Game-SAT: A Preliminary Report

Ling Zhao and Martin Müller

Department of Computing Science
University of Alberta
Edmonton, Canada T6G 2E8
{zhao, mmueller}@cs.ualberta.ca

Abstract. Game-SAT is a 2-player version of SAT where two players (MAX and MIN) play on a SAT instance by alternatively selecting a variable and assigning it a value true or false. MAX tries to make the formula true, while MIN tries to make it false. The Game-SAT problem is to determine the winner of a SAT instance under the rules above, assuming the perfect play by both players. The Game-SAT problem is PSPACE-complete, and it is derived from an application in adversarial planning. Although this problem is similar to QBF, the property of free variable selection can make it easier to convert other problems to this problem, compared to QBF with its fixed ordering.
We have developed a Game-SAT solver, Gasaso, that uses a combination of standard game tree search techniques, search methods that are well known in the SAT community, and specialized pruning techniques. We show empirically how these algorithms and methods perform in this new domain, and give evidence for the existence of phase transitions in this problem.
Keywords: Game-SAT, Heuristic search, Empirical study

1 Introduction

In this report, we introduce the Game-SAT problem, which is a 2-player version of SAT. Two players (MAX and MIN) play on a SAT instance by alternatively assigning a value of their choice, true or false, to an unused variable. MAX wins if and only if the formula is true after all variable values have been assigned. The Game-SAT problem is to determine the winner of a SAT instance assuming perfect play by both players under the rules above. We call a Game-SAT instance *satisfiable* if MAX wins the game, and it is *unsatisfiable* when MIN wins. There is no draw in Game-SAT.

For example, the formula (1) given below is a Game-SAT instance in CNF, and interestingly, the second player (whether MAX or MIN) can always win the game.

$$(V_1 \vee V_2) \wedge (\overline{V_1} \vee \overline{V_2}) \tag{1}$$

$$V_1 \wedge (V_2 \vee V_3) \wedge (\overline{V_3} \vee V_4) \tag{2}$$

Let's see how to play the game on formula (2). If MAX plays first, it has to assign V_1 true to avoid immediate loss, but MIN then can set V_2 to false, and force MAX to set V_3 to true. Finally, MIN can make the formula false by assigning V_4 false. As a result, MIN wins the game.

In [1], Schaefer introduced a large number of different games played on propositional formulas. Interestingly, Game-SAT itself is not among these games, but several simplified versions are. Schaefer's game G_{pos}(POS CNF) is a simplified version of Game-SAT where literals in each clause are non-negated. Schaefer's G_{ω}(CNF) is the version where the order of variables to be assigned is fixed. This game is equivalent to QBF. The Game-SAT problem is PSPACE-complete, since its special case G_{pos}(POS CNF) is PSPACE-complete [1].

Our interest in the Game-SAT problem originated from our efforts to develop a formal model for expressing the dependency of subgoals in adversarial planning. In this model, an atomic goal that can be achieved or foiled by one player is represented by a boolean variable. Complex goals are expressed as boolean formulas over these variables. The dependency between different subgoals is modeled by their shared variables.

Game-SAT shares many similarities with both SAT and QBF. If the two players in Game-SAT cooperate instead of competing with each other, then Game-SAT becomes SAT. Game-SAT has the same complexity as QBF, since they can be reduced to each other in polynomial time. Besides, many methods used in SAT or QBF solvers can be applied to Game-SAT.

However, Game-SAT also has its distinct characteristics. Compared to QBF, Game-SAT has the freedom to choose any unassigned variable, and this property leads to a much larger branching factor compared to QBF, at least in a brute-force search. We have not found any methods to convert between QBF and Game-SAT without introducing an enormous blow-up of the number of variables and clauses. This seems to indicate that Game-SAT can be a useful simple domain to which other problems can be reduced. Game-SAT is also a perfect-information two-player game, so search algorithms that have been successful in games research can be applied to this problem.

We employed both heuristic search algorithms and standard approaches effective in solving QBF or SAT problems, and incorporated them into Gasaso, a Game-SAT solver.

The remaining sections are organized as follows: Section 2 gives details about the solving methods we used. Section 3 talks about the experimental results regarding the improvements from those methods, and gives empirical evidence for the existence of phase transitions in Game-SAT. Finally, Section 4 contains conclusions and future work.

2 A Game-SAT Solver

Gasaso is a Game-SAT solver we built to experiment with different solving techniques. Gasaso is based on traditional minimax search, and incorporates many search enhancements that have been successfully applied in game-playing programs, SAT and QBF solvers. Gasaso is a complete Game-SAT solver - it can solve any given instance given enough time and resources.

We used randomly generated CNF formulas as our test cases. More information about the random instance generator is given in Section 3.1. Since any Game-SAT instance represented as CNF can be transformed to another Game-SAT instance in DNF, without loss of generality, we assume all instances men-

tioned in this paper are in CNF. For brevity, we also assume that literals within a clause do not share variables.

In the following, we give details about those search techniques that have improved the performance of our solver significantly, as well as those that did not work and our explanations.

2.1 Move ordering

At each move a Game-SAT player has to choose a variable and its boolean value. A good move ordering algorithm in Game-SAT leads to pruning large parts of the search tree, and helps to approach the minimal tree for solving the instance. However, since move ordering is called each time when there is a branch, it is also necessary to make it simple to avoid too much overhead.

In Game-SAT, a move is defined as assigning a literal the value true. The corresponding variable is set to true for a positive literal and to false for a negative literal. Initially, we use a simple frequency heuristic to weigh moves using the following formulas, where $Freq(l)$ is the number of occurrences of literal l in the instance. Moves are sorted and tried in descending order of their values.

$$\texttt{MAX to play: } Value(l) = Freq(l) - Freq(\bar{l})$$
$$\texttt{MIN to play: } Value(l) = Freq(\bar{l}) - Freq(l)$$

The history heuristic, originally invented for Chess programs [2] is based on the observation that a move frequently leading to the best minimax score in different game states is likely to be a good move in other game states as well. Let $HistFreq(l)$ be the number of times move l achieved the best minimax score for the current player. Moves are weighed according to the following formulas:

$$\texttt{MAX to play: } Value(l) = \alpha * HistFreq(l) + Freq(l) - Freq(\bar{l})$$
$$\texttt{MIN to play: } Value(l) = \alpha * HistFreq(l) + Freq(\bar{l}) - Freq(l)$$

The history heuristic performs best when it works together with frequency heuristic. In the formula, α is a large constant such that at the beginning, the solver mainly uses the frequency heuristic for move ordering. As the history of moves accumulates, the history heuristic quickly overrides the frequency heuristic. In our experiments, the default value is $\alpha = 256$.

2.2 Transposition table

Since in Game-SAT the order of variable selection is not fixed, the same partially assigned formula (corresponding to a *search state*) can be reached through different move sequences during the search. For example, a sequence of moves (V_1, $\overline{V_2}$, V_3) is equivalent to (V_3, $\overline{V_2}$, V_1). As a result, it is useful to determine these transpositions and to retrieve the previous result from a cache to avoid redundant computation.

2.3 Unit propagation

Unit propagation has been proven crucial in SAT solvers, and this technique is also applicable to Game-SAT. If an instance contains a single-literal clause $L_1 \wedge (\ldots)$ and it is MAX to play, MAX is forced to assign L_1 to true to avoid an immediate loss. If MIN has a big advantage and the move ordering is good, MIN may be able to follow up with another similar forcing move. In this way, a chain of unit propagations in Game-SAT can lead to a quick proof of a win for MIN.

2.4 Static evaluation

Static evaluation determines the result of a partially assigned instance without search. Gasaso uses the following static rules.

1. Single-literal clause for MIN to play: if the instance contains a single-literal clause, MIN has a sure win.
2. Double single-literal clause: if the instance has two single-literal clauses that share no variables, MIN wins the game. Note this case is also covered by the union of the previous rule and unit propagation.
3. Single-clause instance: if an instance contains only one clause, then its result can be determined by simply checking the size of the clause. This rule does not contribute any speedup as it only reduces the search depth by 1 and prunes an insignificant portion of searched space.

2.5 Instance simplification

Instances can be simplified by removing clauses that will not influence the game result. For example, a clause can be removed if one of its literals is assigned true.

Here is another important simplification rule. We call a variable *singleton* if it occurs only once in an instance. Any clause containing two singleton variables can be safely removed from the instance, since MAX has an easy strategy to make the clause true even as the second player, without changing the state of the rest of the instance.

2.6 Move reduction

Instance simplification may result in unassigned variables that no longer occur in the instance, and we call them *unused* variables. Unlike SAT, such unused variables can actually change the game results. If the rest of an instance is a second-player-win, having an unused variable to play changes the game into a first-player win. However, we do not need to generate all moves for unused variables, since they are only helpful for second-player-win scenarios. We used the following rules to reduce moves generated from these variables.

1. Moves from unused variables always have the lowest priority.
2. All moves generated from unused variables are equivalent, and thus we only need to try one of them.

3. An even number of unused variables is equivalent to no unused variables in the instance, and an odd number of unused variables is equivalent to 1 unused variable in the instance.

Another move reduction scheme is dominated move reduction. If the result of playing move B is at least as good as that of playing A, we say that move A is dominated by move B and prune it. For example, if an variable only occurs in positive form, then for MAX, setting it to true dominates setting it to false. Similar rules are used for MIN to play, and for variables occurring only in negative forms.

2.7 Iterative deepening

Although iterative-deepening [3] algorithms are very successful in many game-playing programs, iterative deepening depth-first search performs miserably in Game-SAT compared to depth-first search. We believe that our move ordering does well in guiding a depth-first search to quickly explore a small search tree to determine game results . The major disadvantage of iterative deepening is that it has to perform a complete search if the depth limit of an iteration is smaller than the minimum depth required to solve the instance. A huge number of nodes that need not be searched in the normal depth-first search must be explored.

3 Experimental results and analysis

This section presents experimental results on randomly generated instances, which empirically measure the effectiveness of each search enhancement. We also show the existence of phase transitions in Game-SAT random instances.

3.1 Experiment setup

We used the well-known fixed clause model to generate random Game-SAT instances. Each clause contains a fixed number of different variables, and each variable is negated with probability 1/2. No clauses are identical and all variables must occur at least once in the instance at the beginning.

The solver Gasaso was written in `C++`, and was compiled by `g++ 3.2.2`. All experiments were conducted on a Linux machine with a 1.6GHz Pentium-M CPU and 512M memory.

3.2 Comparisons of search methods and enhancements

We select the following 7 interesting search enhancements, and compare the performance difference to the full solver when each single enhancement is removed.

1. Frequency heuristic (FreqHeur)
2. History heuristic (HistHeur)
3. Transposition table (Cache)
4. Dominated move reduction (DomMove)
5. Unit propagation (UnitProp)

6. Static evaluation (StaticEval)
7. 2-singleton-literal clause removal (Singleton)

We used 1000 randomly generated instances as our test set, and we experimented with both cases when MAX plays first and when MIN plays first. Each instance has 40 variables ($n = 40$), 20 clauses ($l = 20$), and each clause has 3 variables ($h = 3$). During the experiments, we set the cache size to 1 megabytes, and limited the number of nodes searched to 10,000,000 per instance.

The experimental results clearly indicate that the 2-singleton-literal clause removal for instance simplification is crucial, and it is not uncommon to see with this method enabled, the time to solve instances can be improved by a factor of more than 1000. Without it, the solver cannot solve any instance within the node limit of this experiment. Therefore, the results for disabling this method are not shown in the following tables.

Method disabled	# unsolved	solving time	node count
None	0	1.18	622710
Cache	0	1.36	733058
Static evaluation	0	1.51	810889
Unit propagation	0	2.01	1118581
Dominated move	0	9.80	4996560
Frequency heuristic	1	11.34	5607285
History heuristic	3	20.41	10241900

Table 1. Performance of each method when MAX plays first
(1000 random instances, $n = 40$, $l = 20$, $h = 3$)

Method disabled	# unsolved	solving time	node count
None	0	15.18	7616448
Cache	0	17.33	8984572
Static evaluation	2	19.47	10508572
Unit propagation	1	29.91	16483991
Dominated move	2	77.80	37374162
Frequency heuristic	5	138.35	68793868
History heuristic	7	426.06	100371607

Table 2. Performance of each method when MIN plays first
(1000 random instances, $n = 40$, $l = 20$, $h = 3$)

In Tables 1 and 2, we show the number of unsolved instances and the time used and number of nodes searched for those instances solved by all 7 configurations for each case when MAX or MIN plays first. With the best configuration, our solver solves the 1000 instances in a total of about 14 seconds when MAX plays first and there are 917 satisfiable instances. 993 instances were solved by all configurations. When it is MIN to play first, it takes about 47 seconds to solve all instances, and there are 292 satisfiable instances. 997 instances were solved by all configurations. The search speed of our solver in the experiments was about 500,000 nodes per second.

From the tables, we can clearly see that among the remaining 6 techniques, heuristics for move ordering are most effective, and the history heuristic is superior to the frequency heuristic. Static evaluation and transposition table con-

tribute only small improvements in our experiments, although these methods were much more significant when incorporated into a bare-bone solver without any other enhancements. We also tried to increase the size of transposition table up to 128M on this test set, but it did not help much.

3.3 Phase transition

For the fixed clause model, Game-SAT random instances exhibit a sharp transition between all being unsatisfiable and all being satisfiable. We also observed an easy-hard-easy pattern in our experiments. Intuitively, for random CNF instances, the instance will have high probability to be satisfiable (a win for MAX) when there is a small number of clauses, while when the number of clauses increases, the probability for the instance to be unsatisfiable (a win for MIN) approaches 1.

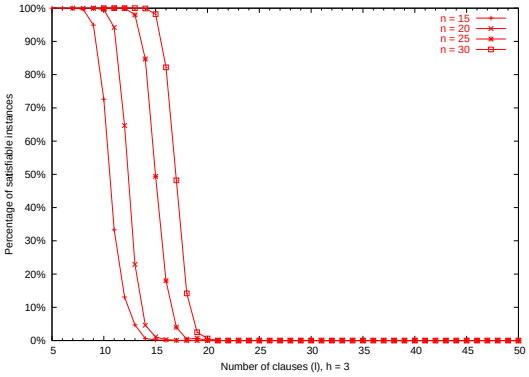

Fig. 1: Percentage of satisfiable instances (MAX plays first)

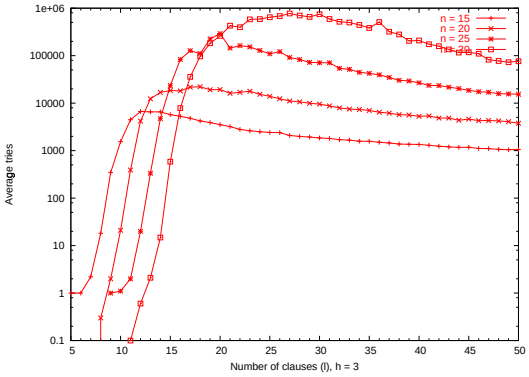

Fig. 2: Average number of nodes searched (MAX plays first)

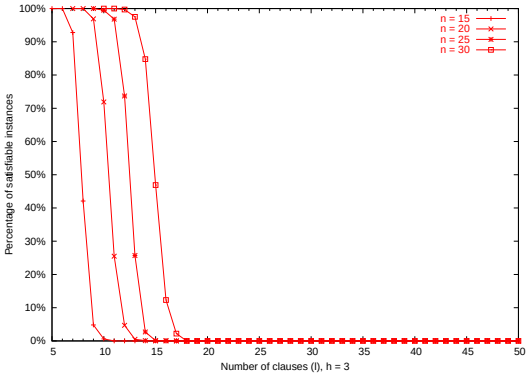

Fig. 3: Percentage of satisfiable instances (MIN plays first)

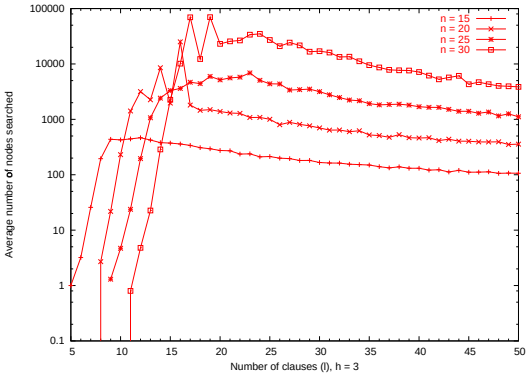

Fig. 4: Average number of nodes searched (MIN plays first)

This experiment uses different values for the number of variables ($n = 15$, 20, 25, 30, 35), and each clause contains 3 literals ($h = 3$). The number of clauses l varies from 10 to 50. We generated a set of 1000 CNF instances for each data point. The number of clauses must be at least $\lceil \frac{n}{l} \rceil$, since we require each variable

to occur at least once in each instance. The percentage of satisfiable instances and number of tries are shown in the following figures. The results for MIN playing first are presented in Figures 1-4.

The phase transition for satisfiability is very evident in Fig. 1 and Fig 3. However, the easy-hard-easy patterns for computational difficulties in Fig. 2 and Fig. 4 are very different from that in SAT problems. The curve from easy to hard is quite stiff, yet from hard to easy, the change is much less prominent. In addition, the peak in number of tries does not coincide with the phase transition for satisfiability. We also observed that statistically it is harder to solve when MAX plays first than when MIN plays first. We have not yet developed any analytical methods to predict the transitions, but it will be very interesting to explain why it happens.

4 Conclusions and future work

In this report, we introduced the Game-SAT problem which has applications in adversarial planning. Despite the similarity between Game-SAT and QBF, we argued that Game-SAT is an interesting problem on its own with different characteristics.

We developed a Game-Sat solver incorporating many search enhancements, and showed experimentally how they perform in this new domain. Since some of the methods were also useful for SAT and QBF solvers, we wish our experiments from the viewpoint of heuristic search in games would be of interest in the SAT and QBF communities, and most importantly, help the cross-fertilizing of the two areas.

Since this is a preliminary report, there is much room left for future work. For example, finding a manageable reduction from QBF to Game-SAT or vice versa would make the research for both problems converge. Our test cases are still limited to either simple examples converted from trivial games or randomly generated instances. We are looking for instances converted from real world applications or non-trivial games, so that Game-SAT can go beyond being only an abstract model. We also need to apply our work on Game-SAT to adversarial planning and see how it performs. As we mentioned before, it will be interesting to analyze in theory the phase transition in Game-SAT.

Acknowledgements

This work has been supported by the Alberta Informatics Circle of Research Excellence (iCORE) and the Alberta Ingenuity Fund.

References

1. T.J. Schaefer. On the complexity of some two-person perfect-information games. Journal of Computing System Science, 16:185–225, 1978.
2. J. Schaeffer. The history heuristic and the performance of Alpha-Beta enhancements , IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 11, no. 11, pp. 1203-1212, 1989.
3. R.E. Korf. Depth-First Iterative-Deepening: An optimal admissible tree search, Artificial Intelligence, 27, 97-109, 1985.