Finding Minimal Inconsistent Subsets

Solaman Huq

April 13th, 2015

1 Introduction

There has been a fair amount of work in the field of System Diagnostics, and in an attempt to reasonably analyze faulty systems, many algorithms have been created to find what we call Minimal Inconsistent Subsets. However, for many algorithms, little work has been done in order to benchmark and compare them. In this paper, we will briefly explain the implementation of a few of these algorithms: Top-Down, Bottom-Up, Random, And Hitting Set Trees, as well as an explanation for our approach to benchmarking these algorithms as well as the results of said benchmarking.

2 Definitions

Before going any further, it is important to know the definition of a Minimal Inconsistent Subset. While this may be obvious to some, a fair explanation is useful for those not familiar with the subject! Suppose we are given a set description, which in most practical applications is a Boolean circuit. Suppose that an instance of this circuit is not functioning properly, and we wish to find an explanation why. This explanation is a subset of constraints, or in this case logic gates within the circuit, such that when we claim the subset's logic gates are "normal" and leave them alone, and claim that all other logic gates are "abnormal" and force their output to an ideal, the final output is inconsistent with a normal output. Hence, we call this subset "inconsistent". Clearly then a Minimal Inconsistent Subset is an inconsistent subset such that removing any one item makes the subset consistent.

3 Algorithm Implementations

All Python implementations for this study can be found at the following link:

github.com/solaman/System-Diagnostic-Analysis/tree/master/Dedekind/Algorithms/

3.1 Hitting Set Trees

There is already documentation on the subject of hitting set trees, so for those interested in seeing the original paper, I have provided a link:

taxonbytes.org/pdf/ChenEtAl2014-HybridDiagnosisApproach.pdf

Otherwise, I am providing pseudo-code for the sake of uniformity. The main algorithm calls upon Logarathmic Extraction, whose job is to find a minimal inconsistent subset. Hitting Set Tree then records all previous MIS in a way to reduce how many subsets *it* has visited (but not Logarithmic Extraction).

Algorithm 1 Hitting Set Tree

```
1: function ComputeAllMIS(setDescription, constraints)
2:     foundMIS, currPath, allPaths ← ∅
3:     return ComputeAllMIS(setDescription, constraints, foundMIS, currPath, allPaths)
4:
5: function ComputeAllMIS(setDescription, currConstraints, foundMIS, currPath, allPaths)
6:     for path ∈ allPaths do
7:         if path ⊆ currPath then
8:             return foundMIS
9:     if isConsistent(setDescription, currConstraints) then
10:        allPaths ← allpaths ∪ currPath
11:        return foundMIS
12:    tempMIS ← ∅
13:    for mis ∈ foundMIS do
14:        if mis∩currPath = ∅ then
15:            tempMIS ← mis
16:    if tempMIS = ∅ then
17:        tempMIS ← LogarithmicExtraction.ComputeSingleMIS(setDescription, currConstraints)
18:        foundMIS ← foundMIS ∪ {tempMIS}
19:    for constraint ∈ tempMIS do
20:        tempPath ← currPath ∪ {constraint}
21:        ComputeAllMIS(setDescription, currConstraints/{constraint}, foundMIS, tempPath, allPaths)
22:    return foundMIS
```

Algorithm 2 LogarthmicExtraction

```
1: function ComputeSingleMIS(setDescription, constraints)
2:     if never called for setDescription and not IsConsistent(setDescription, ∅) then
3:         return ∅
4:     return ComputeSingleMIS(setDescription, ∅, constraints)
5:
6: function ComputeSingleMIS(setDescription, currSubset, currConstraints)
7:     if |currConstraints| = 1 then
8:         return currConstraints
9:     currConstraints_L, currConstraints_R ← Split(currConstraints)
10:    if not isConsistent(currSubset ∪ currConstraints_L ) then
11:        return ComputeSingleMIS(setDescription, currSubset, currConstraints_L)
12:    if not isConsistent(currSubset ∪ currConstraints_R ) then
13:        return ComputeSingleMIS(setDescription, currSubset, currConstraints_R)
14:    currSubset_L ←
15:    ComputeSingleMIS(setDescription, currSubset∪currConstraints_R, currConstraints_L)
16:    currSubset_R ←
17:    ComputeSingleMIS(setDescription, currSubset ∪ currSubset_L, currConstraints_R)
18:    return currSubset_L ∪ currSubset_R
```

3.2 Top-Down

The idea is simple: for a given set of constraints C, the MIS exist as a member of the powerset of C. This algorithm will generate the powerset and go through the powerset in a top-down approach. i.e. for the set $\{a, b, c, d\}$ the set $\{a, b, c, d\}$ will be the first that is visited, then $\{a, b, c\}, \{a, b, d\}, \{a, c, d\}, \{c, b, d\}$, etc. When we find that a set is consistent, we know all of its children will also be consistent and we rule them out. If a set is inconsistent, then we know all of its parents are not MIS. Once we've eliminated all the subsets that we can, only the MIS will remain.

Algorithm 3 Top-Down

```
1:  function ComputeAllMIS(setDescription, constraints)
2:      remainingSubsets ← GeneratePowerset(constraints)
3:      allSets ← copy(remainingSubsets)
4:      allSets ← sort(allSets,descending)
5:      for possibleMIS ∈ allSets do
6:          if possibleMIS ∉ remainingSubsets then
7:              continue
8:          childrenSets, parentSets ← ∅
9:          for set ∈ remainingSubsets do
10:             if set ⊆ possibleMIS then
11:                 childrenSets ← childrenSets ∪ {set}
12:             if set ⊃ possibleMIS then
13:                 parentSets ← parentSets ∪ {set}
14:         if isConsistent(setDescription, possibleMIS) then
15:             remainingSubsets ← remainingSubsets/childrenSets
16:         else
17:             remainingSubsets ← remainingSubsets/parentSets
18:     return remainingSubsets
```

3.3 Bottom-Up

The implementation of Bottom-Up looks a little more straightforward than Top-Down, simply because of how we iterate through subsets. As we iterate through the powerset lattice, if a set is found to be inconsistent, then clearly all supersets are not MIS and we do not have to consider them. Using this logic, any given subset that we iterate over is an MIS if it is found to be inconsistent through induction. In Top-Down, we could not use this heuristic because we would still need to check the children of a subset before we could declare it as a MIS.

Algorithm 4 Bottom-Up

```
1: function ComputeAllMIS(setDescription, constraints)
2:     foundMIS ← ∅
3:     remainingSubsets ← GeneratePowerset(constraints)
4:     remainingSubsets ← Sort(remainingSubsets, ascending)
5:     while remainingSubsets ≠ ∅ do
6:         possibleMIS ← next(remainingSubsets)
7:         parentSubsets ← ∅
8:         for set ∈ remainingSubsets do
9:             if set ⊃ possibleMIS then
10:                parentSubsets ← parentSubsets ∪ {set}
11:        if isConsistent(setDescription, possibleMIS) then
12:            foundMIS ← foundMIS ∪ {possibleMIS}
13:            remainingSubsets ← remainingSubsets/parentSubsets
14:        remainingSubsets ← remainingSubsets/{possibleMIS}
15:    return foundMIS
```

3.4 Random

The last algorithm we implemented takes the best of both Top-Down and Bottom-Up by selecting subsets that are, on average, in the middle of the remaining powerset lattice. By iterating through the subsets in this way, we increase the probability of using some sort of elimination. If the current subset is found to be inconsistent, we can eliminate its parents in our search, and if it is found to be consistent, then we can eliminate its children in our search. In Top-Down or Bottom-Up, only one of these options was available/useful.

Algorithm 5 Random

```
1: function ComputeAllMIS(setDescription, constraints)
2:     foundMIS ← ∅
3:     remainingSubsets ← GeneratePowerset(constraints)
4:     while remainingSubsets ≠ ∅ do
5:         possibleMIS ← random(remainingSubsets)
6:         parentSubsets, childrenSubsets ← ∅
7:         for set ∈ remainingSubsets do
8:             if set ⊃ possibleMIS then
9:                 parentSubsets ← parentSubsets ∪ {set}
10:            if set ⊂ possibleMIS then
11:                childrenSubsets ← childrenSubsets ∪ {set}
12:        if possibleMIS.wasChecked and parentSubsets ∪ childrenSubsets = ∅ then
13:            remainingSubsets ← remainingSubsets/{possibleMIS}
14:            foundMIS ← foundMIS ∪ {possibleMIS}
15:        else
16:            if not isConsistent(setDescription, possibleMIS) then
17:                remainingSubsets ← remainingSubsets/parentSubsets
18:            else
19:                remainingSubsets ← remainingSubsets/childrenSubsets/{possibleMIS}
                   possibleMIS.wasChecked ← True
20:    return foundMIS
```

4 Creating Test Trials

Instead of finding systems out in the wild that were inconsistent, we wished to find a way to generate our own systems to test our algorithms on. This would enable us to find a clean and representative way to test an algorithm on all faulty systems for a given input size. The crucial fact we considered when trying to generate such faulty systems is that they could be represented by Monotone Boolean Functions (MBF's). This makes sense, certainly if you claimed that *all* constriants were working properly, you would get inconsistent output! As long as a subset *contains* an MIS, the subset itself would also be inconsistent.

Such a fact lead us to MBF generation and Dedekind Numbers. However, many of the efficient algorithms out there are more concerned with *computing* Dedekind numbers, as opposed to the MBF's they represent. With this in mind, we created our own algorithm that could generate all MBF's of input size 5 within a reasonable amount of time. An explanation of this algorithm (as well as a Python implementation) can be found at this link:

https:github.com/solaman/System-Diagnostic-Analysis/blob/master/Explanation.pdf

5 Benchmarking

With our various algorithms for calculating MIS, and our data set to run our algorithms on, all that was left was to perform the actual benchmarking! In our benchmarking, we decided to keep track of the number of times the function "isConsistent" was called. We do this because, in practice, the limiting factor is indeed the call to this function, as systems can often times be

very, very large! That being said, we tested our algorithms with input size =4, and input size = 5.

Input size = 4

Algorithm	Min	Max	Mean	Std Dev	Median	Mode
HST	1	30	16	11.70	17	11
Bottom-Up	1	16	11	5.42	11	11
Top-Down	1	16	10	4.80	11	11
Random	1	13	8	3.20	9	10

Input size = 5

Algorithm	Min	Max	Mean	Std Dev	Median	Mode
HST	1	83	42	27.71	44	46
Bottom-Up	1	32	21	8.20	20	26
Top-Down	1	32	21	8.20	20	26
Random	6	24	17	5.55	17	18

Also, since Hitting Set Tree and Random are both random algorithms, we run the experiment 50 times with input size = 5 to see how much the mean number of calls to "isConsistent" changes (we calculate the mean of means, if you will). This is our findings:

Input size = 5

Algorithm	Mean of Means	Std Dev of Means
HST	41.96	.20
Random	16.04	.28

For all algorithms that iterate through the powerset, we find that they perform an expectable number of calls to "isConsistent". More so, because Bottom-Up and Top-Down perform a rigid procedure, it makes sense that there is a particular system that they must check all subsets of in order to find all MIS.
We also find that the Random algorithm performs the best. This is most likely because it is able to take advantage of the pros of Bottom-Up as well as Top-Down, meaning that it is more likely to perform elimination after a check of "isConsistent" than either of the other two.
Lastly, we find that Hitting Set Tree requires many more calls than any of the algorithms that iterate through the powerset of constraints. This is most likely because Hitting Set Tree uses Logarithmic Extraction as a black box, and as it performs many calls to "isConsistent" before finding an MIS, much information is wasted that could be used to efficiently reduce the search space of Hitting Set Tree.

6 Moving Forward

Performing benchmarking on larger MBF's would be challenging if we attempted to generate all of them per input size as we have in this study. We would be better off if we were to generate a set of, say, 100 random MBF's per input size. Further, powersets grow at an exponential rate per input size, so it is certain that the algorithms that use powerset generation will slow down considerably as the constraint size grows. While it can't be helped completely, as the problem is definitely NP-Complete, we can still store subsets that have been visited in an implicit manner, so that the entire powerset does not need be stored in memory. Since Top-Down and Bottom-Up are consistent in their search, it would be easiest for them, but Random would prove to be a bit more tricky.
Aside that, there are always other algorithms to benchmark!