RESEARCH

A polynomial solution for the 3-SAT problem

Geraldine Reichard

Reichard G. A polynomial solution for the 3-SAT problem. J Pure Appl Math. 2024; 8(1): 01-07.

ABSTRACT

In this paper, an algorithm is presented, solving the 3-SAT problem in a polynomial runtime of $O(n^3)$, which implies P = NP.

The 3-SAT problem is about determining, whether or not a logical expression, consisting of clauses with up to 3 literals connected by OR-expressions, that are interconnected by AND-expressions, is satisfiable.

For the solution a new data structure, the 3-SAT graph, is introduced. It groups the clauses from a 3-SAT expression into coalitions, that contain all clauses with literals consisting of the same variables. The nodes of the graph represent the variables connecting the corresponding coalitions.

An algorithm R will be introduced, that identifies relations between clauses by transmitting markers, called upgrades, through the graph, making use of implications. The algorithm will start sequentially for every variable and create start upgrades, one for the variables negated and one for its non-negated literals. It will be shown, that the start upgrades have to be within a specific clause pattern, called edge pattern, to mark a beginning or ending of an unsatisfiable sequence.

It will be proven, that if after several execution steps of algorithm R, two corresponding start upgrades circle, then the expression is unsatisfiable and if no upgrade steps are possible anymore and the algorithm did not return unsatisfiable, the expression is satisfiable.

The algorithm R is similar to already existing solutions solving 2-SAT polynomial, also making use of implications using a graph.

Key words: *Polynomial; Mathematics; Differential calculus; Coalitions; Logic*

INTRODUCTION

The decision problem 3-SAT deals with the question, wether a boolean expression with 3 literals per clause, is satisfiable or not, meaning it exists an assignment of variables, that makes the boolean expression become true [1-3].

The expression consists of clauses, containing 3 literals at most. The literals within the clauses are connected with disjunctions and the clauses with conjunctions. For example the following expression e is a 3-SAT expression:

$$(x_1 \lor x_2 \lor \neg x_3) \land (\neg x_3 \lor x_4 \lor \neg x_5) \land (x_1 \lor \neg x_4) \land (\neg x_1)$$

According to the theorem from Cook 3-SAT is NP-hard [1]. A polynomial solution of 3-SAT would mean the that P = NP [4].

Until now many randomized and deterministic algorithms have been introduced that implement an efficient way to try out variable assignments. An example is the in 1960 introduced DPLL-Algorithmus [5]. This algorithm uses backtracking techniques. Another example is a deterministic algorithm introduced in 2002 by Dantsin et al [3]. This reaches for k-SAT a runtime of $(2 - 2 / (k + 1))^n$ and is based on local search. A similar randomized algorithm was introduced 1999 by Schöning [6].

All of these 3 or k-SAT algorithms are not polynomial and try to test out assignment most efficiently.

This is different with 2-SAT, which is solvable efficiently in polynomial runtime. An example is the algorithm by Krom introduced in 1960, which uses a technique to utilize implications between clauses [4]. If the negative and positive version of a literal appear in two different clauses, for example in the expression $(x_1 \lor x_2) \land (\neg x_2 \lor x_2)$, a third clause $(x_1 \lor x_3)$ is generated. An expression is unsatisfiable, if after a repetitive application of the formula does not contain the clause $(x_1 \lor x_1)$ and the clause $(\neg x_1 \lor \neg x_1)$. Is this the case the expression is called consistent. If all clauses containing the same variable are grouped together, the runtime is cubic ($O(n^3)$). Even and Shamir could reach a runtime of $O(n^2)$ with that method, by ordering the operations to be executed on the clauses [2]. They also introduced a backtracking algorithm that solves 2-SAT in linear time [2].

The technique introduced within the following paper uses a similar technique to Kroms solution of the 2-SAT problem.

As in Kroms algorithm this algorithm R, introduced in this paper, will first group the clauses to coalitions consisting of all clauses

Independent Researcher, Germany

Correspondence: Geraldine Reichard, Independent Researcher, Germany, e-mail: reichardgeraldine@yahoo.com

Received: 2 Jan, 2024, Manuscript No. puljpam-24-6968, Editor Assigned: 3 Jan, 2024, PreQC No. puljpam-24-6968 (PQ), Reviewed: 5 Jan, 2024, QC No. puljpam-24-6968 (Q), Revised: 7 Jan,2024, Manuscript No. puljpam-24-6968 (R), Published: 31 Jan, 2024, DOI:-10.37532/2752-8081.24.8(1).01-07

containing the same 3 variables and then apply a technique called upgrade to detect circles. This upgrade paths represent one or more executions of the implication-rule.

DEFINITIONS

First the 3-SAT graph and its components are defined. Let c be a 3-SAT clause, meaning it has exactly 3 literals connected with disjunctions, as described within Definition 1.

Definition 1
Let *c* be a 3-SAT clause. *c* consist of exactly 3 unique literals, that are connected by disjunctions. Each literal consists of a value of true or false and a corresponding variable.

We say, that these literals are within the same form, if Definition 1.2 applies.

Definition 1.2
Let l_1 and l_2 be literals in a 3-SAT expression e. l_1 and l_2 are within the same form, if they are within different clauses, but consist of the same variable and are both either negated or non-negated.

Now we can define a coalition as a set of unique clauses of a 3-SAT expression, that each contain 3 literals consisting of the same three variables. We also assume, that each clause of a coalition contains a different combination of literals and that, if the expression might contain 1-SAT or 2-SAT clauses, they are formed into equivalent 3-SAT clauses due to help variables as described within Definition 2.

Definition 2 – Coalition
Let $c = (l_1 \vee l_2 \vee l_3)$ be a unique clause within a 3-SAT expression *e*. Let the corresponding variable from l_1 *be* x_1, the variable from l_2 *be* x_2 and the variable from l_3 *be* x_3. Then $C_{(x_1, x_2 ,. x_3)} = \{c_{1, \ldots,} c_k\}$ with $k \leq 8 \in$ □ is the coalition, that contains *c*.

Let $c = (l_1 \vee l_2)$ be a unique 2-SAT clause within a 3-SAT expression *e*, with x_1 being the corresponding variable to l_1 and x_2 being the corresponding variable to l_2.

Then to all coalitions $C \in e$ containing x_1 and x_2 and *a* third variable $y \in e,$ clauses $c_1 = (l_1 \vee l_2 \vee l_3)$ and $c_2 = (l_1 \vee l_2 \vee \neg l_4)$ with $l_2 = y$ *and* $l_4 = \neg y_i$ will be added.

If there are no such coalitions, then *a* new coalition $C_{(x_1, x_2 ,. h_1)}$ will be created with $h_1 \notin e$, containing clauses $c_1 = (l_1 \vee l_2 \vee l_3)$ and $c_2 = (l_1 \vee l_2 \vee l_4)$ with $l_3 = h_1$ *and* $l_4 = \neg h_1$.

Let $c = (l_1)$ be a unique 1-SAT clause within *a* 3-SAT expression *e* with x_1 being the corresponding variable to l_1. Then for every coalition C, that contains x_1 and two other variables, four clauses will be added to C consisting of l_1 combined with and all four unique literal combinations out of the remaining variables. If there are no coalitions within the expression containing x_1, then a coalition $C_{(x_1, x_2 ,. h_1)}$ will be created containing all clauses with l_1 combined with all 4 combinations of literal assignments out of the help variables $h_1 \notin e$ *and* $h_2 \notin e$.

If a coalition contains all 8 possible clauses, then it is called full, which is explained within Definition 3. Also a full coalition makes an expression unsatisfiable, as stated within Theorem 1. This is because the expression demands for each possible literal assignment of the 3 variables also the inverse assignment.

Definition 3
A 3-SAT coalition is called full, if it contains 8 unique clauses consisting of all combinations of 3 literals.

A visualization of a full coalition is shown in Figure 1.

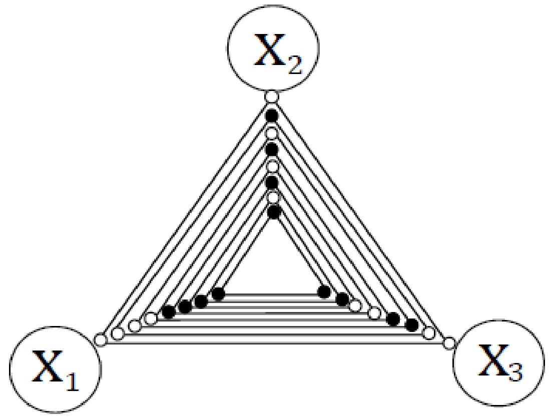

Figure 1) *A graphic representation of a full coalition* $C_{(x_1, x_2 ,. x_3).}$ *The black dots represent negated literals and the white dots non-negated literals. The triangles represent the clauses.*

Theorem 1
A 3-SAT expression e is unsatisfiable, if it contains at least one full coalition.

<u>Proof</u>
Let E_{sol} be a solution of the 3-SAT expression *e* that contains a full coalition $C_{(x_1, x_2 ,. x_3).}$ E_{sol} assigns every variable $x_1, \ldots, x_n$ from *e* a value of either true or false. Let $E_{sol(x_1, x_2 ,. x_3).} \subseteq E_{sol}$ be an assignment for the variables $x_1, x_2, . x_3$ from the full coalition $C_{(x_1, x_2 ,. x_3).}$ Per definition of $C_{(x_1, x_2 ,. x_3)}$ being full, meaning it contains all possible unique combinations of literals out of the three variables x_1, x_2 *and* x_3, $\forall$ possible assignments for $E_{sol(x_1, x_2 ,. x_3).}$ $\exists\, a$ clause c, that claims the inverse assignment for each literal and because of c being in E and E being defined as a 3-SAT expression, meaning its clauses are connected with conjunctions, that makes also E unsatisfiable.

THE 3-SAT GRAPH

The 3-SAT graph for an expression e consists of a set of coalitions C and a set of nodes V, one for every variable in the expression. Each node contains pointers to the coalitions, that contain the nodes corresponding variable. This is stated within Definition 4.

Definition 4 - 3-SAT graph
Let G_e be the corresponding graph to the 3-SAT expression e. Let $V_e = \{v_1, \ldots, v_n\}$ be the set of all nodes, that can be built out of the

variables $x_1,...,x_n$, that appear within expression e with $n \in \mathbb{N}$ and let $C_e = \{C_1,...,C_k\}$ be the set of all unique coalitions, that can be built out of clauses $c_1,...,c_j$ of expression e with $k, j \in \mathbb{N}$.

Each node $v_i \in V_e$ contains the corresponding variable x_i and a set of coalitions C_i, that contain variable x_i.

Each coalition C_i within the graph consists of the 3 corresponding nodes to its variables and its set of clauses.

Then the 3-SAT graph G for expression e is defined as $G_e = V_e \cup C_e$.

For a sample expression E the 3-SAT graph is visualized in the following picture (Figure 2).

$$E = (x_1 \vee \neg x_2 \vee x_3) \wedge (x_1 \vee \neg x_2 \vee \neg x_3) \wedge (x_1 \vee x_3 \vee x_4)$$
$$\wedge (\neg x_1 \vee x_3 \vee x_4) \wedge (\neg x_1 \vee \neg x_2) \wedge (x_1 \vee \neg x_2) \qquad (1)$$

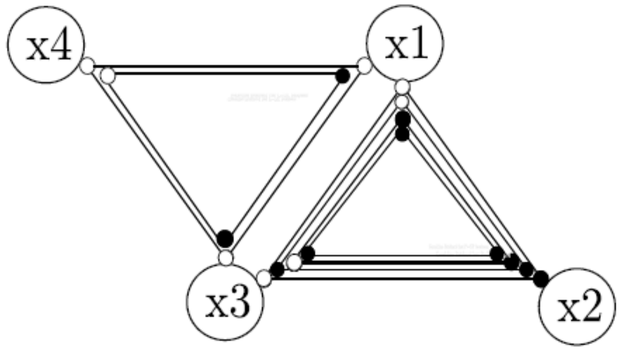

Figure 2) *Visualization of the 3-SAT-graph for expression E.*

Processing dependencies between coalitions
As proven within the last section, a single full coalition causes an expression e to be unsatisfiable. But also there is the possibility of dependencies between clauses of different coalitions.

For example two clauses from different coalitions within a 3-SAT expression $E = (x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_3 \vee \neg x_4 \vee x_5)$ could form a single 4-SAT clause $(x_1 \vee x_2 \vee \neg x_4 \vee x_5)$ per implication rule.

A series of executions of the implication rule could result in multiple 4-SAT clauses, that could add up to a potential full 4-SAT coalition, which would make the expression unsatisfiable without a 3-SAT expression being full within the initial graph. For example by reducing a full 4-SAT coalition to 3-SAT by using help variables, there would be no full 3-SAT clause within the expression, but it would still be unsatisfiable.

Also it could for example be possible, that multiple dependent 3-SAT clauses from different coalitions could form a new 3-SAT coalition with different variables by implications, that could be full.

Theorem 2 states, that only if there is either a full coalition within the expression already, or if there is a way to generate a full coalition by making use of implications within one or more execution steps of implication rules, then e is unsatisfiable and otherwise satisfiable. Later it will be also proven, that the algorithm R, introduced in this paper, is able to find this way of generating a full clause within a 3-SAT expression, if it is possible, in a polynomial runtime.

The algorithm R will not apply the implication rule to form the expression, but send markers called upgrades trough the 3-SAT graph to detect possible implications of clauses either within a coalitions or between different coalitions.

To do so it sends upgrades trough the 3-SAT graph, starting with a pair of two upgrades called start-upgrades.

Theorem 2
If there is no way to form a 3-SAT expression e per repetitive execution of implication rules in such a manner, that there is at least one full coalition, the expression e is satisfiable, and otherwise unsatisfiable.

Proof
Let e be a 3-SAT expression, that is unsatisfiable, but cannot be formed to a 3-SAT expression, that contains a full coalition via repetitive executions of implication rules. Let e' be an expression that contains all clauses from e and all possible clauses, that can be formulated out of e via applications of the implication rule. Because the implication rule only formulates arguments, that are already contained within e, that does not change the satisfiability of e. Each assignment e_{sol}, that satisfies e also satisfies e'.

Per Theorem 2 e' should also contain no full clause and per definition, there could not be new clauses added to e' by using implication rules.

Because e' being not full, for every coalition $C_{(x_1, x_2, x_3)}$ in e', there are still assignment $e'_{sol(x_1, x_2, x_3)}$ left for the variables within $C_{(x_1, x_2, x_3)}$ for that there is no clause $c \in C$, that does demand the inverse assignment to $e'_{sol(x_1, x_2, x_3)}$, which makes $e'_{sol(x_1, x_2, x_3)}$ satisfiable within $C_{(x_1, x_2, x_3)}$ and, because all implications are already priced in the graph of e', a union of possible solutions for every coalition $e'_{sol} = e'_{sol(C_1,...,C_n)}$ with $n \in \mathbb{N}$ would be a solution for e', meaning e' is satisfiable. That would also make e satisfiable, which is a contradiction to the unsatisfiability of e.

Let e be a 3-SAT expression, that is satisfiable, but can be formed to a 3-SAT expression e', that contains a full coalition. It is already proven, that a full coalition makes an expression unsatisfiable and so e' is unsatisfiable. Assuming the implication rules being correct, the satisfiability of e' is the same as the satisfiability of e, so e is also unsatisfiable, which is a contradiction to the assumption of e being satisfiable.

The algorithm R makes use of implications between coalitions without changing the graph. Instead, markers, called upgrades, will be processed throughout the graph.

Upgrades can be assigned to multiple literals and contain pointers to previous and following upgrades. They are defined within Definition 4.

Definition 4 - Upgrade
An upgrade u within the 3-SAT graph G of an expression e with a base literal l is defined a set of literals $L_{ul} = \{l_1,...,l_k\} \subseteq e$ with $k \in \mathbb{N}$.

The literal l is also stored with the set L_{ul} and is called the base literal of u. The variable of l is called the base variable and the value of l, either true or false, is called the base value. Also u contains pointers to one or more previous upgrades $U_{prev} = \{u_1, .., u_k\}$ and upgrade u can have following upgrades $U_{follow} = \{u_1, .., u_j\}$ with $k, j \in$.

Also each upgrade is assigned an upgrade path. In the case of the start upgrade, the path is empty.

Definition 4.1 - Upgrade Path

Let u be an upgrade. Then $P_u \subseteq G$, the upgrade path of u, is defined as the sequence of upgrades, that lead from one or more start upgrades to u. While there can also be multiple ways, that lead from the start upgrades to u, every upgrade lying on a possible way is integrated within the upgrade path.

Definition 4.2 - Corresponding upgrade

Every upgrade u is created with a corresponding upgrade u_c with the base literal of u, l_u being of the inverse form to the base literal of u_c, l_c, *and* $U_{C_{prev}} = U_{prev}$. If it is not possible to create a corresponding upgrade according to the algorithm R, then the upgrade step cannot be performed.

PATTERNS

Patterns are combinations of clauses, that transport upgrades to neighbor clauses in a certain manner or let them run into a circle. This would mean, that an upgrade is unsatisfiable on its path.

Start pattern

First, the algorithm R will search for start pattern. Every start pattern is also an edge pattern, but not every edge pattern has to be a start pattern. If an upgrade later comes into contact with an edge pattern, it is called an end pattern and the upgrade is then called circling.

The idea is, that every unsatisfiable implication sequence within the graph has to start with a pattern, that consists of all four literal-combinations for two variables. The idea of the proof will be, that if this would not by the case, then there are always two possibilities left on how to satisfy the sequence and if the two would have different implications with other clauses, this could not be called the start of the sequence.

To define an edge pattern, first a border pattern will be defined. This is half an edge pattern and consists out of a clause combination of at least two clauses with one literal of the same form and one literal inverse to the literal within the other clause. A border pattern, defined within Definition 5 with an example shown within Figure 3, consists of a clause combinations of at least two clauses, that contain, besides of the literals within the upgrade, one common and one disjunct literal.

An edge pattern is defined within Definition 6.

Definition 5 - Border pattern

Let u be an upgrade. We say, that u contains a border pattern, if L_u contains literals l_1 *and* l_2, that belong to clauses c_1 and c_2 and if c_1 contains also a literal l_{F_1}, that is of the same form as a literal l_{F_2} with $l_{F_1}, l_{F_2} \neq l_1, l_2$, called the common literals, and if c_1 contains a literal l_3, that is in the inverse form to a literal $l_4 \in c_2$ *with* $l_3, l_4 \neq l_1, l_2$. l_1 *and* l_2 are also called the connector literals. If R performs an upgrade via a border pattern, u will be led on via the common literals of the pattern to clauses of neighbor coalitions containing literals of the inverse form to the connector literal with respect to the rules of upgrade connection (Figure 3).

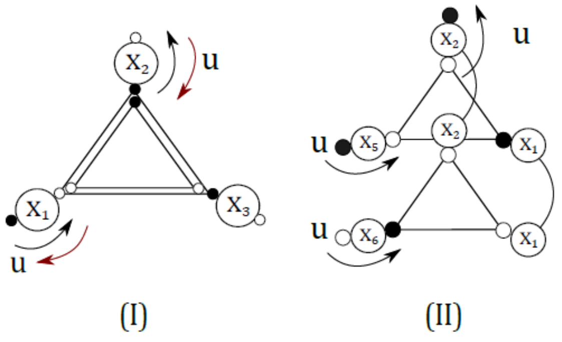

Figure 3) *Two examples of border patterns. In (I) the two clauses from the pattern are within the same coalition. Upgrades received by x_1 are led on via all literals of the inverse form to x_2. If the upgrade is received via x_2, then it can be led on via x_1. Also an upgrade could be received by two clauses from different coalitions and be sent via the common literal, as shown in (II). In both cases the patterns fulfill the requirement of containing one disjunct and one common literal.*

Border patterns transport upgrades on without changing them. An edge pattern consists out of two border patterns with inverse common literals, as defined within Definition 6.

First the algorithm will define initial starting pattern, which will always be an edge pattern, defined within Definition 7. An edge pattern is shown within Figure 4.

Definition 6 - Edge pattern

Let u be an upgrade. We say, that u contains an edge pattern, if L_u contains 2 border patterns, p_1 and p_2, and the common literals of p_1 are inverse to the common literals of p_2 and the other common literal of p_1 is equal to the common literal of p_2.

Definition 7 - Start upgrades

Let e be an edge pattern, then e can be chosen as a start pattern for R, where the 2 corresponding upgrades created out of the disjunct common literals u_1 and u_2 are called the start upgrades. They will divide at the inverse literals of their respective border pattern again into 2 corresponding upgrades each, having u_1 or u_2 as previous upgrades. So there are 4 start upgrades after running trough the start pattern (Figure 4).

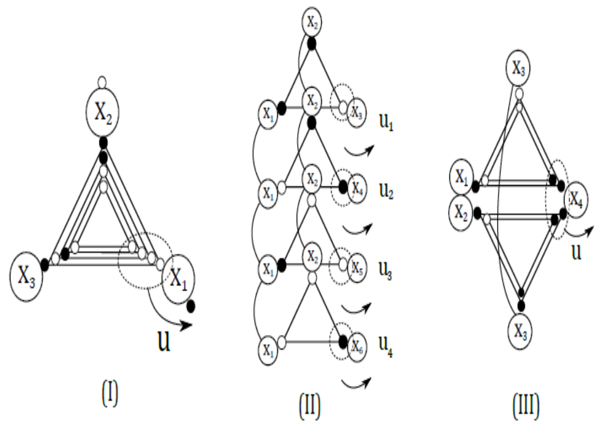

Figure 4) *The first image (I) shows a starting edge pattern within a single coalition. Initially for every combination out of the two literals, one upgrade is built, but they merge back at x_1 to one single start upgrade. The same happens in (III). In (II), all start upgrades are passed on individually via the third literals.*

Theoretically upgrades could merge together right at the beginning. The rule of upgrade connection explains, how upgrades connect, if two or more upgrades are sent to the same literal.

Definition 8: Rule of upgrade connection
If an upgrade u attempts to upgrade to a literal l, that already contains an upgrade u_c, then.

1. If u and u_c are on disjunct upgrade paths, then, if u_c is a leading upgrade, u will be added as a previous upgrade to u_c. If u_c is no leading upgrade, then there will be createda new leading upgrade u_l with u_c and u as previous upgrades.
2. If $P_{uc} \subseteq P_u$, then there is no need to perform the upgrade step.
3. If $P_u \subseteq P_{uc}$, then u will be assigned to l and be passed on.
4. If an upgrade u and its corresponding upgrade u attempt to upgrade to a literal l, then their common previous upgrade u_{prev} will be passed on to l. If all four start upgrades attempt to upgrade to the same variable, then all the upgrade paths integrate the upgrade is called neutral.

If an upgrade u is later led into an edge pattern, u it is called circling. If an upgrade circles, then the algorithm will start with the backtracking process. A small circle is sown within Figure 5 and defined within Definition 9.

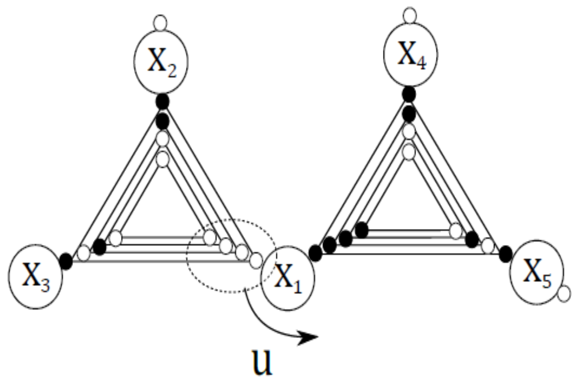

Figure 5) *A small upgrade circle. After the start upgrade got send via x_1, it reaches another edge pattern, that lets the upgrade circle. Because now every start upgrade circles, the expression is unsatisfiable.*

After the start upgrade is sent, within the next coalitions, if there would be another edge pattern, then the expression is called circling.

Definition 9 - Circling
An upgrade u is called circling, if all connector literals of the pattern also receives an upgrade u or from the path of u P_u. If two corresponding upgrades circle, then their common previous upgrades circle. If a neutral upgrade circles, then the expression is unsatisfiable.

Not only edge patterns can circle, but if an upgrade reaches a second edge pattern, then it immediately circles. An upgrade has to reach both connector literals of a border pattern to circle. If the upgrade is sent to a border pattern from the start upgrade, then it is led on via the common literal of the border pattern as in Figure 6.

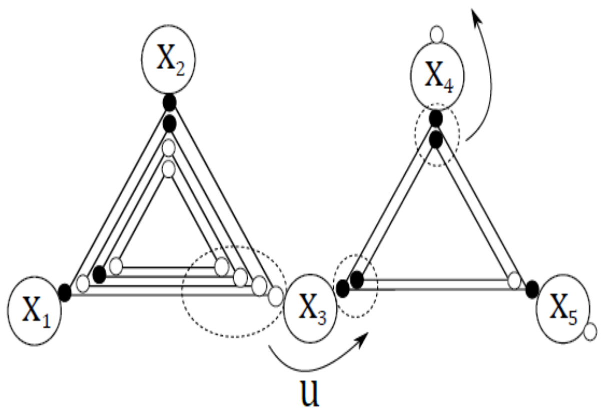

Figure 6) *After the start upgrade is sent to a border pattern from x_3 to $\neg x_3$ by making use of implications, it is led on from $\neg x_4$ to x_4.*

It is also possible, that the sender and receiver clause are connected via two literals. From the start pattern the upgrades will be sent on via the connector literals of the pat tern to all neighbor clauses with literals of the inverse form as stated in Definition 10 and shown within Figure 7.

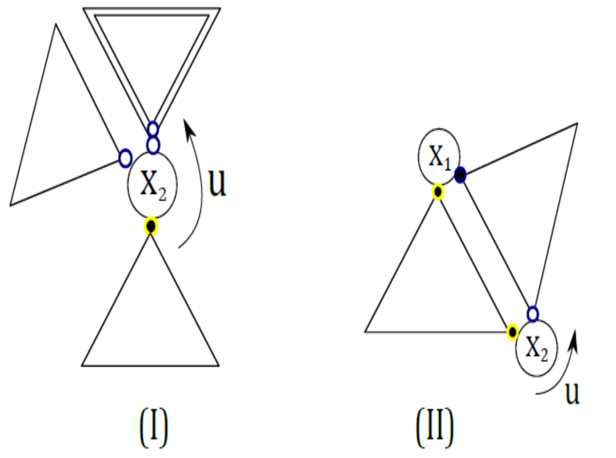

Figure 7) *An upgrade is sent to neighbor clauses. If the sender clause is connected via one inverse literal to all receiver clauses, that do not already have the upgrade, as shown in (I). If two clauses share two common variables, then the upgrade is only sent, if the clauses share besides off the sender and receiver inverse literal a literal of the same form with the second common variable, as shown in (II).*

Intermediate pattern
If an upgrade contains two clauses with only one common variable, one containing the negated form and one the non-negated form of literals with this variable, this is called an intermediate pattern. For all literals of the negated and for all literals of the non-negated form, a new following upgrade will be built and sent on via the connector literals (Figure 8).

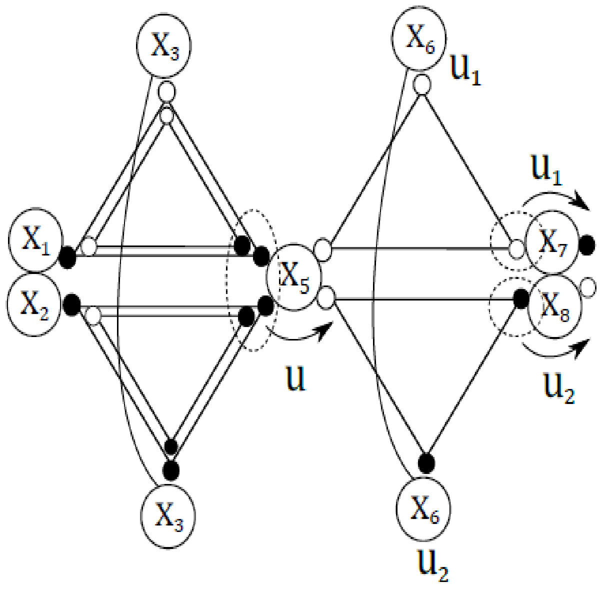

Figure 8) *From the starting edge pattern, an upgrade from $\neg x_5$ to x_5 is performed. Then there is an intermediate pattern, because the upgrade u contains non-upgraded clauses containing the true and false representation of x_6 and also the two clauses have different connector literals x_7 and x_8.*

Intermediate clause
If a clause does not belong to a pattern, it is called an intermediate clause.

Intermediate clauses have three connector literals to other coalitions. If an upgrade reaches an intermediate clause, it is not immediately sent on. Only if one other literal of the pattern also received the upgrade or an upgrade within the same path, it is sent on and only if all three literals received a respective upgrade, it circles, as defined within Definition 12 (Figure 9).

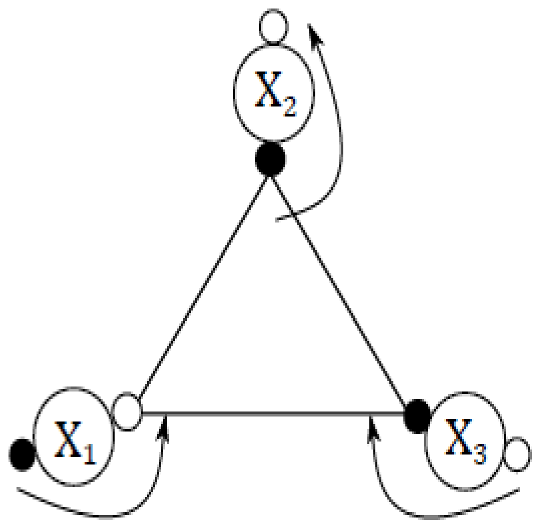

Figure 9) *An intermediate clause. Only if there are two incoming upgrades, for example at x_1 and $\neg x_3$, an upgrade is sent via $\neg x_2$.*

Definition 10 - Intermediate clause
If the literal set of an upgrade *u* L_u contains a literal from a clause *c*, that cannot be assigned to any kind of pattern, *c* is called an intermediate clause. *c* only passes on $_u$, if two literals of *c* received *u* or another upgrade from P_u and circles, only if all three literals received *u* or an upgrade from P_u.

THE ALGORITHM

Now let us put the algorithm together. Figure 10 shows a sample execution of R. If there is no full coalition within the graph making the expression unsatisfiable immediately, the algorithm R starts by searching for an edge pattern. If an edge pattern is identified, the start-upgrades are created and build the current set of upgrade U, that always contains the current tips of all upgrade paths and their set of literals. The start upgrades are sent on via the connector literals to all neighbor literals of the inverse form.

If one or more upgrades encounter another edge pattern or all connector literals of another pattern have been upgraded, the upgrade circles. If it is a neutral upgrade, the expression is unsatisfiable and if also the corresponding upgrade circles, all the common previous upgrades circle. If the start upgrades circle, then the algorithm returns unsatisfiable.

If there is no edge pattern or circle, then all upgrades with border patterns will be sent on to neighbor coalitions. If yes, then within the following neighbor clauses, the algorithm will go back to the step checking for edge patterns or circles (Figure 10).

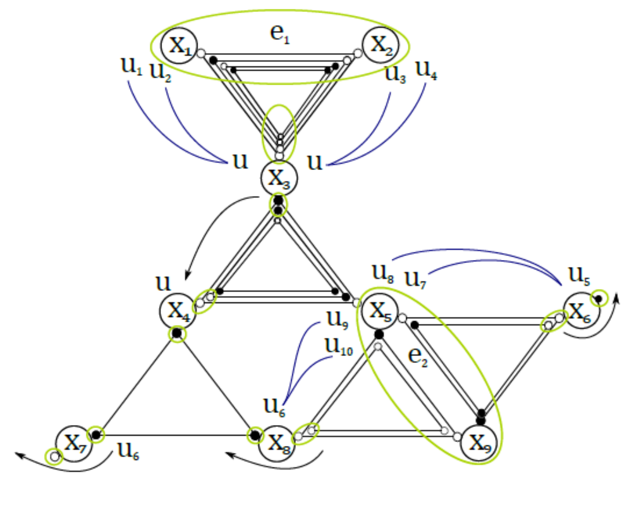

Figure 10) *A sample execution of algorithm R. Starting at the edge pattern e_1, 4 startupgrades u_1, u_4 are created. They merge together at x_3 building the neutral upgrade u. u is sent via a border pattern from.*

If there are no circles or border patterns within upgrades of the current set of upgrades U, intermediate patterns are evaluated. If an upgrade encounters an intermediate pattern, then 2 new following corresponding upgrades for the pattern are created and the upgrades are sent on via the connector literals. Also the algorithm now starts at the step evaluating circles or edge patterns again. When no upgrade steps are possible anymore and the start upgrades do not circle, the algorithm tries to start from another edge pattern, if there are edge patterns left without upgrades and repeats the execution steps. If afterwords still no corresponding start upgrades circle, then the algorithm returns satisfiable. As soon as corresponding start upgrades circle, R returns unsatisfiable.

CONCLUSIONS

The algorithm R creates a new data structure, the 3-SAT graph, which can more easily display dependencies between clauses of input expressions for the 3-SAT problem, by ordering them into coalitions and making use of dependencies between different coalitions. Also different kinds of clause patterns are defined. In that manner, a solution of the 3-SAT problem can be found without just trying out different assignments, but by making use of implications. That makes it possible to find a solution within polynomial runtime by sending markers, called upgrades, through the graph, creating the upgrade paths, storing a history of previous implications and dissolving it, if circles, marking contradicting variable assignments, are found. The approach is similar to existing graph algorithms solving 2-SAT in polynomial runtime already. The runtime of the algorithm R has a worst case complexity of $O(n^3)$. With some optimizations, it should be possible to reduce the runtime algorithm R runtime to be within $O(n^2)$.

REFERENCES

1. Cook SA. The complexity of theorem-proving procedures. Sym Theo Comp. 1971; 143-52.
2. Even S, Itai A, Shamir A. On the complexity of time table and multi-commodity flow problems. In16th annual symposium on foundations of computer science. 1975; 184-93.
3. Dantsin E, Goerdt A, Hirsch EA, et al. A deterministic (2−2/(k+ 1)) n algorithm for k-SAT based on local search. Theor Comp Sci. 2002;289(1):69-83.
4. Krom MR. The decision problem for a class of first-order formulas in which all disjunctions are binary. Math Log Quart. 1967;13(1):15-20.
5. Davis HPM. A probabilistic algorithm for k-sat and constraint satisfaction problems. Found Comp Sci. 1999;410-14.
6. Schoning U. A probabilistic algorithm for k-sat and constraint satisfaction problems. Found Comp Sci. 1999.