Hitting Set: Applying Stochastic Heuristic Search Methods towards Optimal Solutions

Raphael Ernani Rodrigues
Shuyi Zhang

Abstract: Many interesting problems from the real world are equivalent to NP-complete problems, such as Hitting Set. Such problems represent a very hard challenge to computing scientists, because unless $P = NP$ there is no polynomial time algorithm that provides optimal solutions for them. Thus, practical applications rely on approximate algorithms or heuristics. Here we present two stochastic methods based on heuristic search to solve this problem. We demonstrate that, with enough processing time available, it is possible to generate better results than a greedy logarithmic approximation, towards the optimal solution.

1 - Introduction

HITTING SET is an NP-complete problem that can be used to abstract a number of problems in computing science [4]. HITTING SET is defined as follows:

(HITTING SET) Given a collection of sets $C = \{S_1, \cdots, S_m\}$, a minimum hitting set for C is the smallest set H such that, for all $S_i \in C$, $H \cap S_i \neq \emptyset$.

There is a greedy approximation algorithm to solve HITTING SET in polynomial time that is known to produce good results [4]. Here we use stochastic methods to solve that problem, with the goal of observing the advantages and disadvantages of such methods in comparison with existing deterministic strategies.

We have proposed to solve HITTING SET with two different approaches: Monte Carlo Tree Search and Genetic Algorithm. Previous steps of this project described briefly the artifacts that were planned to be generated. Here the software artifacts will be described in more detail.

The experimental evaluation of our approach on a set of more than 400 benchmarks has revealed that it is possible to produce results that are equal or better than the greedy heuristic. However, in

order to achieve such precision, we have to use more processing time. In the worst case, our approaches were over 7000 times slower than the greedy algorithm.

2 – Genetic Algorithm

Genetic algorithms simulate the natural phenomenon that causes the evolution of species [7]. Such phenomenon can be applied to computing science to generate solutions for hard problems. In this kind of strategy, it must be possible to map a given solution of the problem into an individual of a population. It is also necessary to define some measure the fitness of each individual.

In this section we describe how we can solve HITTING SET with a genetic algorithm. First, we present the modeling of the problem into the genetic algorithm approach. Second, we discuss the engineering choices of our implementation. Finally, we list the parameters of our implementation and discuss the effects of each one in the performance of our approach.

2.1 – Modeling

In order to solve HITTING SET with a genetic algorithm, we must define a suitable representation of the solutions of the problem. Such representation is used to describe the individuals and their genetic characteristics.

In this work, we adopt the same modelling proposed by Li and Yunfei [8]. We represent each element of the set $S = S_1 \cup S_2 \cup \cdots \cup S_m$ as a different position in a bit vector, where $S_1, S_2, \cdots, S_m$ are the sets of the input of HITTING SET. In that bit vector, the value true means that the element of that corresponding position belongs to the hitting set. Each position of the bit vector is a chromosome of the individual of the genetic algorithm. Thus, we can model our individuals as arrays of Boolean values, as shown in figure 1.

Bool	Bool	Bool	...	Bool

Figure 1: individual of the genetic algorithm.

The measure of fitness is very straightforward, as well. The value of the fitness of a given individual is given by the number of elements of the array of Boolean that has the value FALSE. The greater the number of FALSEs in the array, the least is the size of the hitting set and, consequently, the better is the solution that it represents.

2.2 – Implementation Details

The Genetic algorithm was implemented inside LLC, the assembly code generator of LLVM [9]. It was implemented in such a way that it replaces the greedy solution implemented by de Krujif et al. [5].

We chose to implement our algorithm inside LLVM for two main reasons. First, our motivation to do this work is the paper published by de Krujif et al.; we wanted to investigate if stochastic methods could give better results in a reasonable amount of time. By implementing our solution inside the same compiler, we make it possible to perform fair comparisons between both approaches.

Second, LLVM provides many resources that make it easy to measure our performance. The compiler infrastructure allows the programmer to easily add parameters and statistics to its tools. Furthermore, it provides a well-suited testing infrastructure, with hundreds of test cases and tools for summarizing statistics.

2.3 – Parameters

Genetic algorithms can be tuned by a large number of parameters that control many aspects of the algorithm, such as the exploration/exploitation ratio and how fast the algorithm converges into a solution. Here we define the parameters that control the behaviour of our algorithm:

- **Rand_Seed** – Pseudo random number generators are deterministic functions if you know the seed beforehand. This parameter controls the seed for the function `rand` of the standard C/C++ library. This parameter allow us to reproduce results from tests and debug the code.
- **Population_Size** – The size of the population is the parameter that controls the amount of individuals that our algorithm handles at the same time. This parameter is capable of controlling the amount of memory that the algorithm will use. Large populations indicate that a large area of the search space can be analyzed at the same time.
- **Number_Of_Generations** – The number of generations control the time spent by the algorithm until it stops the evolution of the populations. This parameter should be tuned together with the parameters that control the speed of convergence (e.g. Tournament_Size).

- **Use_Elitism** – This parameter indicates whether the algorithm keeps or not the best individual from one generation to the next. In some cases the elitism leads to premature convergence. Thus, this parameter will be considered during the experiments.
- **Tournament_Size** – This parameter controls the number of the individuals that "fight" for survival in the tournament. The value of this parameter controls how long it takes until the algorithm converges to a single solution.
- **Mutation_Probability** – The probability of mutation controls the probability of random changes made to the genetic code of the individuals. This parameter controls how often the algorithm explores random areas in the search space.
- **Crossover_Probability** – The crossover probability determines how often two individuals generate descendants with crossover. This parameter controls the exploitation of a given area of the search space.

3 - Monte Carlo Tree Search

3.1 – Modeling

For the HITTING SET problem, we have the input C = collection of S_i, and U = the set of all elements in S_i. We try to model this problem as a tree search problem. When doing the tree search, for an intermediate state(or position) P_i, let H_i stand for the current hitting set, C_i stand for the collection of remaining sets to be hit, and X_i stand for the set of remaining elements *e* such that $e \in U$ but $e \notin H_i$. The search starts with $H_0 = \emptyset$, $X_0 = U$ and $C_0 = C$ (original collection). For each move, we pick one element $e \in X_i$ and put it into H_i, i.e. $H_1 = H_0 \cup \{e\}$, $X_1 = X_0 - \{e\}$, and $C_1 = C_0 - \{S_i \mid e \in S_i\}$. The size of valid moves equals to $|X_i|$. This search step terminates until we find $C_1 = \emptyset$ and $X_i = \emptyset$. The terminal state score is evaluated by the final size of H, where the smaller |H| is, better the result is.
We propose to use Nested Monte Carlo Tree Search (NMCS) [3] to solve this problem. According to Browne et al. [2], NMCS is claimed to have good results in some EXACT SET COVER problem like Sudoku, so we also expect to get good result in this HITTING SET problem, which can be reduced to the SET COVER problem.

3.2 – Approaches

a. Current work: NMCS with AMAF Heuristics [1]

All-Moves-As-First (AMAF) [6] heuristics is introduced to Monte Carlo Tree Search for getting more statistics. Since Monte Carlo Tree Search is usually time consuming, especially NMCS, this method can help to improve both the precision and efficiency of our Hitting Set solution. However, the AMAF stats may not be very reliable in some particular types of game. We think that AMAF will be a good heuristic because if we get a near-optimal solution with a particular playout move sequence, we can switch the order of these moves and still get the same near-optimal solution.

E.g. In our Hitting Set problem, if we are in a position P with valid moves M_1, M_2, M_3… M_k, each move has 3 AMAF stats: First is the best AMAF score; second is the number of times this move is picked; third is the average AMAF score. If a playout sequence starting from M_1 is M_1 -> M_3 -> M_2. Then the moves M_1, M_2, M_3 will receive the same AMAF score from the result of playout. After each move under P receives at least 1(or some threshold) AMAF score, the move with the best AMAF score will be returned. This approach will be applied into the each level of the NMCS.

3.3 – Algorithm Details

```
int nested (position, level, seq)
    1   best score = INF
    2   while not end of game
    3       seqSoFar = seq
    4       While ({move to sample} != ∅)
    5           m = chooseMove (position)
    6           if level is 1
    7               move = argmin_m (sample ( play (position, m), seqSoFar))
    8           else
    9               move = argmin_m (nested ( play (position, m), level – 1, seqSoFar))
    10          if score of move < best score
    11              best score = score of move
    12              best seq = seqSoFar
    13      update AMAF stats foreach move in seqSoFar
    14      {move to sample} = {move to sample} – m;
    15  bestMove = move of best AMAF stats
```

```
16      position = play (position,bestMove)
17      seq = seq + bestMove
18  return best score
```

The above detailed algorithm is based on the Cazenave's algorithm with the best sequence and AMAF stats updated. To be clearer, for each move we sample, we will receive the best score and the best move sequence starting from this move. These two values are returned by recursive calling the function nested with current level -1. Then for each move in the sequence, we will update the AMAF stats, which includes the **best AMAF score**, the **number of visits** on this move, and the **average AMAF score** of this move. After sample enough moves from a position P, we will choose the move with the best AMAF score as the best move.

Similar to the ideas from Akiyama, we also have different tie-break policies when we have multiple moves have the same AMAF stats, we can break the tie based on 4 different policies: 1. pick the move with the highest average AMAF score; 2. pick the move that is most sampled; 3. pick the move that is most picked; 4. Pick the move uniform randomly.

3.3.1 – Use AMAF Stats to Reduce the Number of Sampling

We will have two different ways to choose the move as a starting move to sample at each position: 1. sample starting all the available moves; 2. only sample staring the moves without any AMAF stats, namely, the number of visits equals to 0. This will help us reduce the number of sampling and the time complexity of this algorithm. However, there is a trade-off between the speed and the accuracy since we collect fewer stats than sampling starting all moves.

3.3.2 – Biased Sampling

If we choose to only sample starting the moves without any AMAF stats. The order of sampling will matter. Here we consider that the move can hit more sets in current state will have a higher priority to be sampled. We will pick moves to start sampling according to their priority.

3.4 – Implementation Details

The Monte Carlo Tree Search part is an independent C++ class that stands alone from the LLVM LLC part. However, we still put it into their MemoryIdempotenceAnalysis.cpp file to get their input more easily. My implementation will receive the generated collection of sets as an input, which is in the

structure of *List<set<element> >* from our implementation inside LLC, and return the solution *set<elements>* as a hitting set to the LLC. This makes the code easier reuse and modify.

Our implementation is based on c++98 standard. In order to speed-up, we choose *tr1::unordered_map* for storing the *element,* which has O(1) average access time. It save much time when we look up a hash table and delete an *element*. We also used *tr1::mt19937* for generate better random number sequence.

3.5 – Parameters

For our current AMAF implementation done so far, there are the following parameters to be tuned:

- Number_Of_Runs – Number of runs of NMCS function, the best solution among those runs of NMCS calls will be returned.
- Nested_Level – The level for the nest in NMCS. Usually we choose the value from 1, 2 and 3.
- Move_Choose_Policy – How we choose a child move for starting sampling
    1. All – sample all available children moves.
    2. No_AMAF – sample the move without any AMAF stats.
    3. No_AMAF_Biased – sample the move can hit the most set currently and without AMAF stats first.
- Tiebreak_Policy – The policy to break tie where there are more than 1 move have the same best AMAF score, there are 3 policies:
    1. AVG – Pick the move with the highest average AMAF score.
    2. Freq – Pick the move that is most picked.
    3. Hitting_Number – Pick the move that hits the most number of set in current state.
    4. RANDOM – Randomly pick 1 move.

4 – Experiments

In this section we describe the experimental evaluation of our stochastic approach. We have implemented our techniques in LLVM-3.0 and tested it with LLVM test-suite, a set of more than 400 benchmarks publicly available and distributed together with LLVM.

4.1 – Nested Monte Carlo Tree Search

The LLVM test-suites consists Single-Source test cases (about 250 programs) and Multi-Source test cases (about 150 programs). Single-Source test cases consist of small and medium test cases, while

Multi-Source ones consist of large and complicated test cases. We experimented both Single-Source and Multi-Source test cases several times to get the following results.

4.1.1 – Single-Source Test Cases

The first experiment is that we compares our Nested Monte Carlo Tree Search implementation with Number_Of_Runs = 5, Nested_Level = 1, Move_Choose_Policy = 1 (All), and Tiebreak_Policy = 1(AVG) comparing with the original greedy approximation algorithm:

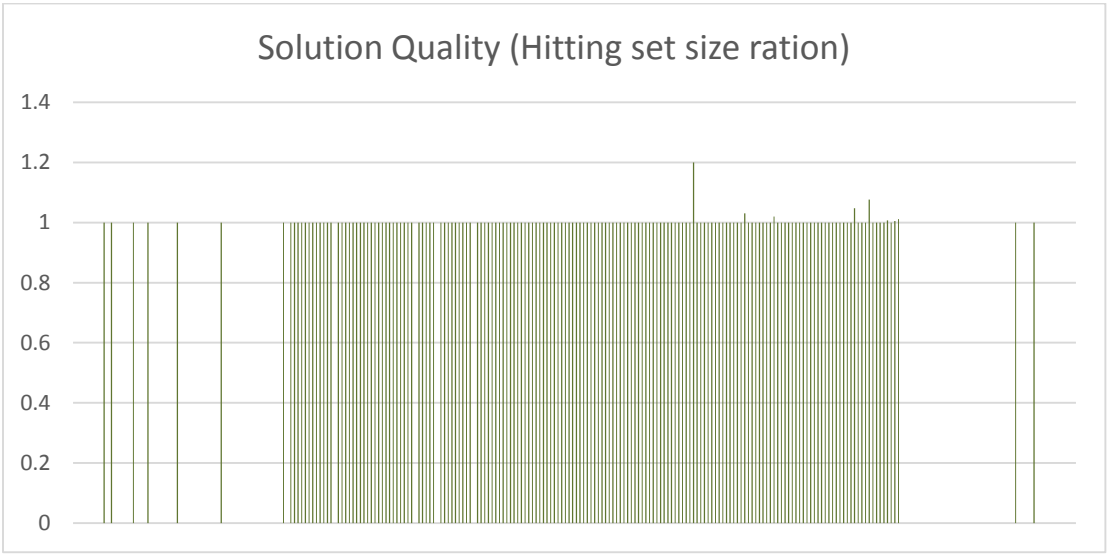

Figure 1. Solution Quality Ratio (Hitting set size of approx. alg. / Hitting set size of NMCS alg. (5,1,1,1))

Figure 1 is the solution quality ratio between the two algorithms which is measured by #region of approx. alg. / #region of NMCS alg. Since the least region generated means the better the solution quality is. We can observe that in all cases, the solution provided by NMCS search algorithm is better or equal to the greedy algorithm. This is the same as our expectation.

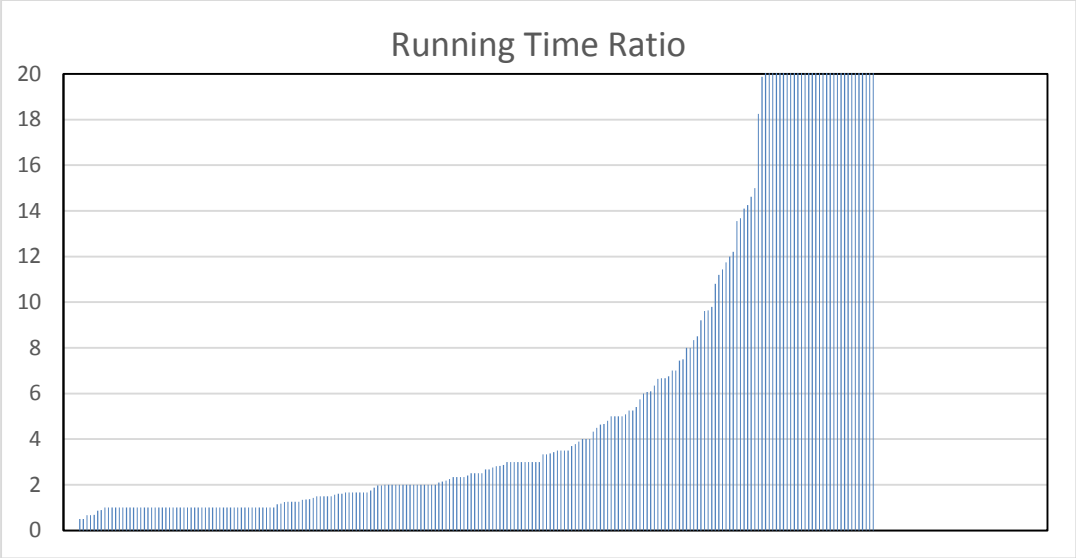

Figure 2. Running Time Ratio (Time of NMCS alg./ Time of approx. alg. (5,1,1,1))

Figure 2 shows the running time ratio of our NMCS algorithm vs greedy algorithm. We found that only 20% of the test cases we are on par or better than greedy algorithm in terms of running time. For the rest cases, we are much slower than their algorithms. Also, we also found that the right most 20% cases in Figure 2, the number or sets in the input are over 100, while there are fewer intersection between these set. In this situation, our NMCS with Move_Choose_Policy = All will search much deeper and it causes the running time grows dramatically. Our slowest cases is 7585 times slower than approximate algorithm.

4.2.1 – Multi-Source Test Cases

Multi source test cases consists of many very large test cases. Since if we want to apply Move_Choose_Policy = All in this case, it will consume too much time, we compare our NMCS algorithm with Number_Of_Runs = 5, Nested_Level = 1, Move_Choose_Policy = 2 and 3 (No_AMAF and No_AMAF_Biased), and Tiebreak_Policy = 1(AVG). In this way, we only choose the child move with no AMAF stats and it will save very much time in very large cases.

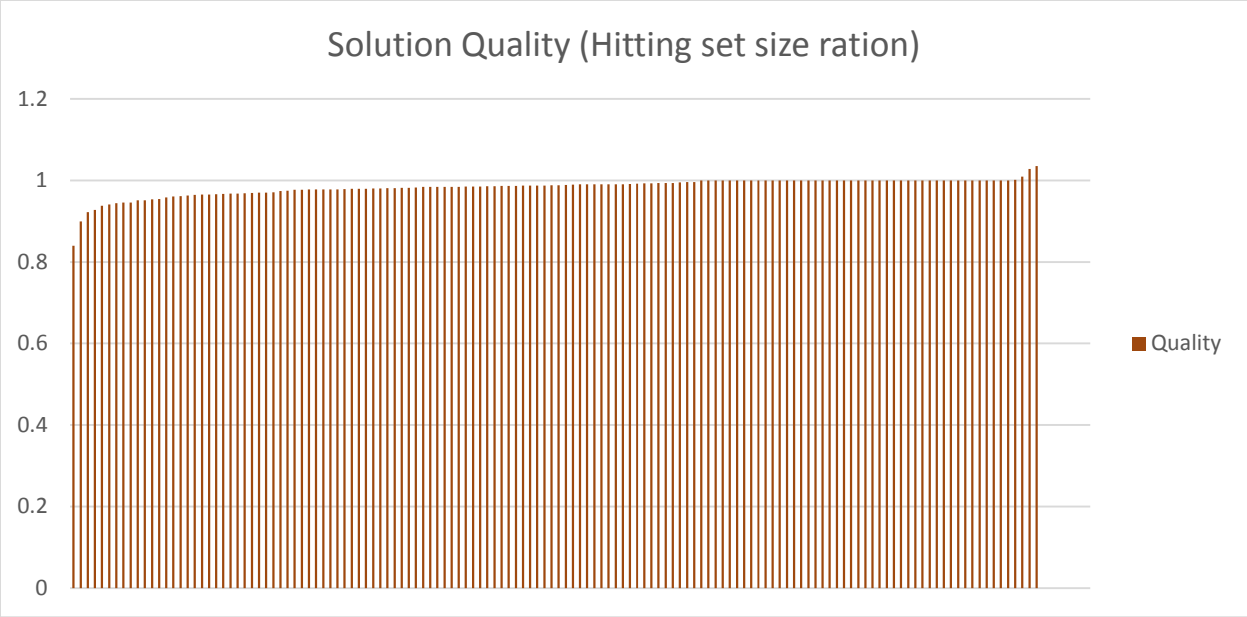

Figure 3. Figure 1. Solution Quality Ratio (Hitting set size of approx. alg. / Hitting set size of NMCS alg. parameters = (5,1,2,1))

Figure 3 shows the solution quality of NMCS with Move_Choose_Policy = No_AMAF, in comparison with the greedy algorithm on Multi-Source test cases. The figure indicates that 62.5% of our result is slightly worse than the greedy algorithm. This is acceptable since we use AMAF statistics reduces a lot of simulations.

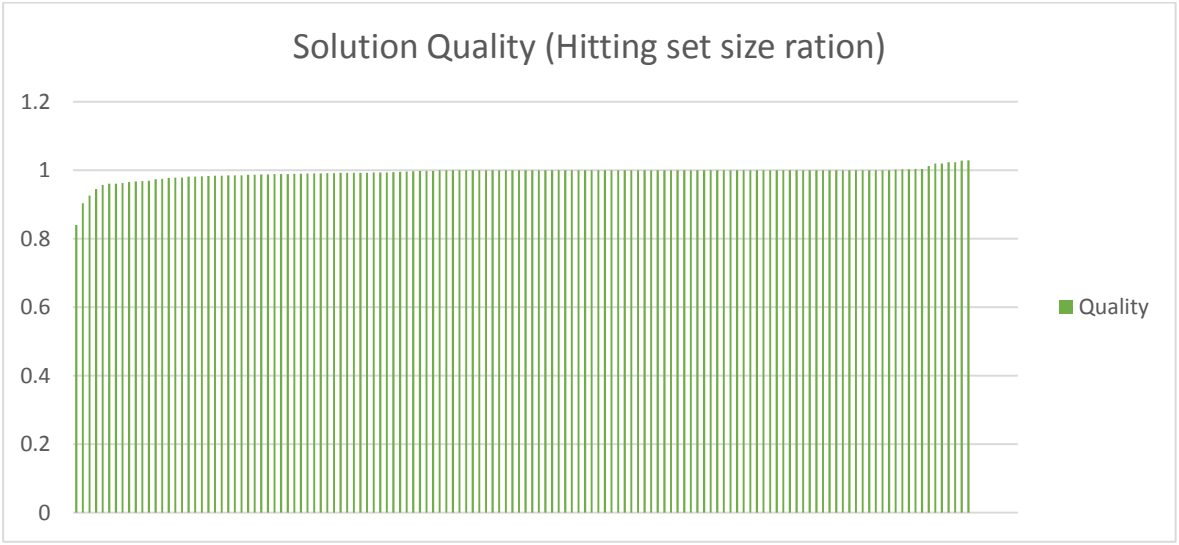

Figure 4. Figure 1. Solution Quality Ratio (Hitting set size of approx. alg. / Hitting set size of NMCS alg. parameters = (5,1,3,1))

However, if we apply the biased child move choosing policy, we can improve our solution quality. Figure 4 shows that only 38.8% of our solution is a little bit worse than the approximate one. Comparing to the unbiased move choosing policy, it is a huge improvement.

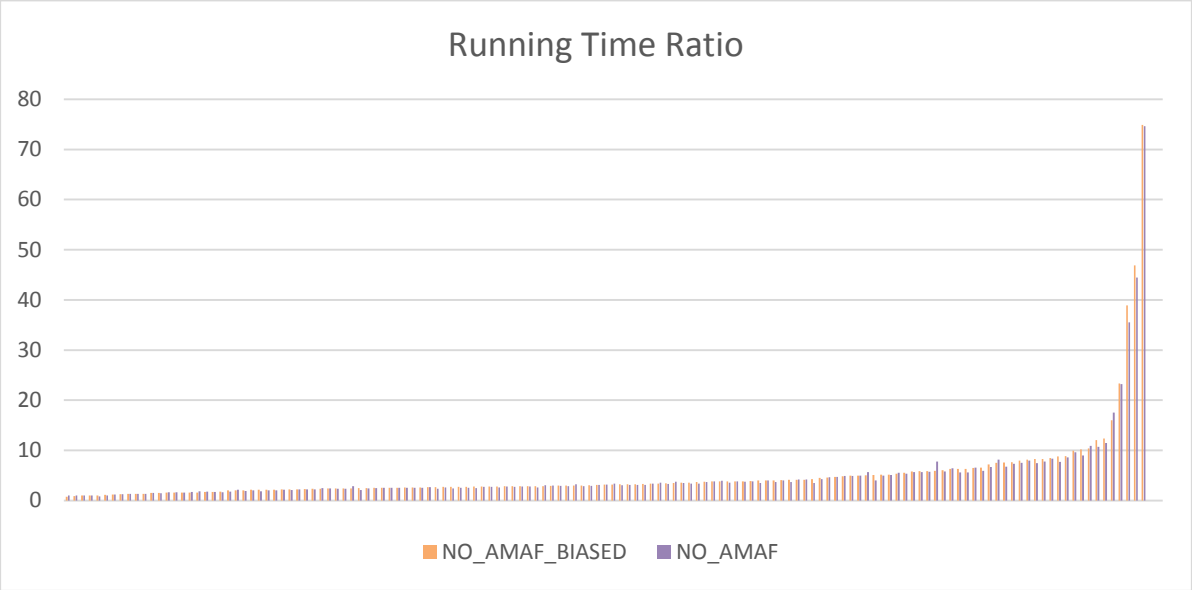

Figure 5. Running Time Ratio (Time of NMCS alg. / Time of approx. alg. parameters = (5,1,2,1) and (5,1,3,1))

In comparison of NMCS with Move_Choose_Policy = No_AMAF and approximate algorithm in terms of running time. In 65% cases, NMCS is blow 4 times slower than the approximate algorithm in very large test cases. In addition, the worst case is only 74.66 times worse than the approximate one. Comparing to MNCS with Move_Choose_Policy = All where the worst case is 7585 times slower in Single Source test, it's a remarkable improvement.

From the NMCS experiments, we found that NMCS with Move_Choose_Policy = All is the most accurate one, but it will take too much time for running for the worst case input (too many sets and very few intersection). However, NMCS with Move_Choose_Policy = No_AMAF and No_AMAF_Biased have a reasonable running time, especially No_AMAF_Biased one's precision is near the approximate solution. Therefore, I think the NMCS with Move_Choose_Policy = No_AMAF_Biased one has the most practical value comparing to the original one.

4.2 – Genetic algorithm

We have also performed a series of experiments with our genetic algorithm. In this experiment we have analyzed the whole LLVM test-suite (Singlesource and Multisource). The following results contain only the 100 largest programs, which generated the results with largest sizes. We executed the experiments in a machine with an intel Core i7 4700MQ 2.4GHz and 12GB of memory.

We tried different parameter configurations, but the one that gave the best balance between speed/precision was:

- Population_Size: 100
- Number_Of_Generations: 100
- Use_Elitism: True
- Tournament_Size: 2
- Mutation_Probability: 0.1
- Crossover_Probability: 0.8

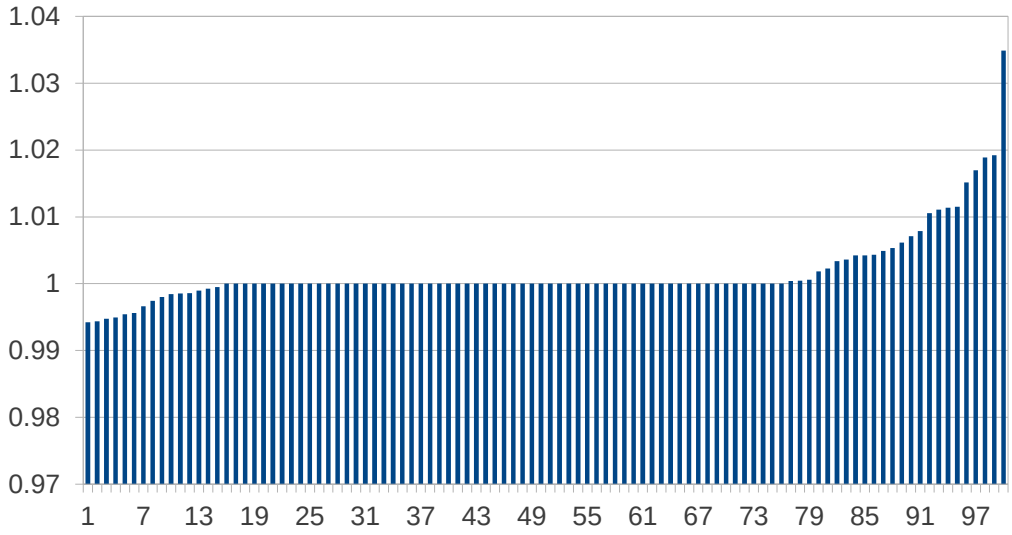

Figure 6 – Genetic algorithm – Precision

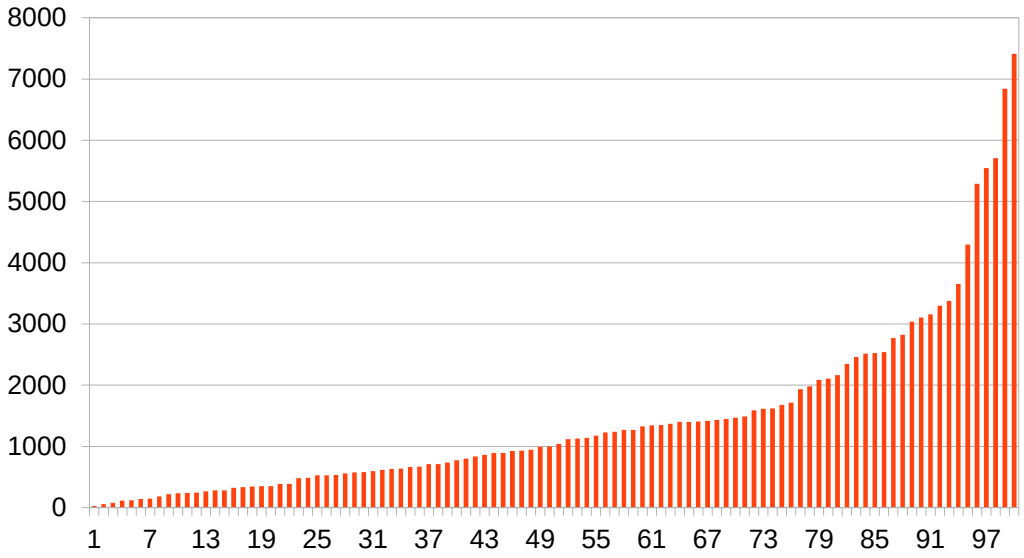

Figure 6 – Genetic algorithm – Time

Figure 6 shows the precision obtained with the genetic algorithm and Figure 7 shows the analysis time, both normalized by the result obtained in the greedy algorithm. In the configuration descripted above we were able to give a result equal or better than the genetic algorithm in 85% of the test cases. Furthermore, when our implementation returned a result worst than the greedy algorithm, our result was always less than 1% worse than the greedy algorithm. However, our processing time was dramatically higher than the one of the greedy algorithm. In average, we were more than 1000 times slower than our baseline. In the worst case, we were more than 7000 times slower.

5 – Conclusion and Future Works

The existing approximate algorithm for solving Hitting Set Problem is highly efficient and can produce good quality solutions with a logarithm approximation ratio. In our project, we tried two different ways to approach the problem. First is the Genetic algorithm; we implemented a solution briefly described in the literature and defined a set of parameters to tune the different aspects of the algorithm. Second is Nested Monte Carlo Tree Search, we try to modeling the Hitting Set Problem into a search problem and we apply the AMAF heuristics in two ways. First it helps to achieve more precise solution and our experiments show that it can get more-optimized solutions. Then we made a trade-off between the precision of the solution and the running speed by only

choosing moves without AMAF stats, this helps us to get almost the same level of efficiency as the approximate one. Last, we use biased move choosing policy to improve the precision which is on par with the approximate one.

Although we get good results in the perspective of solution precision, we still cannot beat the approximate algorithm in both precision and efficiency at the same time. We hope we can try more parameter tuning (e.g. try different tiebreak policies) in the future to get better results based on our existing implementation. Furthermore, we are hoping to try some other methodologies like Nested Rollout Policy Adaptation [10] to have a better policy when choosing the move in NCMS in order to save more time and improve our solutions.

6 – References

[1] Akiyama, Haruhiko, Kanako Komiya, and Yoshiyuki Kotani. "Nested Monte-Carlo search with AMAF heuristic." *2012 Conference on Technologies and Applications of Artificial Intelligence*. IEEE, 2010.

[2] Browne, Cameron B., et al. "A survey of monte carlo tree search methods." Computational Intelligence and AI in Games, IEEE Transactions on 4.1 (2012): 1-43.

[3] Cazenave, Tristan. "Nested Monte-Carlo Search." *IJCAI*. Vol. 9. 2009.

[4] Cormen, Thomas H., Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. Introduction to algorithms. Vol. 2. Cambridge: MIT press, 2001.

[5] de Kruijf, Marc A., Karthikeyan Sankaralingam, and Somesh Jha. "Static analysis and compiler design for idempotent processing." ACM SIGPLAN Notices 47, no. 6 (2012): 475-486.

[6] D. P. Helmbold and A. Parker-Wood, “All-Moves-As-First Heuristics in Monte-Carlo Go,” in Proc. Int. Conf. Artif. Intell., Las Vegas, Nevada, 2009, pp. 605–610.

[7] Davis, L. (Ed.). (1991). *Handbook of genetic algorithms* (Vol. 115). New York: Van Nostrand Reinhold.

[8] Li, L., & Yunfei, J. (2002). *Computing minimal hitting sets with genetic algorithm*. ZHONGSHAN (SUN YATSEN) UNIV (CHINA).

[9] Lattner, C., & Adve, V. (2004, March). LLVM: A compilation framework for lifelong program analysis & transformation. In *Code Generation and Optimization, 2004. CGO 2004. International Symposium on* (pp. 75-86). IEEE.

[10] Rosin, Christopher D. "Nested rollout policy adaptation for Monte Carlo tree search." *Proceedings of the Twenty-Second international joint conference on Artificial Intelligence-Volume Volume One*. AAAI Press, 2011.

Appendix A – Instructions to compile and run our program

- Create a folder to host the compiler (to be referenced as <root>).
- Go to the folder <root>

  cd <root>
- Download the GitHub source of LLVM with Idempotence extensions

  git clone https://github.com/mdekruijf/llvm.git

  cd llvm

  git checkout idempotence extensions
- Download clang

  cd tools

  svn co http://llvm.org/svn/llvm-project/cfe/trunk clang -r 149259
- Go to the folder <root>/llvm/lib/codegen

  cd <root>/llvm/lib/codegen
- Replace the files MemoryIdempotenceAnalysis.cpp and ConstructIdempotentRegions.cpp with the ones provided in the submission of this report.
- Configure and Make the executable in LLVM's root folder.

  cd <root>/llvm
  ./configure -disable-optimized
  make

  The executables will be generated in the folder <root>/llvm/Debug+Asserts/bin
- List the parameters added in the executable llc

  cd <root>/llvm/Debug+Asserts/bin
  llc -help-hidden
- Compile and analyze C/C++ programs using our program

  export PATH=<root>/llvm/Debug+Asserts/bin:$PATH

  clang -s -emit-llvm <file>.c -o <file>.bc

  llc -idempotence-construction=size <parameters defined in this report> <file>.bc -o <file>.s