DPLL (aka insert great title here)

*Sophia Ardell*

*Optimization and Uncertainty*

*Spring 2016*

1 Description of Work

For the rest of the paper, assume that for any problem, there are *v* variables that can be assigned as negated or not negated, and *n* clauses that must be satisfied. Blocks are determined by the variables’ type (being choice or chance), where a variable that is the same type as the previous one is in the same block, but if it differs, it is the first variable in a new block.

Additionally, I will define several terms throughout my description. First, indicates a literal that is not negated, and indicates a negated literal. I will be using , or a specific number, as a subscript to indicate clause length. To represent the given problem space, I use the variable . The number of active clauses of length in the given problem space containing a specific literal that is not negated is given as , and for negated literals is given as .

Below is the pseudocode for general approach used for a given file. There is no output to the function as the program updates attributes of testProblem objects. All components will be expanded on in the following sections.

Algorithm 1a: Approach Given Specific File

**Input:** file name, use of unit clauses permitted, use of pure variables permitted, splitting heuristic type

**Output:** none

testProblem(file, unit, pure, split):

set relevant counters and booleans

start = current time

read(file) //initializes 2D array with data specific to *P*

create block index array

sat = solve(-*v*) //*-v* indicates no variable is selected, only time this occurs in code

matchSolution = difference between solution found indicated by sat and solution in file

totPosSplits = 2v – 1

percentSplits = 100 \* numSplits / totPosSplits

timeTaken = current time - start

Algorithm a: Overall Approach

1.1 Two-Dimensional Array

The majority of all data in this problem is stored in a two-dimensional array. The organization of this array is depicted below.

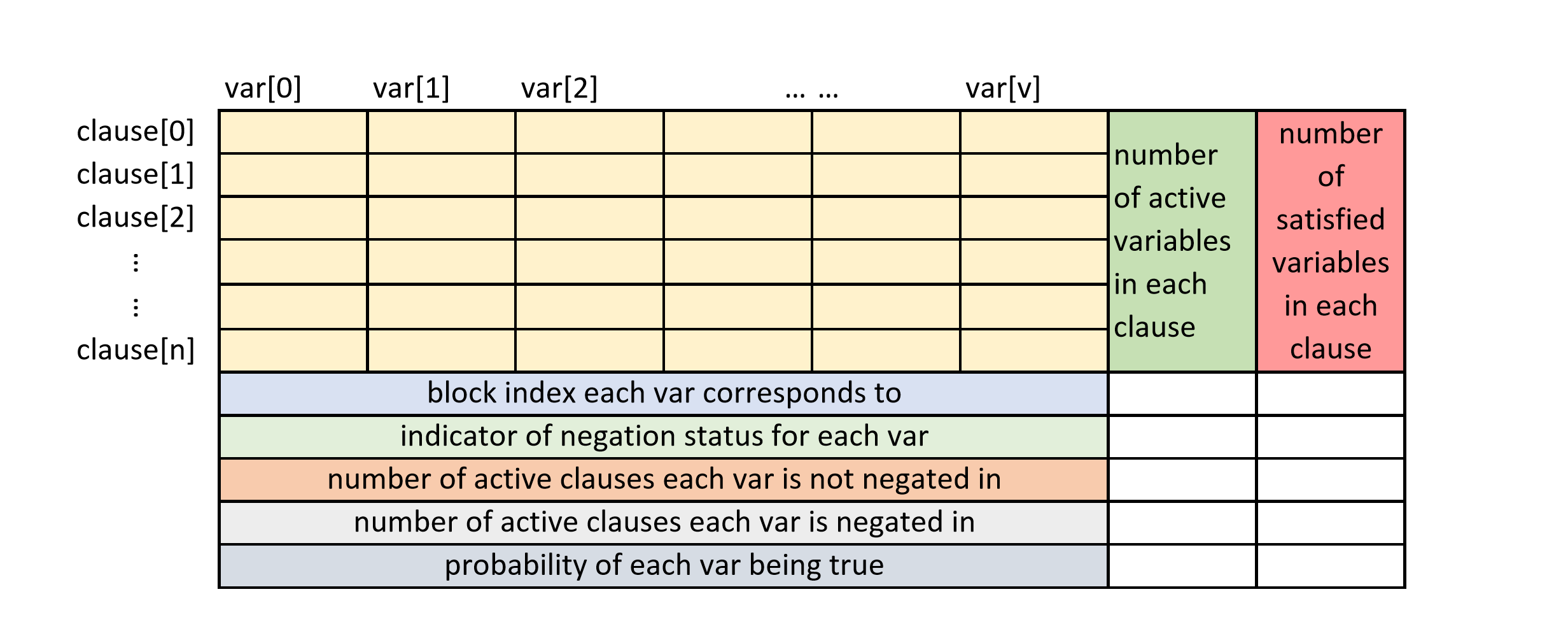


Figure : Illustration of Data Organization

Each of the first *v* columns corresponds to a variable. Each of the first *n* row corresponds to a clause. Every column up to *v* in each row indicates the variable assignment in that specific clause, such that a negated variable is represented by a -1, a not negated variable is represented by a 1, and a variable that is not present in the clause is represented by a 0. For the duration of the program, these indices are not altered. The remaining two columns in each row indicate how many variables that are in the clause have not been assigned a value yet (known as active variables), and how many variables in the clause have been satisfied. These counters are updated throughout the program. One way that failure of the current variable assignments is evaluated is if there is a clause present that has no active or satisfied variables, which would make the clause, and therefore the problem, unsatisfiable.

Below the clauses are five more rows, each of which hold information about each variable. The first row below the clauses indicates which block number each variable belongs to. The second row tracks if a variable has been assigned yet, with a 1 representing an assignment of not negation, -1 representing an assignment of negation, and 0 indicating that the variable is unassigned. The third row tracks how many active clauses the variable is not negated in. The fourth row similarly tracks how many active clauses the variable is negated in. Finally, the bottom row indicates the probability of each variable being true, which is a value between 0 and 1 if it is chance, and -1 if it is a choice. The first and fifth rows are not updated once the array is first set, and the second, third, and fourth rows are updated throughout the problem.

1.2 Block Index Array and Optimizing For-Loops

As variables for a specific problem are being read in, an array is initialized that is *v* long. The array holds the index of the first variable in each block. The first item in the array is always 0, as this will always be the starting index of the first block. The second item is initially assigned 1, as there must be at least 1 variable in the first block. Starting with the second variable in the list, the code compares each variable to the previous one. If they are the same type, the starting index of the next block is increased by 1. Otherwise, the index that is used to update the block array increases by one, and the starting index of the next block is set to be the current block’s starting index plus one.

This is very helpful to have, as there are many times throughout the code that you need to loop through all unassigned values in the current block. Before any of these functions are called, a naïve search is run that finds the first variable that is unassigned. The goal in each of these functions is to find the best variable in the current block to make the entire problem satisfied as quickly as possible. Therefore, the naïve solution is passed in as a parameter to see if heuristic can find a better solution. From this, you can find its block number in constant time from the 2D array, and then use this as an index to access the starting index of the next block in constant time. Since the naïve solution passed in implies that all preceding variables have been assigned, you now have both the start and end index of the loop in constant time, and you potentially avoid checking unnecessary values by starting at the naïve solution instead of at the beginning of its block. While not particularly helpful on small problems, this adjustment is useful for larger problems, especially ones with big blocks.

1.3 Solving Technique

Algorithm 1b: Solve - Recursive DPLL

**Input:** signedindex of current variable

**Output:** probability of satisfaction given negation assignment of variable

solve(varIndex):

**if** varIndex != -(total variables) **then** update array

check satisfaction

**if** unsatisfied **then** backtrack on array and **return** 0

**if** satisfied **then** backtrack on array and **return** 1

choose literal *v* using naïve approach

change *v* if UCP, PVE, or splitting heuristic find better *v* (descending order of import)

calculate probability of satisfaction for remainder of problem given:

**solve(***v***)**

**solve(***-v***)**

**if** varIndex != -(total variables) **then** backtrack on array

**return** probability of satisfaction for remainder of problem

Algorithm b: Recursive Implementation of DPLL

The algorithm focused on one function named “solve” to find the highest probability of satisfaction recursively. The solve function takes the column of the variable that is being assigned a value, which is positive or negative to indicate its negation status, as the only parameter. It then calls an update function which alters the 2D array based on this variable assignment. The array is then evaluated to see if it is satisfied. If the update was not able to finish, or the current variable assignment makes the problem unsatisfiable, then the program backtracks, removing all counters that were previously updated by the most recent variable assignment. It returns a probability of 0 satisfaction. If this is not the case, the code then checks to see if all of the clauses have been satisfied. If so, the program backtracks, and returns a probability of 1.

If the problem is neither not satisfiable or satisfied, then the program selects the next variable to assign a negation status to, the specifics of each selection method are expanded on in 1.4. First, the program runs the naïve algorithm, which finds the first variable that has not yet been assigned. If the problem permits unit clause propagation, it checks to see if any are possible and selects this variable along with its negation assignment if so. If there are no unit clauses, it goes through the same process for pure variable elimination. If there are no pure variables, then the program implements the splitting heuristic if there is one, or automatically selects the variable found by the naïve solution otherwise. In this last two cases, the variable does not automatically have an assignment.

The variable that is selected is the next variable that will be passed into a recursive call to the solve function. In the case of the variable not having a predetermined negation assignment, the remainder of the problem is evaluated with it being negated and not negated, which is done by calling the solve function on both options. If the variable had a predetermined negation assignment, the recursive call on the solve function is done once with this assignment. The probability of satisfaction is determined based on the variable type, whether one option is automatically guaranteed to be 0 (which is the case with single assignment variables determined by unit clause propagation), and is described below.

Relevant equations for calculating probability of satisfaction, where *w* is probability of a not negation assignment, *t* is the probability of satisfaction if the variable is not negated, and *f* is the probability of satisfaction if the variable is negated.

|  |  |
| --- | --- |
| Variable type | Probability Equation |
| Choice |  |
| Chance |  |
| Single Assignment Choice |  |
| Single Assignment Chance |  |

Table : Relevant equations for calculating satisfiability

After the actual success probability been evaluated, the program backtracks for that specific variable, then returns the probability of success.

1.4 Variable Selection

There are up to four steps for selecting a variable each step of the problem. First, a naïve search is run. This algorithm finds the first variable that is not assigned. Then, it checks to see if there are any unit clauses present in the current active clauses (which is expanded on in 1.3.1). If there are not, or the algorithm doesn’t allow for unit clause propagation, it checks for pure variables (expanded on in 1.3.2). If none are found or pure variable elimination isn’t allowed, then it checks for a better variable than the one selected by the naïve solution that is in the same block using a splitting heuristic, if it is permitted. Otherwise, it uses the variable selected by the naïve search.

In the case of unit clause propagation, a variable, along with its negation assignment, is selected. The remainder of the problem is only evaluated with the variable having this assignment, as testing the other possibility would be more computationally expensive and would definitely only provide unsatisfiable solutions. This is because assigning the variable an alternate negation assignment would guarantee that there is now at least one clause that is unsatisfied, which makes the problem unsatisfied.

In the case of pure variable elimination, the variable is still tested with both assignments. This is because even though the variable only appears one way in the active clauses, satisfaction is still possible without this assignment.

In the case of the naïve, pure, or heuristic selected variable, the remainder of the problem is evaluated first with it not negated, then with it negated, as both assignments could lead to the problem being satisfied.

1.4.1 Unit Clause Propagation

First, the algorithm checks to make sure that unit clause propagation is permitted. There is also a counter that tracks the current number of unit clauses in the problem. If either case is false, the algorithm returns that there are no unit clauses. Otherwise, the code loops through all clauses. For each clause, it checks if it specifically has more than one active variable and if it has any satisfied variables. If either are true, it goes to the next clause. If it finds a unit clause, the code then loops through all variables in the clause row, checking to see which one is still active and what sign it has. Once this is determined, the algorithm terminates, returning this information. If it reaches the end of the clauses and does not find a unit clause, it returns that no unit clauses exist.

1.4.2 Pure Variable Elimination

First, the algorithm checks to make sure pure variable elimination is permitted, and that the current block is made up of choice variables. If either case is false, the algorithm returns that there are no pure variables. Otherwise, the code loops from the variable found by the naïve solution to the end of the block. It checks first to see if the variable hasn’t been assigned yet, and then checks if it is a pure variable. If either of these cases are false, it continues to the next variable. If it has found a pure variable, it automatically returns that variable, along with its negation status (though this is not used, it could be potentially useful to later designs of solving implementation). If the code reaches the end of the variables and doesn’t find a pure variable, it returns that no pure variables exist.

1.5 Splitting Heuristics

Before describing the different splitting heuristics, I will clarify terms used below. In each case, the variable with the largest weight in that specific block will be selected without regard to negation, as both ultimately will be tested in order to find all potential solutions. The selected literal is given as .

The first two heuristics were designed to create as many unit clauses as possible, since if the splitting heuristic is deployed, then there were no unit clauses present. Variables that are given with required negation assignment lead to one less recursive call, as the alternate assignment would guarantee that the code is unsatisfied. This saves run time. In original testing of the code, the unit clause propagation ran much faster than pure variable elimination, so these heuristics were only designed around maximizing the likelihood of unit clauses. By selecting for variables that appear in the most short clauses, the clauses are satisfied or shortened. When clauses that are short decrease in size, they become unit clauses more quickly.

The last two heuristics were designed based on a similar concept as pure variable elimination. While the testing of pure variable elimination was quite slow, I wanted to test if I could improve upon the general concept behind it, where you choose which variable in a block to eliminate next based on how frequently it appears in clauses with a specific assignment. In this way, you satisfy the most clauses possible in a given iteration, which implies that the minimum amount of clauses left to satisfy remain, thus likely reducing the amount of assignments that need to be made to find satisfiability.

To see whether this algorithm worked because it was selecting the most frequent variable with a specific assignment, or just one of the most frequent variables, I made an algorithm that was identical except it only accounted for frequency of the variable in active clauses, regardless of the assignment. I anticipate that this version will also reduce run time, as it will partially achieve the goal in the above paragraph, and also shorten clauses. In this way, the maximum amount of shortened clauses is not satisfied necessarily, but the maximum amount of clauses that are either shortened or satisfied occur.

In most cases, I did not regard the negation of the variable in clauses. I assumed that selecting the best variable under any criteria would likely lead to the most clauses satisfied, and if the negation was different in the active clauses it occurred in, then the active clause would be shorter. This would therefore lead to a greater likelihood of the creation of unit clauses, saving run time in future updates. The final two splitting heuristics were designed to make clear whether or not this was a good assumption to make.

1.5.1 Power Weighted Clause Length Heuristic

This heuristic is designed to prioritize variables within the current block that are in the most active clauses that are short, regardless of the negation of the variable. As length of the active clauses that the variable appears in get longer, they contribute less to the weight.

1.5.2 Weighted Clause Length Heuristic

This code works to serve the same goals as the heuristic in 1.5.1. However, it only gives greater weight to the two shortest possible clause lengths, as I wanted to see if this would change the run time relative to the heuristic in 1.5.1. Clauses of length two were especially prioritized since variables that appeared in clauses of length two guarantees a unit clause in the following iteration if the clause is not satisfied.

1.5.3 Maximum Active Clauses Heuristic

This heuristic finds the variable in the given block that is in the most active clauses, regardless of the sign. In addition to the reasoning explained in 1.5, this heuristic is expected to perform well, since it chooses the variable that appears the most overall, which will satisfy the most clauses and/ or shorten the existing clauses.

1.5.4 Maximum Signed Active Clauses Heuristic

This heuristic finds the variable in the given block that appears the with a specific assignment in the most active clauses.

2 Experimental Methodology

I tested different seven different variable selection methods for DPLL: naïve, naïve with unit clause propagation, naïve with pure variable elimination, naïve with unit clause propagation and pure variable elimination, and all four splitting heuristics with unit clause propagation and pure variable elimination. All seven were run on 12 different problems, which were selected on the principle that it took the naïve approach at least three minutes to solve each one. This made for a total of 84 tests.

Since every variable selection method ultimately finds the same answer, each test was allowed to take as long as it needed to complete. The measure of success for each method was based on how long they took to solve the problem, both absolutely and relative to the naïve solution. The number of times a splitting heuristic was used instead of unit clause propagation and pure variable elimination was also used as a measure of success, though I anticipated that the success measured by this would correlate with amount of time taken.

All tests were run on a single machine with a 2.5GHz processor running OS 10586.218 for Windows 10 Home. The programs were written in Java.

**3 Results**

Tests were performed on 12 different problems with a varying arrangements of the variable type. Each had 125 clauses, which ranged from a length of 2 to 8 (the average length being approximately 5 for all cases), and had 36 variables. An algorithm’s success was measured by the time taken to find the solution.

**3.1 All Data**

Table : Summary of problems algorithms were tested on

A description of the problems the algorithms were tested on, along with the best and worst results. P indicates a probabilistic block; C indicates a choice block. PWCL stands for Power Weighted Clause Length, WCL stands for Weighted Clause Length, MSAC stands for Maximum Signed Active Clauses, and MAC stands for Maximum Active Clauses.

|  |  |  |  |
| --- | --- | --- | --- |
| File Name | Block Pattern | Best Performing Algorithm | Worst Performing Algorithm |
| e1 | C | (tie) PWCL and WCL | Naïve |
| e2 | C | PWCL | PVE |
| e3 | C | (tie) PWCL and WCL | Naïve |
| er3 | CP | (tie) PWCL and WCL | (tie) Naïve and PVE |
| r1 | P | (tie) PWCL and WCL | (tie) Naïve and PVE |
| re1 | PC | (tie) PWCL and WCL | PVE |
| re2 | PC | (tie) PWCL and WCL | (tie) Naïve and PVE |
| re3 | PC | (tie) PWCL and WCL | (tie) Naïve and PVE |
| rere2 | PCPC | (tie) PWCL and WCL | (tie) Naïve and PVE |
| ns1 | PC | (tie) PWCL, WCL, MSAC, MAC | (tie) Naïve and PVE |
| rere3 | PCPC | (tie) PWCL, WCL, MSAC, MAC, UCP&PVE, UCP | (tie) Naïve and PVE |
| rere1 | PCPC | (tie) PWCL and WCL | (tie) Naïve and PVE |

Table 3: Time taken by all algorithms on each problem

Table represents the amount of time taken by each algorithm on all problems, given in seconds. The gradient presented on the table goes from green to red represents relative speed from fast to slow.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | Naïve | Unit Clause Propagation (UCP) | Pure Variable Elimination (PVE) | UCP & PVE | Max. Active Clauses | Max. Signed Active Clauses | Weighted Clause Length | Power Weighted Clause Length |
| e1 | 269 | 17 | 266 | 17 | 6 | 7 | 0 | 0 |
| e2 | 348 | 24 | 352 | 24 | 5 | 7 | 1 | 0 |
| e3 | 972 | 61 | 335 | 64 | 15 | 19 | 1 | 1 |
| er3 | 242 | 18 | 242 | 18 | 7 | 8 | 4 | 4 |
| r1 | 908 | 39 | 907 | 39 | 12 | 16 | 1 | 1 |
| re1 | 586 | 19 | 592 | 19 | 10 | 12 | 6 | 6 |
| re2 | 492 | 24 | 492 | 24 | 11 | 12 | 6 | 6 |
| re3 | 282 | 19 | 285 | 19 | 11 | 11 | 7 | 7 |
| rere2 | 265 | 19 | 263 | 20 | 12 | 13 | 9 | 9 |
| ns1 | 199 | 6 | 200 | 6 | 1 | 2 | 1 | 1 |
| rere3 | 205 | 1 | 207 | 1 | 1 | 1 | 1 | 1 |
| rere1 | 158 | 14 | 159 | 14 | 11 | 11 | 8 | 8 |

Table represents the amount of unit clause propagations each algorithm did on each problem. The gradient goes from red to green, or least to most propagations.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | Naïve | UCP | PVE | UCP & PVE | Max. Active Clauses | Max. Signed Active Clauses | Weighted Clause Length | Power Weighted Clause Length |
| e1 | 0 | 1932162 | 0 | 1933212 | 659948 | 841447 | 60299 | 57060 |
| e2 | 0 | 2780797 | 0 | 2795646 | 526329 | 770435 | 67513 | 63969 |
| e3 | 0 | 6377869 | 0 | 6381118 | 1383868 | 1775438 | 84874 | 86748 |
| er3 | 0 | 1974904 | 0 | 1974945 | 788712 | 890435 | 374900 | 374148 |
| r1 | 0 | 3776607 | 0 | 3776607 | 1188664 | 1603807 | 112427 | 106902 |
| re1 | 0 | 2084365 | 0 | 2086410 | 1173539 | 1365857 | 648303 | 646802 |
| re2 | 0 | 2585360 | 0 | 2582669 | 1140680 | 1267138 | 596502 | 608428 |
| re3 | 0 | 2206739 | 0 | 2208811 | 1227921 | 1290678 | 791463 | 790698 |
| rere2 | 0 | 2087593 | 0 | 2087525 | 1252933 | 1323327 | 878841 | 876671 |
| ns1 | 0 | 705667 | 0 | 709723 | 196137 | 243997 | 112399 | 112729 |
| rere3 | 0 | 168157 | 0 | 166767 | 143077 | 145944 | 100749 | 100664 |
| rere1 | 0 | 1738894 | 0 | 1739446 | 1238701 | 1309133 | 880492 | 877158 |

Table 4: Unit Clause Propagations done by all algorithms on each problem

Table 5: Unit Clause Propagations done by all algorithms that used UCP in each problem

Identical to Table 4, except the functions that did not use UCP were removed as to better show the relative scale between those that did.



Table shows the average performance of each algorithm as ratio of the average performance of the naïve algorithm.

|  |  |  |
| --- | --- | --- |
|  | Average time ratio with naive | average split ratio with naive |
| Naïve | 1 | 1 |
| UCP | 0.05468 | 0.033074 |
| PVE | 0.948194 | 0.999849 |
| UCP & PVE | 0.055252 | 0.033078 |
| Max. Active Clauses | 0.02479 | 0.013914 |
| Max. Signed Active Clauses | 0.02782 | 0.015944 |
| Weighted Clause Length | 0.013607 | 0.005975 |
| Power Weighted Clause Length | 0.013368 | 0.005973 |

Table 6: Average algorithm performance relative to naïve performance

Points to discuss:

* Success is not much smaller of choosing by max active var vs. choosing by clause length
* Difference between power weighted clause and weighted for just two smallest ones
* How many UCP did PWCL and WCL get vs. MSAC and MAC
  + Did PWCL and WCL cause more UCP
  + Also examine relative total var splits

**3.2 Effect of Unit Clause Propagation**

It can be seen that the gradient in Table 3 is in a similar pattern in Table 4. Table 5 was created to show the similarity more clearly, as the inclusion of the 0 values caused by the two algorithms that did not use unit clause propagation greatly increased the range of the gradient and so the pattern was less obvious. From this similarity, one can conclude that there is a clear relationship between the amount of unit clause propagations occur and the speed at which the algorithm can complete.

While I had initially guessed that more unit clause propagations would lead to the most successful algorithms, it appears that the relationship is not that simple. As indicated by Table 2, the algorithms that had the most success were the Power Weighted Clause Length Heuristic and Weighted Clause Length Heuristic. Yet for algorithms that used Unit Clause Propagation, they generally used them the least. Meanwhile, the slowest run times in this category was Naïve with Unit Clause Propagation and Naïve with both Unit Clause Propagation and Pure Variable Elimination, which used the most Unit Clause Propagation split. This seems to indicate that having Unit Clause Propagation definitively helps improve algorithm run time, having a secondary selection method is key

3.3 Performance of Pure Variable Elimination

3.4 Signed vs. Unsigned Maximum Active Clauses

4 Conclusions

7. Results: A description and analysis of your results (see below for more details).

8. Further Work: What could be done to push this work a little farther?

9. Conclusions: A summary of your results.

For this assignment, I want you to do items 1, 5, 6, 7, and 9. I would see Section 5 as mostly a description of the splitting heuristics you devised and why you thought they would work (very important). Section 6 is more or less written for you, since I have prescribed the experiments to run (although you may have run other ones, which would be great). Section 7, one of the two most important sections (along with Section 5), is an analysis of whether the heuristics actually worked. Section 9 is what it says it is.

Your data is not self-explanatory. In Section 7, you should use graphs, tables, and/or charts to present your data and support your conclusions. But, this section should not be just a collection of captioned graphs, tables, and/or charts whose interpretation is “left as an exercise for the reader.” There should be text that explains your results and the claims in your text should refer to the graphs/table/charts in support of those claims. For example “As the table in Figure 1 shows, the eﬀect of...” All graphs/table/charts should be captioned. Any graphs should have a title, axis labels with units, and a legend if more than one series of values is being shown.