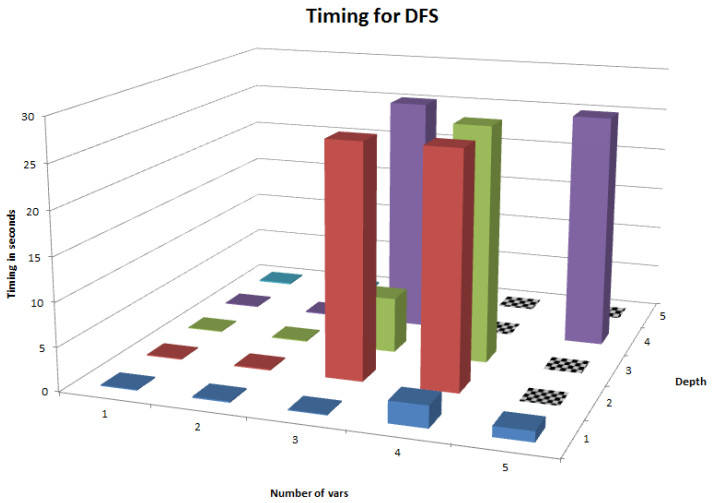

Fig. 3.

During the course of our analysis, we realized that CREST would often take a long time to analyze programs of large depth or large number of variables. For this reason, in our test script, we decided to make CREST timeout when the analysis took longer than 40 seconds.

Using the aforementioned script, we generated a test-set of 75 C files, which consists of 3 unique C files for each tuple (number-variables, nested-depth), where number-variables and nested-depth range from 1 to 5.

Table 1 represents the timings when running CREST on our test-set using DFS (depth-first-search) search-mode.

Table 1. DFS timings (in seconds)

1	0.131	0.163	0.112	2.482	1.19
2	0.129	0.147	26.738	26.751	—
3	0.111	0.133	6.354	27.138	—
4	0.124	0.135	27.068	—	26.789
5	0.167	0.244	1.393	—	—

Figure 1 gives a graphical view of Table 1. The checkered squares of Figure 1 represent tests where the timing went past our threshold of 40 seconds. When we tested several of these programs by hand, we waited for an hour or so, and CREST still did not finish analyzing it. The results seem to confirm our guess that CREST runtime increases as the number of variables and the number of nested if-statements increase. We were a bit disconcerted to see that several of our test cases were unable to be solved by CREST in a feasible amount of time. Our previous discussion already mentioned some of the performance issues of non-linear equation solving.

Figure 2/Table 2 show the results of measuring branch coverage for the DFS run-mode. By branch coverage, we mean the number of branches in the control-flow-graph that CREST was able to get to. Since every if-statement has a corresponding else-statement in our test set, the number of total branches for a given program is twice the depth. We

Table 2. DFS branch-coverage-ratio

1	1	1	1	1	1
2	0.75	0.75	0.25	0.25	0
3	0.667	0.667	0.667	0.333	0
4	0.625	0.75	0.25	0	0
5	0.6	0.7	0.6	0	0

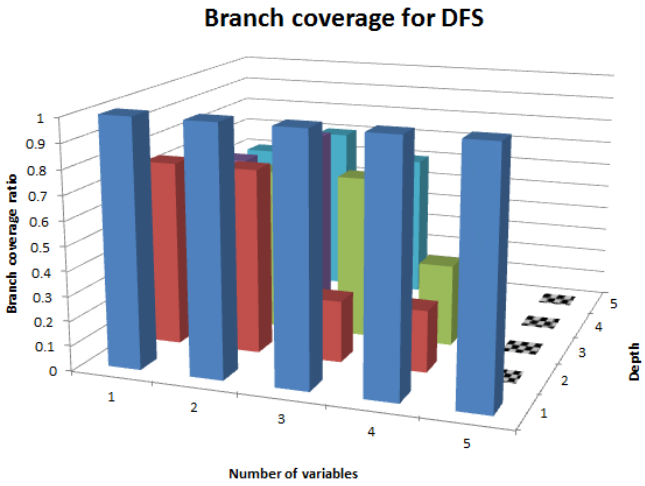

Fig. 4.

found that as the number of variables in a program increased, the branch-coverage-ratio decreased, because it made it more difficult for Z3 to solve the constraints. Additionally, as we increase the program depth by one, the number of total branches increases by 2. This lowers the branch-coverage-ratio since CREST now has more branches to solve.

Table 3. Results for running CREST for depth = number-variables = 3

	DFS	Random	Hybrid
Runtime (seconds)	6.354	13.572	13.49
Branch-coverage-ratio	0.667	0.222	0.5

Table 3 summarizes the performance and accuracy of CREST on the test-files corresponding to depth = number-variables = 3. We find that DFS gives the best performance and accuracy of the three run-modes, while random gives the worst results.

5 Our Improvements to CREST

We accomplished several tasks that we believe will make CREST a more usable tool. We created a web interface for CREST, located at `http://dahu.in/crest.html`. At this web interface, one can submit a C file of his choosing, and examine the output that CREST would generate. We believe that this makes it much easier for one to test out CREST, without having to actually go to the trouble of downloading and installing it. From our own experience, the installation of CREST is quite a painful one, as the documentation on the CREST website does not cover many crucial steps that we discovered only through trial and error.

The second improvement that we made to CREST (as mentioned previously) is the development of a Python script that automatically instruments C code so it can work with CREST. CREST has the unfortunate property that all C files must explicitly declare their variables in a CREST format. Our script automatically instruments C code so this step is not necessary. In fact, our script recursively instruments all C files in a given directory, so an entire project can be instrumented for CREST very easily.

6 Conclusion and Future Work

There are several conclusions we can draw in this report.

The first one is that CREST(-Z3) as a tool has to be made more robust. Some of the limitations we have found (like the one connected to interprocedural analysis), are flaws coming directly from the analysis process.

In order to really extend the functionality of CREST for real world programs, one also has to consider floating point numbers as possible data types, as for those nonlinear equations in boolean are more likely to appear. For the supported types, the original CREST suffices on our set of real world programs.

Concerning the question of which problem solver to use, we have seen that there are limitations from theory. Nevertheless, computer algebra systems contain the most recent algorithms coming from mathematicians all over the world. The solving part could be partially outsourced instead of using solvers implemented in Z3 or other software projects. This would in our opinion increase the quality of those solvers in terms of equations they are able to solve.

A possible future task could be to deal with the constraints of CREST(-Z3). Resolving them is not trivial. Furthermore, one could make the solving process more flexible, i.e. the solver replacable without any instrumentation, and possibly having several by hand at once.

Another future task is related to the question whether we are content with just node coverage on the control flow graph or not. CREST-Z3 is performing only node coverage, and maybe it would be interesting to also perform some criteria like prime path coverage (see). As with considering paths, we also gain more constraints on our variables, it might lead to an easier way to solve for those variables or prove the infeasibility. It would lead to a more flexible way of test generation that would be of interest for academia as well as for industry.

References

[PBM11] Karim Doueb Prosenjit Bose and Pat Morin. Skip lifts: A probabilistic alternative to red-black trees. *Combinatorial Algorithms: Lecture Notes in Computer Science*, Volume 6460:226–237, 2011.