

A polynomial time algorithm for 3-SAT

Ortho Flint, Asanka Wickramasinghe, Jay Brasse, Chris Fowler

Abstract

In this paper, we provide a polynomial time (and space), algorithm that determines satisfiability of 3-SAT.

1 Introduction

The Boolean satisfiability problem (or SAT), was the first problem to be proven NP-complete by Cook in 1971. 3-SAT is one of Karp's original 21 NP-complete problems. At last count, there are over 3,000 NP-complete problems considered to be important. Any problem in NP can be presented as a SAT problem. A SAT problem can be transformed into a 3-SAT problem, where any increase in the number of clauses is a polynomial bound. We provide a deterministic polynomial time algorithm that decides satisfiability of 3-SAT. Proofs and code can be found at: polynomial3sat.org

2 Preliminaries and Definitions

Definition 2.1. *A 3-SAT is a collection of literals or variables (usually represented by integers), in groups called clauses where no clause has more than 3 literals, and at least one clause does have 3 literals. If it's possible to select exactly one literal from each clause such that no literal l , and its negation $\neg l$ (or denoted $\neg l$, meaning not l), appear in the collection of chosen literals, we say that the 3-SAT is satisfiable, otherwise we say it's unsatisfiable. For satisfiability, the collection of literals chosen is called a solution, which is also known as a truth assignment. Note that the size of a solution set is smaller than the size of the collection, if the collection had at least two clauses of which the same literal was chosen.*

It is important to note, that our definition of a solution (definition 2.10), is inextricably tied to our constructs for a collection of clauses.

When we think of literals (also called *atoms*), we can consider an edge joining two vertices, each with an associated literal, if and only if, it's not a literal and its negation. Under no circumstance would a literal and its negation be connected by an edge. There are also no edges between two literals from the same clause. So conceptually, there exist edges that connect every literal to every other literal with the two restrictions that were just stated. Then it follows, that a collection of literals for some solution is such that, a literal from each clause is connected to every other literal from that collection. In graph theory, such a graph is called a *complete* graph K_n , n being the number of vertices, which here it's also the number of clauses. We shall denote this graph as: K_C where c is the number of clauses.

Definition 2.2. *An edge-sequence is an ordered sequence with elements 1 and 0. The ordering is an ordering of the clauses, with indexing: $C_1, C_2, C_3, \dots, C_c$ where a position in the sequence having a 1 or 0 entry, is associated with the literal corresponding to that position. An edge-sequence I , for an edge with endpoints labelled x and y , where $x \neq -y$, the literals associated with the endpoints, is denoted by $I_{x,y}$. The endpoints must always be from different clauses. We call the positions in $I_{x,y}$ that correspond to a clause C_i the cell C_i . The cells C_j and C_k containing the endpoints, x and y for $I_{x,y}$ have only one entry that is 1 in the positions associated to x and y . When an edge-sequence is constructed, a given position in $I_{x,y}$ is 1 if the associated literal is not $-x$ or $-y$. The initial construction of $I_{x,y}$ is subject to certain rules defined in 2.6 and 2.7, which may produce more zero entries. Lastly, removing one or more cells from $I_{x,y}$ is again a (sub) edge-sequence, denoted by $I_{x,y}^*$, if the cells containing the endpoints for $I_{x,y}$ remain.*

Definition 2.3. *A loner cell contains just one 1 entry for some literal. And a loner clause contains just one literal.*

Definition 2.4. A vertex-sequence is an ordered sequence with elements 1 and 0. The ordering is an ordering of the clauses, with indexing: $C_1, C_2, C_3, \dots, C_c$ where a position in the sequence having a 1 or 0 entry, is associated with the literal corresponding to that position. A vertex-sequence V , for a vertex associated with literal x , is denoted by V_x . We call the positions in V_x that correspond to a clause C_i the cell C_i . The cell C_j containing the vertex x for V_x has only one entry that is 1 in the position associated to x . When a vertex-sequence is constructed, a given position in V_x is 1 if the associated literal is not $-x$. The initial construction of V_x is subject to certain rules defined in 2.6 and 2.7, which may produce more zero entries. Removing one or more cells from V_x is again a (sub) vertex-sequence, denoted by V_x^* , if the cell containing x remains.

Definition 2.5. When an entry 1 in an edge-sequence or a vertex-sequence, becomes zero, we call it a bit-change. If a bit-change has occurred in an edge-sequence or a vertex-sequence, we say the sequence has been refined, or a refinement has occurred. A zero entry never becomes a 1 entry.

It's worth noting here that if an edge-sequence $I_{a,b}$ has a zero entry in some position for a literal c , then there is no K_C , using literals a, b and c together. In fact, this is what a bit-change is documenting in an edge-sequence.

Definition 2.6. The loner cell rule, LCR, is that no negation of a literal belonging to a loner cell can exist in an edge-sequence or vertex-sequence. If such a scenario exists in an edge-sequence $I_{x,y}$ or a vertex-sequence V_x where z is the loner cell literal, then all positions associated with literal $-z$ incur a bit-change. If this action of a bit-change for $-z$, creates another loner cell where the negation of the literal in the newly created loner cell is still present in $I_{x,y}$ or V_x the action of a bit-change for the negation is repeated. Hence, to be LCR compliant may be recursive, but all refinements are permanent for any edge or vertex sequence.

LCR compliancy is determined for an edge-sequence $I_{x,y}$ or a vertex-sequence V_x if either $I_{x,y}$ or V_x is being constructed. LCR compliancy is determined after any intersection between two of more edge-sequences is performed. LCR compliancy is determined if an edge-sequence or vertex-sequence incurred any refinement.

Definition 2.7. *The K-rule is that no cell from an edge-sequence $I_{x,y}$ or a vertex-sequence V_x can have all zero entries. If this is the case, then $I_{x,y}$ or V_x , equals zero, and $I_{x,y}$ is removed from its S-set, or V_x is removed from the vertex-sequence table. Note that their respective removals, is a refinement.*

K-rule compliancy is determined for an edge-sequence $I_{x,y}$ or a vertex-sequence V_x , if either $I_{x,y}$ or V_x are being constructed. K-rule compliancy is determined after any intersection between two or more edge-sequences is performed. K-rule compliancy is determined if an edge-sequence or vertex-sequence incurred any refinement. And finally, the K-rule is violated if all the vertex-sequences associated with a clause, are zero. In such a case, it's reported that the 3-SAT is unsatisfiable.

Now, we must define what it means to take an intersection or union of two or more edge-sequences. No intersections or unions are taken with vertex-sequences.

Definition 2.8. *We take the intersection or union of two n length edge-sequences, A and B , by comparing position i of A and B , using the Boolean rules for intersections (denoted by \cap), and unions (denoted by \cup), for all positions, $i = 0, 1, 2, \dots, n-1$.*

Recall that the entry for position i of A and B , is either 1 or 0.

Then for an intersection, we have:

$$1_A \cap 0_B = 0_A \cap 1_B = 0_A \cap 0_B = 0. \text{ And } 1_A \cap 1_B = 1.$$

And for a union we have:

$$1_A \cup 0_B = 0_A \cup 1_B = 1_A \cup 1_B = 1. \text{ And } 0_A \cup 0_B = 0.$$

Definition 2.9. *An S-set is a collection of edge-sequences whose endpoints are from two clauses, C_i and C_j where $i \neq j$. The number of constructed edge-sequences to be an S-set is $|C_i||C_j|$ minus the non edge-sequences of the form: $I_{i,-i}$. For 3-SAT, there can be at most 9 edge-sequences in an S-set.*

Definition 2.10. *A solution for a collection of c clauses must have a corresponding collection of edge-sequences, for some K_C . More precisely, the intersection of all the edge-sequences together, for a K_C , does not equal zero. ie. A solution K_C exists if $\bigcap_{i,j} I_{i,j} \neq 0$, where i and j are every pair of endpoints from the collection of edges for a K_C . A K_P , $p < c$, exists if the set of all sub edge-sequences \mathcal{D} for K_P are such that the intersection of \mathcal{D} does not equal zero. ie. A K_P exists if $\bigcap_{i,j} I_{i,j}^* \neq 0$, where i and j are every pair of endpoints from the collection of edges for K_P . It is to be understood that an edge-sequence for a K_C or K_P , means the edge-sequence associated with an edge for a K_C or K_P .*

3 Description of the algorithm

In this section we describe the basic algorithm. The description of the algorithm will consist of describing pre-processing and how the ordered sequences are compared with each other and what actions are to be taken based on those comparisons.

Pre-processing

Pre-processing begins by building the first tables from a given DIMACS file. While doing this, we learn of redundancies that might as well be eliminated. However, it's worth noting that the algorithm to process a 3-SAT does not require any redundancies to be removed.

Definition 3.1. *Let \mathcal{T} be a given collection of clauses. We call a literal x , a pure literal, if its negation $-x$, does not appear in \mathcal{T} . ie. $\nexists -x$ in \mathcal{T} .*

Definition 3.2. *A clause of the form: $(l, -l, x)$, $(l, -l)$ or $(l, l, -l)$ is called a quantum clause. A quantum clause is a possible randomly generated clause.*

To start, we order the literals represented by integers, within each clause. If there exists among the given clauses, two or more clauses which are the same clause, but the literals appear in a different order, it will be discovered.

Only one copy of each unique clause is required to process a given 3-SAT. We learn if any clause has literal duplication: (a, a, a) or (a, b, b) or (a, a) . Randomly generated clauses could be of this form, which we reduce to: (a) , (a, b) and (a) , respectively. When we have ordered the clauses, we have also documented each literal and its negation, and to which clauses they are associated. At this point we will have discovered any *pure* literals. The clauses containing one or more *pure* literals, can be removed, where one of the *pures*, represents its clause. We can remove a clause with a *pure* because any solution found with the remaining clauses, can include all the *pures*, since no solution includes their negations. We will also discover any *quantum* clauses, and they can be removed because any solution for the remaining clauses can always add a literal from a *quantum*. For example, let a *quantum* clause be $q = (l, -l, x)$, and there is a solution with the remaining clauses which may use l or $-l$ or neither. For neither, l or $-l$ can be selected to be part of the solution.

Removing *pures* and *quantums* can be recursive, as it was for the example found in the paper at: <http://polynomial3sat.org>.

If there exists among the collection of clauses given, one or more clauses containing just one literal, we remove these clauses. Obviously, the single literal is the only literal to represent the clause for any solution. And since these literals must be in every solution, their negations are removed from the collection of clauses given. This action may also be recursive. If this action empties a clause of all its literals, then there is no solution for the collection of clauses given.

There is a possible redundancy that we ignore, which we call *Global Inclusion*. Suppose among the given clauses, there exist two clauses: $C_i = (a, b, x)$ and $C_j = (a, b)$. It's clear that if a solution exists, then either literal a or b must be part of that solution, thus C_i is redundant and could be removed. With respect to *Global Inclusion*, in context of processing a 3-SAT, suppose cell C_j instead has 1 entries for three literals a , b and z . Now, suppose after some processing, that the entry for z in cell C_j is zero, in all edge and vertex sequences. ie. $V_z = 0$. Then cell C_i can be removed from all sequences. And after all processing is complete, it can be determined if a or b will be representing clause C_i .

Construction of the edge and vertex sequences

Let $n \leq 3c$, where c is the number of remaining clauses and n is the sum of the sizes, for the c clauses. Then after pre-processing, the n vertex-sequences, grouped by clause association, are constructed first, as outlined in definition 2.4, and then *LCR* and *K*-rule is applied. It would be common practice to order the clauses, and the literals within each clause, and use this same ordering for both the edge and vertex sequences. After pre-processing, the remaining clauses not removed, have at most, 9 edge-sequences constructed from them pairwise. The edge-sequences are constructed as outlined in definition 2.2, and then *LCR* and *K*-rule is applied. Observe that the number of edge-sequences would be less than $\binom{n}{2}$. It must always be less than, because we did not subtract the over count of non-existent edges with i) both endpoints in the same clause or ii) the non edge-sequences between a literal and its negation. Of course, if each clause had just one literal and there was a solution, pre-processing would have presented the solution or pre-processing removed all literals from at least one clause, establishing unsatisfiability. Either way, no edge-sequences would have been constructed. Each pair of clauses from the collection of c clauses, forms an S -set. Thus, the number of S -sets is $\binom{c}{2}$, where any S -set contains at most, 9 edge-sequences. We shall denote an S -set with the indices of the two clauses used to construct its edge-sequences. ie. $S_{i,j}$ has edge-sequences whose endpoints are from clauses C_i and C_j .

The refinement rules refer to a bit-change occurring, an edge removed from its S -set, or a literal having a zero entry in every edge and vertex sequence, all being the result of the Comparing of S -sets. Given a collection of clauses, if any of these refinements occurred that were not the result of the Comparing of S -sets, it was the result of *LCR* and *K*-rule compliancy, while constructing the edge-sequences and vertex-sequences. We shall apply the actions outlined in the refinement rules even when constructing the vertex and edge sequences. The example (at polynomial3sat.org), demonstrated that certain edge-sequences were zero upon construction, as they failed *LCR* and *K*-rule compliancy. To be systematic, we would first construct all the vertex-sequences as described in definition 2.4. Then apply *LCR* and *K*-rule to all of the vertex-sequences. If the actions outlined in the refinement rules can be taken on any vertex-sequence, we do so. Next, we construct all the edge-sequences as described in definition 2.2. Then apply *LCR* and *K*-rule

to all of the edge-sequences. If the actions outlined in the refinement rules can be taken on any vertex or edge sequence, we do so. This process may be recursive where all refinements are permanent.

The Refinement Rules 1, 2, 3 and 4

When a refinement occurs and one of the rules 1, 2, 3 or 4 actions are to be applied to an edge-sequence or a vertex-sequence, *LCR* and *K*-rule compliancy must always be determined afterward. Again, the actions for the refinement rules could be recursive and all refinements are permanent.

Rule 1: a bit-change occurred

Let I_{a_i, b_j} where i denotes cell i , and j denotes cell j , be an edge-sequence where at least, one 1 entry for a position (occurrence) associated with literal c became zero. Then we change 1 entries to zero for all positions associated with c in I_{a_i, b_j} . If there was just one position for c , then no action is taken.

Now, we consider all edges $I_{a, b}$ that are not I_{a_i, b_j} . ie. at least one endpoint is not from either cell i or j . We will change a 1 to zero for every occurrence of a literal who had at least one occurrence become zero in I_{a_i, b_j} for all edge-sequences labelled $I_{a, b}$. The result is all $I_{a, b}$ which of course includes I_{a_i, b_j} have the same set of literals whose associated entries are zero.

Rule 2: the follow up to a bit-change

If $I_{a, b}$ has zero entries for c , then $I_{a, c}$ will have zero entries for b , and $I_{b, c}$ will have zero entries for a . All three edge-sequences are documenting the same fact, namely, that there is no solution with a , b and c together. So, if an edge-sequence $I_{a, b}$ for which at least one entry became zero, say for literal c , we apply rule 1. Then in $I_{b, c}$ and $I_{a, c}$ we make all 1 entries zero, that correspond to literals a and b , respectively.

Rule 3: $I_{x_i, y_j} = 0$, and was removed from its S -set $S_{i,j}$

Let I_{x_i, y_j} be an edge that was removed from S -set, $S_{i,j}$. The sub-subscripts for the edge-sequence I_{x_i, y_j} indicate that one endpoint is from cell i , and the other endpoint is from cell j . First, we remove all occurrences of I_{x_m, y_n} where $m+n \neq i+j$, from their respective S -sets.

Next, we update all the vertex-sequences for literals x and y by: All the vertex-sequences for x have all positions associated with y incur a bit-change. And all the vertex-sequences for y have all positions associated with x incur a bit-change.

Recall, that any kind of refinement requires testing LCR and K -rule compliancy, even on vertex-sequences. This in turn could cause the vertex-sequence to equal zero. In such a case, we apply all actions outlined in rule 4.

We also ensure that all refined vertex-sequences for a literal x , have the same set of literals whose associated entries are zero.

Now, all positions associated with y incur a bit-change, in $I_{x, \#}$ where $\#$ is any literal that forms an edge with x . Then all positions associated with x incur a bit-change, in $I_{\%, y}$ where $\%$ is any literal that forms an edge with y .

This whole process may cause another edge removal, in which case we apply rule 3, recursively if need be. This in turn could cause a vertex-sequence to equal zero. In such a case, we apply all actions outlined in rule 4.

Rule 4: a literal's vertex-sequence equals zero

Let a literal x be such that its vertex-sequence violates the K -rule. Then its vertex-sequence V_{x_i} , where x belongs to cell C_i is equal to zero.

If V_{x_i} is now zero, then all V_x vertex-sequences are now evaluated to be zero, and all of them are removed from the vertex-sequence table.

When a vertex-sequence equals zero we do: All positions associated with x incurs a bit-change in every vertex and edge sequence. This action in turn

causes all edge-sequences of the form: $I_{x,\#}$, where $\#$ is any literal that forms an edge with x , to equal zero.

This whole process may cause another edge-sequence (not of the form $I_{x,\#}$), to be removed, in which case we apply rule 3, recursively if need be. This in turn could cause a vertex-sequence to equal zero. In such a case, we apply all actions outlined here, in rule 4.

The Comparing of the S -sets

Essentially, the Comparing process is the algorithm. All data structures are simply updated based on the outcome of Comparing S -sets with one another.

Definition 3.3. *When every S -set has been Compared with every other S -set, we say that a **run** has been completed. If c clauses are considered, then there are $\binom{c}{2}$ S -sets, thus the number of S -set comparisons for a **run** is $\binom{c}{2} < c^4$.*

Definition 3.4. *A **round** is completed if the Comparing process stops because no refinement occurred during an entire **run**. We say that the S -sets are **equivalent** when a round is completed.*

We note that if the first round was not completed, it was the case that the vertex-sequences associated to a clause, were evaluated to be zero, so they were discarded. This violation of the K -rule stops all processing, as there is no solution for the collection of clauses given.

To Compare, we take two S -sets and **determine** if an edge-sequence $I_{x,y}$ from one of the S -sets, can be refined by a union of the intersections between $I_{x,y}$ with each of the edge-sequences, from the other S -set. Either $I_{x,y}$ the edge-sequence under determination, is refined or it remains the same. This is done for each edge-sequence from both of the S -sets, in the same manner. As a matter of practice, we determine in turn, each edge-sequence from one S -set first, and then we determine in turn, each edge-sequence from the other S -set. Below, we construct two S -sets to describe in more detail all the steps to be taken.

Let the S -set: $S_{i,j}$ contain 9 edge-sequences with endpoints from clauses: $C_i = (1, 2, 3)$ and $C_j = (a, b, c)$ giving: $I_{1,a}, I_{1,b}, I_{1,c}, I_{2,a}, I_{2,b}, I_{2,c}, I_{3,a}, I_{3,b}, I_{3,c}$

Let the S -set: $S_{k,l}$ contain 9 edge-sequences with endpoints from clauses: $C_k = (4, 5, 6)$ and $C_l = (d, e, f)$ giving: $I_{4,d}, I_{4,e}, I_{4,f}, I_{5,d}, I_{5,e}, I_{5,f}, I_{6,d}, I_{6,e}, I_{6,f}$

If $S_{i,j}$ and $S_{k,l}$ have 9 edge-sequences each, then there were no negations between the literals in C_i and C_j , or between the literals in C_k and C_l .

We say that **determining** all edge-sequences from one S -set first, is doing one direction denoted by: $S_{k,l} \xrightarrow{1} S_{i,j}$. And determining all edge-sequences once, for both S -sets, is doing both directions, denoted by: $S_{k,l} \xrightleftharpoons{2} S_{i,j}$

Suppose we **determine** $I_{4,d}$ of $S_{k,l}$ first. Then we have:

$$(I_{1,a} \cap I_{4,d}) \cup (I_{1,b} \cap I_{4,d}) \cup (I_{1,c} \cap I_{4,d}) \cup (I_{2,a} \cap I_{4,d}) \cup (I_{2,b} \cap I_{4,d}) \cup (I_{2,c} \cap I_{4,d}) \cup (I_{3,a} \cap I_{4,d}) \cup (I_{3,b} \cap I_{4,d}) \cup (I_{3,c} \cap I_{4,d}) \leq I_{4,d}$$

The Φ rule: Wrt. an intersection between $I_{4,d}$ and an edge-sequence from $S_{i,j}$, suppose the intersection resulted with occurrences of literal z and its negation $-z$, having zero entries, where at least one occurrence is not in any of the endpoint cells. Then the intersection can be evaluated to be zero, since if these two edge-sequences belonged to a solution together, they would also belong to two other solutions, one using z and another using $-z$, if both z and $-z$ had occurrences not in the endpoint cells.

The one efficiency present even in the original *naive* version, was eliminating any unnecessary intersections by one easy check. After an edge-sequence is selected for determination, say I_{x_r, y_s} where $x_r \in C_r$, $y_s \in C_s$, the edge-sequences from the other S -set in the Comparing, that do not have 1 entries for the endpoints x_r and y_s when intersected with I_{x_r, y_s} , will be zero. Recall, that the two cells containing the endpoints, only have a single 1 entry corresponding to the two endpoints' positions, in their respective cells. Thus, if the other edge-sequence does not have a 1 entry for those same positions, the intersection will be zero, due to K -rule violation. So, after the selection of an edge-sequence to be determined, we select the edge-sequences from the other S -set, if they have 1 entries in both positions corresponding to the endpoints

of I_{x_r, y_s} . Of course, we could also check to see if there are 1 entries in I_{x_r, y_s} corresponding to the endpoints of the other edge-sequence as well.

Let's suppose then, that every edge-sequence in $S_{i,j}$ above, did have 1 entries for both endpoints a_k and d_l of $I_{4,d}$. Now suppose 6 intersections become zero, after the intersections were taken and *LCR* and *K*-rule was applied to each intersection, and we now have:

$$0 \cup (I_{1,b} \cap I_{4,d}) \cup 0 \cup (I_{2,a} \cap I_{4,d}) \cup 0 \cup 0 \cup 0 \cup 0 \cup (I_{3,c} \cap I_{4,d}) \leq I_{4,d}$$

$$\text{which is equivalent to: } (I_{1,b} \cap I_{4,d}) \cup (I_{2,a} \cap I_{4,d}) \cup (I_{3,c} \cap I_{4,d}) \leq I_{4,d}$$

Now, we take their union. Of course, *LCR* and *K*-rule compliancy need not be checked for any union, since it would not have been possible to create a new loner cell scenario, nor a cell with all zero entries.

Then to complete the determination of $I_{4,d}$ we need to compare position by position, to see if $I_{4,d}$ has been refined.

More precisely, if we have: $(I_{1,b} \cap I_{4,d}) \cup (I_{2,a} \cap I_{4,d}) \cup (I_{3,c} \cap I_{4,d}) = I_{4,d}$, then $I_{4,d}$ is unchanged, and we move on to the next edge-sequence to be determined. Or instead, we have: $(I_{1,b} \cap I_{4,d}) \cup (I_{2,a} \cap I_{4,d}) \cup (I_{3,c} \cap I_{4,d}) < I_{4,d}$, then $I_{4,d}$ has been refined. We need to know which literals incurred a bit-change, and then we apply the appropriate actions of the refinements rules, and as always, followed by testing *LCR* and *K*-rule compliancy.

To summarize, an edge-sequence $I_{4,d}$ to be **determined**, has 4 possible scenarios.

- 1) $I_{4,d} \neq 0$, and is unchanged.
- 2) $I_{4,d} \neq 0$, and is refined. We determine which literals had a bit-change and follow all appropriate refinement rule actions, recursively if need be.
- 3) $I_{4,d} = 0$, because $(I_{1,b} \cap I_{4,d}) \cup (I_{2,a} \cap I_{4,d}) \cup (I_{3,c} \cap I_{4,d}) \leq I_{4,d}$ became zero after an appropriate refinement rule action was taken, and then *LCR* and *K*-rule was applied. In this case, edge-sequence $I_{4,d}$ is discarded, and the actions stated in refinement rule 3 are taken, recursively if need be.

4) If each intersection: $(I_{1,b} \cap I_{4,d})$, $(I_{2,a} \cap I_{4,d})$ and $(I_{3,c} \cap I_{4,d})$ had also been zero at the outset, then $I_{4,d} = 0$. And, as with 3), $I_{4,d}$ is discarded and the actions stated in refinement rule 3 are taken, recursively if need be. When an edge-sequence equals zero, it was the case that none of the edge-sequences from an S -set could be part of a solution with the edge-sequence that was being determined. And every solution has an edge-sequence in every S -set.

Recall definition 3.3, that after all pairs of S -sets have been compared with each other, a **run** has been completed, which is $\binom{c}{2} < c^4$, for c clauses. Another **run** will commence if any refinement occurred. Eventually, a **run** will incur no refinement, not even a bit-change, at which point a **round** has been completed, and the S -sets are said to be **equivalent** which means at least one solution exists for the given 3-SAT. Or, the algorithm stopped because unsatisfiability had been discovered during **round** 1. Discovering unsatisfiability is when every literal from some clause is such that their vertex-sequences equal zero.

Summary: Pre-processing begins with a DIMACS file submission, providing clauses, where each clause is at most size 3. After pre-processing, edge and vertex sequences are constructed from the remaining clauses. The edge-sequences are grouped in their respective S -sets and the vertex-sequences are grouped by their clause association in the vertex-sequence table. When the vertex and edge sequences are *LCR* and *K*-rule compliant, the Comparing process begins. The Comparing process stops if: i) a clause was such that all its literals' vertex-sequences are zero, where it's reported that the given collection of clauses has no solution. Or, ii) one or more **runs** take place, where the last **run** had no refinement. This signals the end of a round, and the S -sets are said to be **equivalent**, which implies satisfiability.

References

See paper with the proof of correctness and termination for references.