Examining the effectiveness of branching heuristics in a CDCL-type SAT solver for Argumentation Framework

Ramraj Thirupathyraj[a] (ramraj1521@gmail.com), Sivakumar Palanisamy[b, 1](sivakumar.dpalanisamy@gmail.com)

[a] Department of Artificial Intelligence and Data Science, Coimbatore Institute of Technology, Coimbatore, Tamil Nadu, India 641014.

[b] Zoho Corporations Pvt. Ltd., Chengalpattu, Tamil Nadu, India 603202.

Corresponding Author:

Sivakumar Palanisamy

Zoho Corporations Pvt. Ltd., Chengalpattu, Tamil Nadu, India 603202.

Tel: (91) 80122-89764

Email: sivakumar.dpalanisamy@gmail.com

[1]Permanent Address: 76, Periyar Nagar, Chennimalai, Erode, Tamil Nadu, India 638051.

Examining the effectiveness of branching heuristics in a CDCL-type SAT solver for Argumentation Framework

Ramraj Thirupathyraj[a,1], Sivakumar Palanisamy[b,*]

[a]*Department of Artificial Intelligence and Data Science, Coimbatore Institute of Technology, Coimbatore, Tamil Nadu, India 641014.*
[b]*Zoho Corporations Pvt. Ltd., Chengalpattu, Tamil Nadu, India 603202.*

Abstract

This paper primarily focuses on investigating the importance of branching heuristics in solving random instances generated by the D(n, p, q) random model of Argumentation Framework (AF). In an effort to make such an investigation, the MiniSat solver with three different types of branching techniques is used to find a solution for the random AF instances. Firstly, the default VSIDS branching is considered since it will be useful to draw a conclusion if the modified branching performs better than the default one. Secondly, the branching rule BRmcha along with the BH3 branching heuristic is used in the MiniSat solver. Apart from these two techniques, a novel branching strategy (AROFS) mainly based on AF properties is introduced for this study. The branching technique (AROFS) is modified from VSIDS and uses certain techniques. As the last two branching techniques are designed particularly for solving AF instances and are not yet applied in any of the publicly available solvers, a study on it could benefit the modern AF solvers. Based on the empirical studies performed, we observed a heavy difference between the performance of the solver on employing it with the mentioned branching strategies which emphasises the importance of branching techniques in solvers. It is known that there is an easy-hard-easy pattern of hardness associated with the phase transition in the D(n, p, q) random model which affected the solver functioning before and after the phase transition. The solver with VSIDS branching performed poorly compared to the solver with other two branching techniques for q < q* (threshold for the phase transition), particularly from the value of q when the fraction of success instances started to increase. This shows the weakness of VSIDS branching in solving the harder AF instances. After the phase transition, the solver with all branching techniques under study performed without much difference.

Keywords: Argumentation Framework, SAT, MiniSat, Branching Heuristics

1. Introduction

1.1. Abstract Argumentation framework

An Abstract Argumentation Framework [AF], introduced by Dung in 1995, is represented as a pair (A, R) where A represents the set of arguments and $R \subseteq A \times A$ represents attack relation between two arguments in A. AF can hence be represented using a directed graph D(V, E) by respectively representing arguments in A and attack relations in R as vertices V and directed edges E of a directed graph.

For a given AF (A,R), Dung defined various semantics of acceptance in the form of Extensions in order to check if a particular argument or a set of arguments can be accepted (Baroni & Giacomin, 2009). **Extensions** are nothing but a subset of arguments that can be collectively accepted. A set S can be an extension if and only if it is **conflict-free** (i.e arguments in S should not attack another argument in S) and **admissible** (i.e any argument that attacks an argument in S is attacked by an argument in S). Various extensions defined by Dung are explained as below:
Complete Extension: A set of arguments S is a complete extension if and only if it is an admissible set and every argument that can be accepted with respect to S is present in S.
Preferred Extension: A set of arguments S is a preferred extension if and only if it is a maximal element among all the admissible sets of a given AF.
Stable Extension: A set of arguments S is a stable extension if every argument not in S is attacked by at least one argument in S.

[*]Corresponding author
Email address: sivakumar.dpalanisamy@gmail.com (Sivakumar Palanisamy)

Grounded Extension: A set of arguments S is a grounded extension if and only if it is the minimal complete extension of a given AF.

Of particular interest, this paper will deal with the computation of admissible sets, based on which the solver performance with all mentioned branching techniques will be examined.

1.2. D(n, p, q) – random model of directed graphs

In the literature, the Erdos-Renyi model of random graphs is widely used to generate random instances of AF. Represented by D(n, p), where n is the number of nodes, the model generates random instances by allowing each of the $_nC_2$ directed edges to be present in the graph with p probability.

(de la Vega, 1990) In 1990, Vega proved that kernels almost always exist in a random directed graph generated by the D(n, p) model as $n \to \infty$. A kernel K of a graph G (V, E) is a subset of vertices in V which are **'independent'** (i.e for all vertices not in K, there must be a successor in K). Vega's results can be directly applied to AF instances, where the kernel of a directed graph is nothing but the Stable extension of an AF instance. After observing the right proportion of symmetric attacks produced by Vega's model irrespective of the attack relation probability, (Gao, 2017) Yong Gao introduced a new random model for AF with a new controlled feature to produce symmetric attacks and provided a plausible interpretation for Vega's result in AF for the existence of stable extension almost always as $n \to \infty$.

The model introduced is represented as D(n, p, q) where n represents the number of arguments in an AF. With probability p, two arguments can have an attack relation and with the existence of the attack relation to be true, the attack relation can be symmetric with q probability. Hence, the probability for an attack relation to be one-way is 1-q with both directions of an attack having equal probability to exist. The theoretical studies carried out in (Gao, 2017) established the exact threshold for the phase transition of the probability of the existence of preferred or stable extension in the D(n, p, q) model.

The empirical studies done in section-4 of (Gao, 2017) to prove the theoretically established threshold properties, introduced a new backtracking-style algorithmic solver with the branching rule BRmcha and various branching heuristics for extension computation in AF generated by the D(n, p, q) model and in section-5 of (Gao, 2017), further works were called for the state-of-the-art AF solvers to be applied for the new random model to better understand the effectiveness of branching heuristics used. As most of the modern AF solvers are based on SAT solving techniques with CDCL being their core functionality, this paper will mainly focus on employing the branching rule BRmcha and the branching heuristic BH3 to a CDCL-type SAT solver. In order to understand the efficacy of the applied branching technique, random AF instances which are generated using the D(n, p, q) model with different values of p and q are solved using a SAT solver with its default branching and with the introduced branching.

1.3. MiniSat-2.2 and SAT solver compatibility for AF

MiniSat(Eén & Sörensson, 2004) is a SAT solver that makes use of conflict-driven clause learning, two-literal watching scheme, VSIDS branching heuristic and non-chronological backtracking for solving a given propositional formula. Among numerous CDCL-type SAT solvers available, MiniSat is chosen for this empirical studies because of its well-documented nature and its main functionalities being used in many modern SAT solvers like Glucose, Syrup and Maple LCM (Heule et al., 2018; Balyo et al., 2017).

Given an AF instance, the computation of the existence of a non-empty admissible set or stable / preferred extension is an NP-Complete problem (Table 1 of (Dvořák et al., 2014)). The boolean SAT problems are well studied and there exists numerous powerful solvers in the form of MiniSat, Glucose, CryptoMiniSat and Maple LCM. Hence, based on the reduction approach mentioned in Section-3.1.1 of (Charwat et al., 2015), the AF can be translated into a propositional logic (in CNF format) which can be used to solve and compute the extension for a given AF. As mentioned in section 1.1, this paper will deal with the admissible set computation for a random AF instance.

Using SAT encoding techniques described in section 3, an AF instance is translated into a CNF formula and written into a DIMACS file which is then used by the solver for admissible set computation. Since a DIMACS file which is generated for a given AF instance is used for all solvers (MiniSAT with various branching techniques under study), the time taken comparison report presented in this paper will include the time taken by the solver for admissible set computation alone.

1.4. Branching Techniques

MiniSat uses VSIDS (Variable State Independent Decaying Sum) (Liang et al., 2015) as the default branching heuristic where the activity of variables that are involved in recent conflicts are bumped up and given priority in the decision making, while the activity of variables are periodically reduced so that the variables that are not associated with the recent/current decision branch are not given priority in the decision process. Apart from VSIDS, the branching rule BRmcha and the branching heuristic BH3 are implemented in MiniSat for examining their performance. In addition to the above two, a whole new branching technique is introduced and is being examined for the D(n, p, q) random model. This new branching method exhibits the number of times an argument is involved in attack relationships and utilizes it so as to affect as many clauses of a given SAT problem as possible. In the rest of the paper, this custom branching technique will be called AROFS(Attack Relation Occurrence Factor Summation).

As per the observations made, random AF instances almost always have an admissible set at the value of q close to the threshold q* described in (Gao, 2017). In the experiment conducted with different combinations of p and q, the MiniSat with BRmcha-BH3 / AROFS clearly outperformed the MiniSat with VSIDS branching.

The rest of the paper follows the mentioned order. The encoding logic used for argumentation semantics will be explained in section 2 while section 3 will describe the different branching heuristics applied to the SAT solver in detailed algorithmic definitions. Section 4 will explain empirical implementation details and a detailed report on the observations made. In section 5, a conclusion and future extension of this paper will be presented.

2. SAT Encodings for Argumentation semantics

Based on the standard encoding techniques described in (Besnard & Doutre, 2004), the AF instance can be converted into a CNF propositional formula that can be solved by a SAT solver. The SAT encoding details for admissible set computation is described in this section. For an AF(A, R), a set $adm \subseteq A$ can be admissible if:

i) it is **conflict-free** i.e no attack relation should exist between the argument in adm.

$$\forall_{a,b \in adm} \quad (a, b) \notin R$$

ii) it **defends** all attacks to the arguments in adm.

$$\forall_{a \in adm, b \notin adm}: \quad (b, a) \in R, \ \exists_{c \in adm}: \quad (c, b) \in R$$

2.1. CNF encoding for conflict-free property

Propositional Logic:

$$\forall_{a \in A}(\forall_{(b,a) \in R} \quad (a \Rightarrow \neg b))$$

CNF Format:

$$\forall_{a \in A}(\wedge \quad (\forall_{(b,a) \in R} \quad (\neg a \vee \neg b)))$$

which can be simplified as:

$$S_{\textit{conflict-free}} = \forall_{a \in A}(\wedge \quad (\neg a \quad (\forall_{(b,a) \in R} \quad (\vee \neg b)))$$

2.2. CNF encoding for admissible property

Propositional Logic:

$$\forall_{a \in A} \quad (\wedge \quad (a \Rightarrow (\forall_{(b,a) \in R} \quad (\wedge \quad (\forall_{(c,b) \in R} \quad (\vee \quad c))))))$$

CNF Format:

$$S_{\textit{admissible}} = \forall_{a \in A} \quad (\forall_{(b,a) \in R} \quad (\wedge \quad (\neg a \vee \quad (\forall_{(c,b) \in R} \quad (\vee \quad c)))))$$

The complete CNF formula for admissible set computation:

$$S_{\textit{admissible-set}} = S_{\textit{conflict-free}} \wedge S_{\textit{admissible}}$$

3. Implementation of branching heuristics in MiniSat

3.1. BRmcha and BH3 branching

3.1.1. Computation of hostile and friends set

For the implementation of BRmcha branching rule and BH3 branching heuristics, the solver would need to compute and update the list of hostiles and friends based on the latest decision made. To make such computation less costly (in terms of time complexity), a 2D boolean matrix (`attack_details`) is used to store the attack relationship between all pairs of nodes in an AF instance. The details of this boolean matrix is as below.

$\forall_{(b,a)\in R}$: `attack_details`$[b][a] = True$

```
hostile(x) : [y forall y in variableSet:
                 attack_details[y][x] = True
                 attack_details[x][y] = False
             ]
friends(x) : [y forall y in variableSet:
                 attack_details[y][x] = False
                 attack_details[x][y] = False
             ]
```

Based on this matrix, the *currentHostileSet* and *currentFriendSet* will be computed for each of the decision processes. Hence, in case of conflict, these two lists should be reverted to their past state based on the backtracking level that the SAT solver decides to jump back. This is made possible by using (*friendLimStack* and *hostileEndLimStack*) for maintaining the details of friends and hostile set in stack. With careful consideration of space constraint, it will not be efficient to push the two lists into the stack for every decision level. Hence, the stack operations are carried out for certain limit values which can be used to compute the list of active friends (Range: 0 to *cfEndLim*) in currentFriendSet and active hostiles (Range: 0 to *chEndLim*) in *currentHostileSet*. These limits will be updated during *currentFriendSet* and *currentHostileSet* computation, pushed to stack before making a decision and popped out from stack while back tracking.

Apart from maintaining these limits, it is necessary to maintain the state of every element in *currentHostileSet* to keep track of its hostile activeness and the decision level at which it is removed from the hostile list. The variables in *currentHostileSet* will only be removed while back-jumping. For instance, if a variable is added from *currentFriendSet* to *currentHostileSet* in decision level 5, it will be removed from the list during back-jumping to level 4. If a decision variable in a decision process attacks a variable in the *currentHostileSet*, the hostile variable will be transformed in such a way that its actual value and the current decision level at which its hostile nature is removed will be maintained so as to revert it to its past state while back jumping.

After proper construction of *currentHostileSet* and *currentFriendSet*, *pivotSet* will be selected based on BRmcha branching rule before updating the activity of pivot elements based on BH3 heuristics. Finally, the *orderHeap* (max heap) of MiniSat will be built for the pivot set and the variables in *pivotSet* are selected in the order of their activity value. When the pivot set becomes empty, the next decision variable will be taken from the list of unassigned variables which will contain the elements of the friend set. This is because of the way in which the SAT solver works since all hostile and attacked variables based on decisions made until this point would be assigned false. Hence, the remaining unassigned variable list will only contain variables from the friends set. The algorithm for handling the update operation on *currentHostileSet* is described below.

Algorithm 1: updateHostileSet (next,currentHostileSet,currentFriendSet)

```
Global Variables:
  cfEndLim          % end limit of friendSet
  chEndLim          % end limit of hostileSet
if currentHostileSet is empty then
    currentHostileSet = hostile(next);
else
    forall friend in currentFriendSet [Range: 0 to cfEndLim] do
        if hostile(next) contains friend then
            currentHostileSet.append(friend);
            chEndLim++;   % new variable addition will be made at the end of the list.
        end
    end
    forall hostile in currentHostileSet [Range: 0 to chEndLim] do
        if next attacks hostile then
            Transform (hostile, level()); % hold its value and the level when it lost its hostile nature.
        end
    end
end
```

It can be seen from the above algorithm that the element addition and removal in *currentHostileSet* is coupled with the *chEndLim* and transform function. At any point of the algorithm execution, the variables that occur in the list below the limit *chEndLim* and the variables that are not transformed will be considered as current hostile elements. In a similar way, *updateFriendsSet()* method can be implemented where the element to be removed will be moved to the end of the *currentFriendSet* list and *cfEndLim* will be decrement so as to make the friend element currently inactive. As elements will only be removed from the *currentFriendSet*, there is no need to transform the elements in the list for removal.

3.1.2. BRmcha branching rule

Branching rule BRmcha is the most-constrained-hostile-argument first technique. As per the section 4 of (Gao, 2017), the branching rule is used to select the pivot set from which the next decision will be made. Before implementing this rule, it is necessary to maintain the details of the cur rentFriendSet and currentHostileSet for the latest branch in the decision tree since the details of these sets are needed to construct the pivot set.

Based on BRmcha, pivot set $P = N_F^-(h)$ such that for all hostiles of the decision variable, the friend set that attacks a hostile should be minimal. In other words, the hostile that is attacked by the least number of friends in the friend set will be considered and the corresponding friend set will be made as the pivot set.

Algorithm 2: getPivotSet (currentHostileSet, currentFriendSet)

```
minPivotSize = 0;
pivotSet = [];
while currentHostileSet is not empty do
    hostile = currentHostileSet.next();
    tempHostile = [];
    forall friend in currentFriendSet do
        if friend attacks hostile then
            tempHostile.add(friend);
        end
    end
    if tempHostile.size() < minPivotSize then
        pivotSet = tempHostile;
        minPivotSize = tempHostile.size();
    end
    tempHostile.clear();
end
return pivotSet;
```

3.1.3. BH3 branching heuristic

Once the pivot set is constructed, the order in which the elements of pivot are considered is decided based on the activity of the variables in the set, which is computed by BH3 heuristic. BH3 branching is nothing but the most-effective-argument-first technique where the most effective argument is decided by $\mid N_H^+(v) \mid - \mid N_F^-(v) \setminus N_F^+(v) \mid$. As per this formula, the variable that attacks more number of elements in currenHostileSet and non-symmetrically attacked by less number of elements in currentFriendSet will be the most prioritized one in the pivot set.

For all elements in the pivot set, the result of the given formula will be their activity value which is used for ordering them. In MiniSat solver, orderHeap has been used for picking the variables in order. The orderHeap is reused for our purpose and is built newly after deciding the activity of all variables in the pivot set. This is primarily because of the fact that the pivot set drastically changes after making each and every decision and there is a need to remove the variables that are not in the current pivot and add new elements to the pivot set. To avoid such complexity of element removal and addition in heap, it will be newly built after a decision process.

3.1.4. Illustration of changes in MiniSat

The algorithmic description of BRmcha branching rule and BH3 branching heuristic presented above must be suitably adapted in MiniSat algorithm so that it will be possible to perform a study on its effectiveness. The major components of the branching technique to be incorporated are currentFriendSet, currentHostileSet and pivotSet. This subsection will briefly explain the changes done in the methods of MiniSat algorithm to properly maintain these data structures for correct functioning of branching as expected.

Algorithm 3: updateActivity (pivotSet,currentHostileSet, currentFriendSet)

```
forall pivotElement in pivotSet do
    if valueOf(pivotElement) != Undef then
        continue;    % no need to consider assigned variables for pivotSet.
    end
    pivotElement.activity = 0;    % the previous values of the element will not be considered.
    forall hostile in currentHostileSet[Range: (0 to chEndLim)] do
        if pivotElement attacks hostile then
            pivotElement.activity += 1;
        end
    end
    forall friend in currentFriendSet[Range: (0 to cfEndLim)] do
        if hostile(pivotElement) has friend then
            pivotElement.activity -= 1;    % non-symmetrically attacked by friend.
        end
    end
end
orderheap.build(pivotSet);    % orderHeap is a Global variable used in MiniSat.
```

The method *pickBranchLit()* in Algorithm 4 is used in MiniSat to select the next decision literal (to return a literal, the selected variable will be assigned with random/default/user-defined polarity). The method usually picks the root node from the orderHeap and returns corresponding literal. When no variable is in orderHeap, litUndef will be returned. In the implementation of BRmcha and BH3, the orderHeap might become empty even if there are some unassigned variables because the orderHeap will contain the elements of the pivot set which is only a part of the variables list. Hence, if an unassigned variable is available, it is necessary to pick one. Once the next variable is selected, the limit variables must be pushed to their corresponding stacks before updating *currentHostileSet, currentFriendSet and pivotSet* lists. Finally, *updateActivity()* will be called to construct the orderHeap based on the computed value of pivotSet elements' activity values.

Algorithm 4: MiniSat::pickBranchLit ()

```
next ← pick randomly or from orderHeap
———————— changes related to BRmcha and BR3 ————————
if next is varUndef then
    ———— This will happen if pivot becomes empty ————
    next ← pick from variable list
    if next is varUndef then
        return litUndef;    % No unassigned variable exists
    end
end
friendStackLim.push(cfLim);    % Stack operation
hostileEndLimStack.push(chEndLim);    % Stack operation
updateHostileSet();    % Algorithm 1
updateFriendSet();    % Implementation similar to Algorithm 1
pivotSet ← getPivotSet();    % Algorithm 2
updateActivity();    % Algorithm 3
———————— end of changes related to BRmcha and BR3 ————————
return Lit(next);    % Handling in this line in MiniSat is not changed except for the default polarity updated to False
```

Another important change is done in the *cancelUntil()* method where the pushed stack values of limits (in Algorithm 4) are popped out based on the backtracking level. As these limit variables are global, it is sufficient to just pop out the correct values.

Algorithm 5: MiniSat::cancelUntil (blevel)

```
if currentDecisionLevel > blevel then
    % unassign all variables assigned until the decision level blevel and insert them back to orderHeap
    % reduce trailLim

    ———————— changes related to BRmcha and BR3 ————————

    chEndLimStack.shrinkUntil(level);
    chEndLim = chEndLimStack.pop();
    cfEndLimStack.shrinkUntil(level);
    cfEndLim = cfEndLimStack.pop();
    unTransform(currentHostileSet, blevel);    % revert the element to its original value based on blevel
    updateActivity();    % Algorithm 3

    ———————— end of changes related to BRmcha and BR3 ————————
end
```

The *updateActivity()* call done after stack operations will make use of these global variables to compute the active friends and hostiles and perform the necessary activity-update handlings. As a final change, the default *varBumpactivity()* and *varDecayActivity()* of MiniSat, which are used for variable activity increment and variable activity decay respectively, are not required for our branching technique and hence, not used in the implementation.

3.2. AROFS branching

Attack Relation Occurrence Factor Summation (AROFS) is based on the number of times an argument is involved in attack relationships. The intuition behind this branching technique is to pick a variable that will influence as many clauses as possible so as to reach the solution or conflict quickly. This branching technique almost follows the same process as VSIDS. In case of conflict, VSIDS uses incFactor (increment factor) to bump variables' activity whereas AROFS uses attackRelOccFactor (attack relation occurrence factor - **arOccFactor**) for such activity bumping. For all variables in an AF, (Algorithm 7) arOccFactor will be computed during the clause addition in MiniSat algorithm where it should pass the necessary condition that the clause must not be learnt or unit. With such conditions, it can be clearly seen that the mentioned factor will be computed only for problem clauses that are not unit.

Algorithm 6: MiniSat::varBumpActivity (var)

```
activity(var) += arOccFactor(var);    % check Algorithm 7 for arOccFactor computation.
if activtiy(var) > 1e100 then
    % default MiniSat activity rescaling logic
end
orderHeap.decrease(var);    % MiniSat handling
```

Another major difference between VSIDS and AROFS is in variable activity decay handling. VSIDS periodically reduces the activity of all variables using a decay factor so as to give high priority to the variables that occur in recent conflict but this activity decay is not needed in AROFS. This is to prevent high priority to variables occurring in recent conflicts with minimal capacity to affect many clauses in the clause database while the variables that existed in some past conflicts might have the capacity to produce a major effect. The number of clauses a variable might affect can be decided based on its attackRelOccFactor. A variable (say V1) with low *arOccFactor* will be given higher priority than a variable (*say V2*) with high *arOccFactor* only if V1 occurs in many conflicts so that its activity surpasses V2's activity after several increments. Such behaviour partially promises to provide high priority to variables that occur in many recent conflicts.

The *arOccFactor* computation takes advantage of the CNF encoding since for an attack relationship, CNF encoding will create a clause with two negated variables. Hence, the *arOccFactor* for the two variables of those clauses will be incremented by some constant (say 0.01) for every such clause occurrence. The variable activity rescale handling of VSIDS is implemented in AROFS also. Hence, if the activity of a variable is greater than 1e100, the activity of all variables are rescaled by multiplying with 1e-100. The *arOccFactor* of a variable might take a big value if the number of arguments and their attack relations becomes large. This will result in frequent activity rescaling for fewer conflict occurrences. To avoid such a scenario, the *arOccFactor* of all the variables are reduced by dividing their value by the number of variables before beginning the solver process.

Algorithm 7: bumpAROccFactorInClauseAddition (clause)

```
  Global variable:   occIncfFactor;    % might take any value. Default: 0.01
if clause.size != 2 or clause.isLearnt then
    return;    % clause may be unit/learnt OR clause will not represent an attack relation
end
Lit firstLit = clause.lit[0];
Lit secLit = clause.lit[1];
if sign(firstLit) is True and sign(secLit) is True then
    Var firstVar = var(firstLit);
    Var secVar = var(secLit);
    arOccFactor(firstVar) += occIncfFactor;
    arOccFactor(secVar) += occIncfFactor;
end
```

4. Empirical Observations

4.1. Implementation details

The implementation of branching heuristics is done in C++ since it is the default MiniSat language. The graph generation code is implemented in python and DIMACS files are generated by using the attack set details of generated graph instances. Using python system commands, the compiled MiniSat solver (with default and with modified branching heuristics) are invoked with two arguments specifying the path to the input and output files. The time taken for this system call is measured and used for comparison. The MiniSat algorithm will write the satisfiability of the problem in the mentioned output file. The studies were performed on the Linux platform in a computer with Intel Dual core processor. The experiments were carried out for various values of n, p and q and for every combination of n, p and q, 25 graph instances were generated. The time taken report presented in the following subsection is based on the average time taken value for solving the generated 25 instances of AF. In the rest of this section, the usage of the term "time taken" refers to the average time taken value.

4.2. Comparison Report

The experiment clearly depicts a significant improvement in the performance of the solver for a given AF instance when the default branching is replaced with either BRmcha-BH3 branching or AROFS branching technique. To begin, the time taken between the solvers is negligible for small values of n since the solver with any branching reached an outcome in almost the same time for n = 128 and any p and q. In such scenarios, the difference in time taken was of the order of 0.01 second. This can be seen in the plot (B) of figure 4.1 where the maximum difference between the time taken by any two solvers is around 0.013s at q = 0.26. For higher values of n, the difference is no longer negligible as heavy performance variation was observed between the solvers. As shown in plot (A) of figure 4.1, for n = 256, the solver with VSIDS took more time than the solver with BRmcha-BH3 / AROFS and this difference proportionally increased for even higher values of n which can be confirmed based on the results of n = 512 shown in figure 4.2. However, a higher performance difference was noticed only between VSIDS and other branching techniques. Between the two modified branching systems, the difference was very small with BRmcha-BH3 branching had shown slightly higher performance than AROFS for higher values of n. Interestingly, any considerable performance gap between the solver with any branching was observed only for the lower values of q before the end of phase transition. When q increased beyond q* (phase transition threshold mentioned in (Gao, 2017)), the solver with all branching methods performed almost equally. In the two plots of figure 4.1 and in the figure 4.2, the time taken by solvers almost matched with one another from the value of q between 0.34 - 0.36 which is close to the theoretical threshold value q* for p = 0.75.

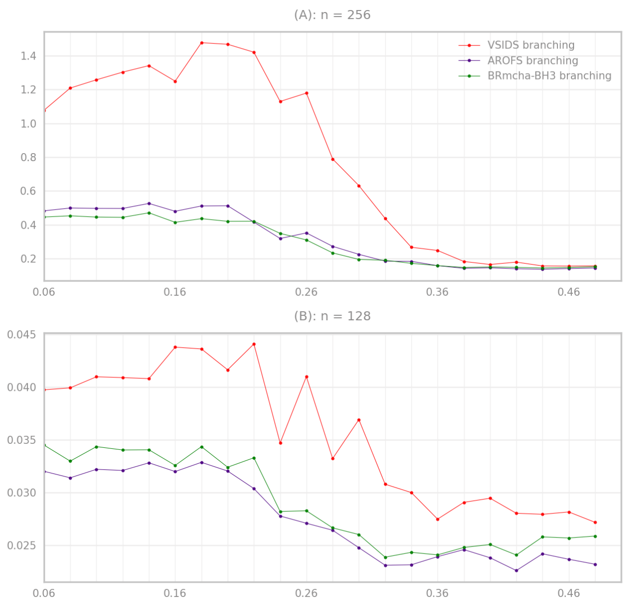

Figure 1: Time taken comparison between the solver with three branching heuristics for different values of n. The vertical axis and horizontal axis represent time taken in seconds and probability q respectively. Legends and probability p = 0.75 apply to both plots.

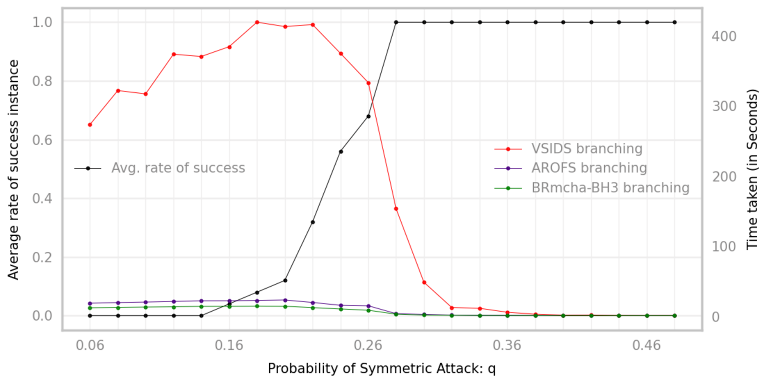

Figure 2: For n = 512 and p = 0.75, symmetric attack probability q is plotted against average rate of success instance and time taken by solver with different branching mentioned in legend.

The above result was observed for all values of n, p and q and is because of the increase in the fraction of *Yes instance* for which solver (with all branching heuristics) is able to solve the problem in nearly equal amount of time. Also, as per the empirical result of (Gao, 2017), it is well-established that the easy-hard-easy pattern of hardness can be seen when AF instances are generated by the D(n, p, q) model. Hence, because of the less hard AF instances after the phase transition, the solver is able to solve any instance easily with all branching techniques under study which is evident from the experimental results for $q >= 0.36$ shown in figure 4.2. With empirical studies of (Gao, 2017) into consideration, the hard instances of AF that occur during phase transition are solved more efficiently by the solver with AROFS / BRmcha-BH3 than the solver with VSIDS. To support this, VSIDS branching performed poorly for the value of q between 0.18 - 0.22 when n = 512 and p = 0.75 . Apparently, during this interval, the phase transition was observed with a sharp increase in the rate of *Yes instance* as seen in figure 4.2.

5. Conclusion and Future work

In an effort to examine the effectiveness of BRmcha-BH3 branching and to study their performance against various values of the structural properties of AF generated by D(n, p, q) random model, the BRmcha-BH3 technique's superiority over VSIDS in MiniSat is successfully ob served. As an important note, BRmcha-BH3 proved to be more effective than VSIDS for harder instances of AF. Apart from this, the newly introduced AROFS also showed a great potential in solving harder AF instances. As anticipated, it played a crucial role in MiniSat for reaching a result much faster than the MiniSat with VSIDS. On its comparison with BRmcha-BH3 branching system, both impacted the working of the solver to a greater extent and improved its performance with BRmcha-BH3 showed slightly better results than AROFS.

Based on this study carried out using MiniSat, we can notice the importance of branching heuristics in SAT solving and in solving AF using SAT techniques through reduction approach. With the details and knowledge on the implementation of various branching techniques studied in this paper, we can extend the work on modern SAT solvers like Glucose, `Maple_LCM_Dist_ChronoBT` (2018 SAT Competition winner (Heule et al., 2018)) and other similar solvers. This is primarily because of the fact that the MiniSat is not the state-of-the-art solver. Hence, it will be beneficial to conduct a detailed study on any of the above mentioned SAT solvers or mu-toksia which is a SAT-based AF solver that won the 2019 ICCMA competition (Niskanen & Järvisalo, 2019). As another important extension of this paper, it will be worthy to conduct an empirical work on how the newly introduced AROFS branching performs on other random models of AF and analyse its efficiency in solving practical AF problem instances.

References

Balyo, T., Heule, M., & Järvisalo, M. (2017). *Proceedings of SAT Competition 2017: Solver and Benchmark Descriptions* volume B-2017-1 of *Series of Publications B*. Finland: Department of Computer Science, University of Helsinki.

Baroni, P., & Giacomin, M. (2009). Semantics of abstract argument systems. In G. Simari, & I. Rahwan (Eds.), *Argumentation in Artificial Intelligence* (pp. 25–44). Boston, MA: Springer US. URL: `https://doi.org/10.1007/978-0-387-98197-0{_}2`. doi:10.1007/978-0-387-98197-0{_}2.

Besnard, P., & Doutre, S. (2004). Checking the acceptability of a set of arguments. In *NMR* (pp. 59–64). Citeseer volume 4.

Charwat, G., Dvořák, W., Gaggl, S. A., Wallner, J. P., & Woltran, S. (2015). Methods for solving reasoning problems in abstract argumentation – a survey. *Artificial Intelligence*, *220*, 28–63. URL: `https://www.sciencedirect.com/science/article/pii/S0004370214001404`. doi:`https://doi.org/10.1016/j.artint.2014.11.008`.

de la Vega, W. (1990). Kernels in random graphs. *Discrete Mathematics*, *82*, 213–217. URL: `https://www.sciencedirect.com/science/article/pii/0012365X9090327E`. doi:`https://doi.org/10.1016/0012-365X(90)90327-E`.

Dvořák, W., Järvisalo, M., Wallner, J. P., & Woltran, S. (2014). Complexity-sensitive decision procedures for abstract argumentation. *Artificial Intelligence*, *206*, 53–78. URL: `https://www.sciencedirect.com/science/article/pii/S0004370213001069`. doi:`https://doi.org/10.1016/j.artint.2013.10.001`.

Eén, N., & Sörensson, N. (2004). An extensible sat-solver. In E. Giunchiglia, & A. Tacchella (Eds.), *Theory and Applications of Satisfiability Testing* (pp. 502–518). Berlin, Heidelberg: Springer Berlin Heidelberg.

Gao, Y. (2017). A random model for argumentation framework: Phase transitions, empirical hardness, and heuristics. In *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI-17* (pp. 503–509). URL: `https://doi.org/10.24963/ijcai.2017/71`. doi:`10.24963/ijcai.2017/71`.

Heule, M., Järvisalo, M., & Suda, M. (Eds.) (2018). *Proceedings of SAT Competition 2018: Solver and Benchmark Descriptions* volume B-2018-1 of *Department of Computer Science Series of Publications B*. Finland: Department of Computer Science, University of Helsinki.

Liang, J. H., Ganesh, V., Zulkoski, E., Zaman, A., & Czarnecki, K. (2015). Understanding vsids branching heuristics in conflict-driven clause-learning sat solvers. `arXiv:1506.08905`.

Niskanen, A., & Järvisalo, M. (2019). μ-toksia (version 2019-10-31): Solver for static and dynamic argumentation frameworks.