A polynomial solution for the 3-SAT problem

B.Sc. Geraldine Reichard
reichardgeraldine@yahoo.com

10.03.2023

Abstract

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

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.

The algorithm will eventually identify other kinds of pattern within the upgraded clauses. Depending on the pattern, the algorithm either sends the upgrades on through the graph, creates new following upgrades to extend the *upgrade path*, a subgraph storing all previous upgrades, or if all connector literals of a pattern have received upgrades of the same path or two corresponding following upgrades circle, marks the upgrade as *circling*. If an upgrade *circles*, then it is unsatisfiable on its path. 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.

1 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.
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:

$$(x_1 \vee x_2 \vee \neg x_3) \wedge (\neg x_3 \vee x_4 \vee \neg x_5) \wedge (x_1 \vee \neg x_4) \wedge (\neg x_1)$$

According to the theorem from Cook 3-SAT is NP-hard [1]. A polynomial solution of 3-SAT would mean the that $P = NP$.
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 $(x_1 \vee x_2) \wedge (\neg x_2 \vee x_3)$, a third clause $(x_1 \vee x_3)$ is generated. An expression is unsatisfiable, if after a repetitive application of the formula does not contain the clause $(x_1 \vee x_1)$ and the clause $(\neg x_1 \vee \neg x_1)$. 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 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.

2 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 $c = (l_1 \lor l_2 \lor l_3)$ 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 $\mathsf{C}_{(x_1,x_2,x_3)} = \{c_1, .., c_k\}$ with $k \leq 8 \in \mathbb{N}$ is the coalition, that contains c.*
Let $c = (l_1 \lor l_2)$ 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 $c_1 = (l_1 \lor l_2 \lor l_3)$ and $c_2 = (l_1 \lor l_2 \lor \neg l_4)$ with $l_3 = y$ 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 $c_1 = (l_1 \lor l_2 \lor l_3)$ and $c_2 = (l_1 \lor l_2 \lor l_4)$ 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 $C_{(x_1,h_1,h_2)}$ 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.

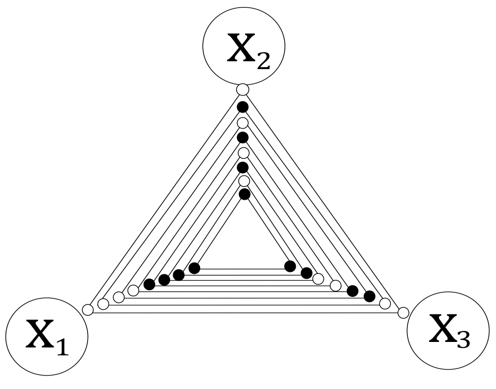

Figure 1: A graphic representation of a $full$ coalition $C_{(x_1,x_2,x_3)}$. The black dots represent negated literals and the white dots non-negated literals. The triangles represent the clauses.

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

proof. Let E_{sol} be a solution of the 3-SAT expression e that contains a $full$ coalition $\mathtt{C}_{(\mathtt{x}_1,\mathtt{x}_2,\mathtt{x}_3)}$. E_{sol} assigns every variable $x_1, ..., x_n$ 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 $\mathtt{C}_{(\mathtt{x}_1,\mathtt{x}_2,\mathtt{x}_3)}$. Per definition of $\mathtt{C}_{(\mathtt{x}_1,\mathtt{x}_2,\mathtt{x}_3)}$ 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. □

3 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 $V_e = \{v_1, .., v_n\}$ 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 $\mathtt{G}$ *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.

$$\begin{aligned} E = (x_1 \vee \neg x_2 \vee x_3) \wedge (x_1 \vee \neg x_2 \vee \neg x_3) \wedge (x_1 \vee x_3 \vee x_4) \\ \wedge(\neg x_1 \vee \neg x_3 \vee x_4) \wedge (\neg x_1 \vee \neg x_2) \wedge (x_1 \vee \neg x_2) \end{aligned} \tag{1}$$

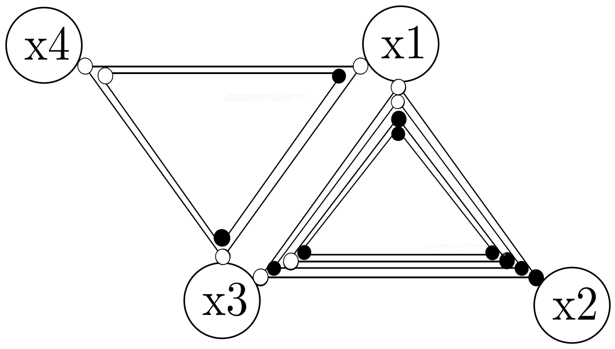

Figure 2: Visualization of the 3-SAT-graph for expression E.

3.1 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 $E = (x_1 \vee x_2 \vee x_3) \wedge (\neg x_3 \vee \neg x_4 \vee x_5)$ 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. $\Rightarrow$ 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 $é$ 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 $é$.
Per Theorem 2 $é$ should also contain no $full$ clause and per definition, there could not be new clauses added to $é$ by using implication rules.
Because $é$ being not full, for every coalition $C_{(x_1,x_2,x_3)}$ in $é$, there are still assignment $é_{sol(x_1,x_2,x_3)}$ 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 $é_{sol(x_1,x_2,x_3)}$, which makes $é_{sol(x_1,x_2,x_3)}$ satisfiable within $C_{(x_1,x_2,x_3)}$ and, because all implications are already priced in the graph of $é$, a union of possible solutions for every coalition $é_{sol} = é_{sol(C_1,...,C_n)}$ with $n \in \mathbb{N}$ would be a solution for $é$, meaning $é$ is satisfiable. That would also make e satisfiable, which is a contradiction to the unsatisfiability of e. $\square$
$\Leftarrow$ Let e be a 3-SAT expression, that is satisfiable, but can be formed to a 3-SAT expression $é$, that contains a $full$ coalition. It is already proven, that a full coalition makes an expression unsatisfiable and so $é$ is unsatisfiable. Assuming the implication rules being correct, the satisfiability of $é$ is the same as the satisfiability of e, so e is also unsatisfiable, which is a contradiction to the assumption of e being satisfiable. $\square$

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 $L_{u_l} = \{l_1, .., l_k\} \subseteq e$ with $k \in \mathbb{N}$. The literal l is also stored with the set L_{u_l} 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 $k, j \in \mathbb{N}$.*

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.*

3.2 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.*

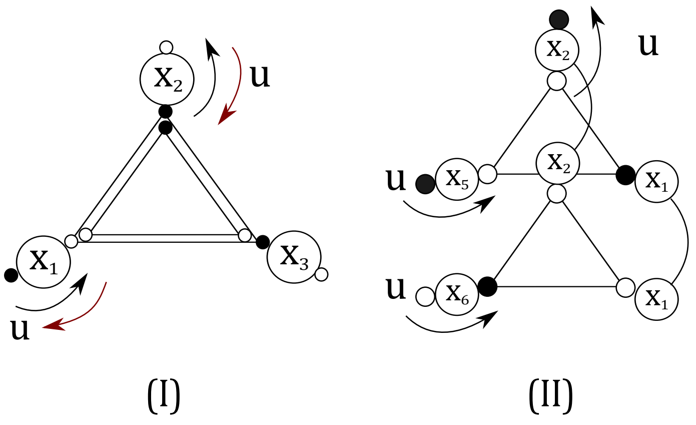

Figure 3: Two examples of border patterns. In (I) the two clauses from the pattern are within the same coalition. Upgrades received by x_1 are led on via all literals of the inverse form to x_2. If the upgrade is received via x_2, then it can be led on via x_1. Also an upgrade could be received by two 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.*

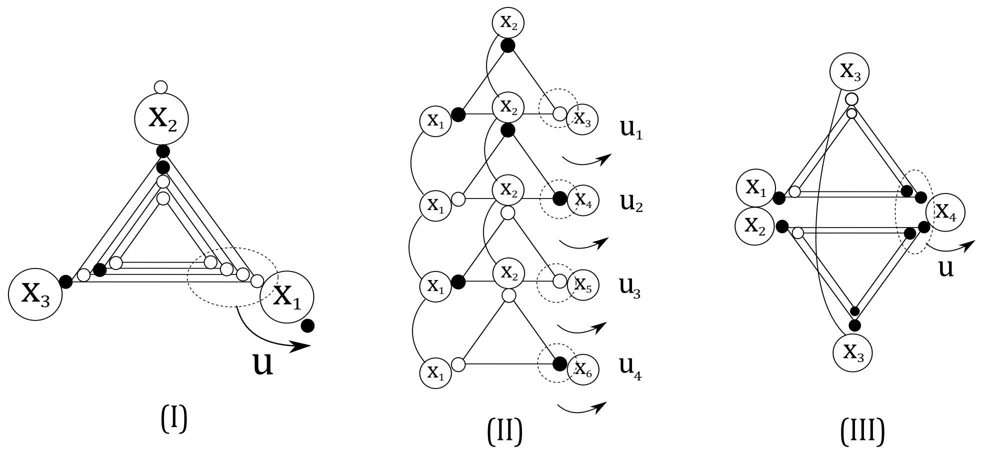

Figure 4: The first image (I) shows a starting edge pattern within a single coalition. Initially for every combination out of the two literals, one upgrade is built, but they merge back at x_1 to one single start upgrade. The same happens in (III). In (II), all start upgrades are passed on individually 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 created a new leading upgrade u_l with u_c and u as previous upgrades.
(2) If $P_{u_c} \subseteq P_u$, then there is no need to perform the upgrade step.
(3) If $P_u \subseteq P_{u_c}$, then u will be assigned to l and be passed on.
(4) If an upgrade u and its corresponding upgrade $\acute{u}$ 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.

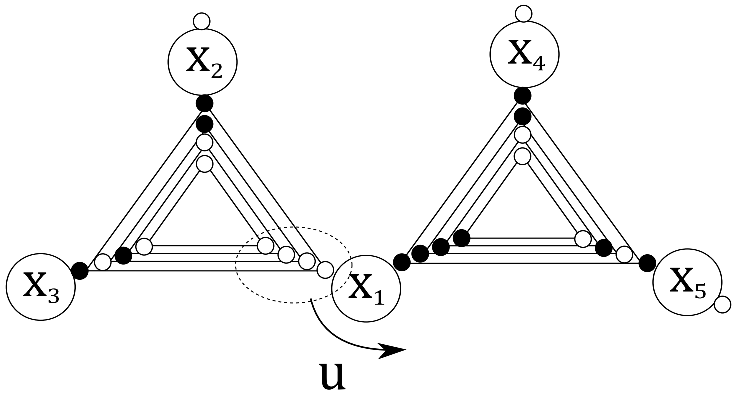

Figure 5: A small upgrade circle. After the start upgrade got send via x_1, it reaches another edge pattern, that lets the upgrade *circle*. Because now every start upgrade circles, the expression is unsatisfiable.

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.

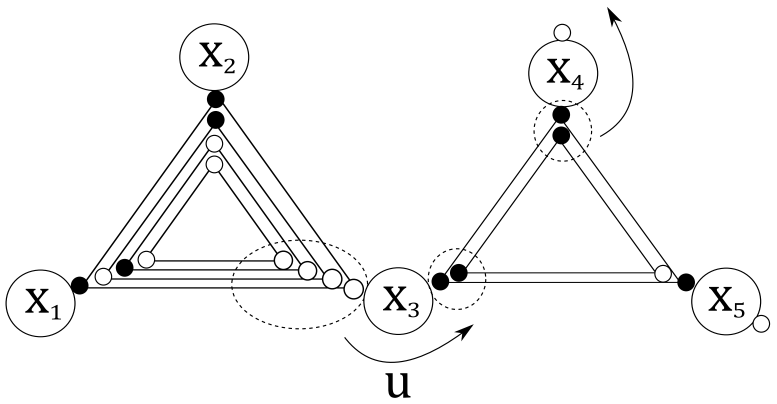

Figure 6: After the start upgrade is sent to a border pattern from x_3 to $\neg x_3$ by making use of implications, it is led on from $\neg x_4$ to x_4.

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 pattern to all neighbor clauses with literals of the inverse form as stated in Definition 10 and shown within Figure 7.

Definition 10 - Upgrade sending *Let u be an upgrade. If u is within a pattern p, that sends on u per definition of p via one or more literals $l_s \in L_s \subseteq L_u$ to the set of receiving literals within neighbor clauses $l_r \in L_r$ with $L_u \cap L_r = \emptyset$, then $\forall l_r \in L_r$ and $\forall l_s \in L_s : l_r = \neg l_s$, with respect to the rule of upgrade connection, if the sender and receiver clauses share one common variable. If they share two common variables v_1 and v_2, then the upgrade is only sent via v_1, if additional the literals with base variable v_2 in the sender and receiver clauses are of the same form.*

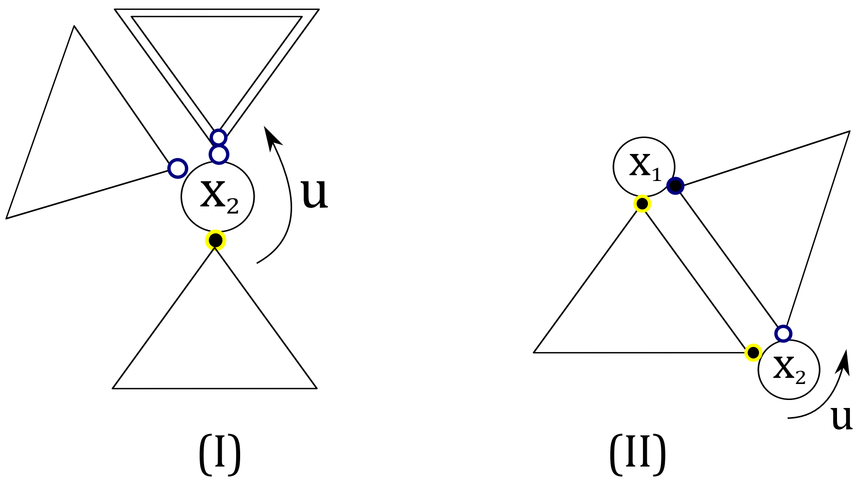

Figure 7: An upgrade is sent to neighbor clauses. If the sender clause is connected via one inverse literal to all receiver clauses, that do not already have the upgrade, as shown in (I). If two clauses share two common variables, then the upgrade is only sent, if the clauses share besides off the sender 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.

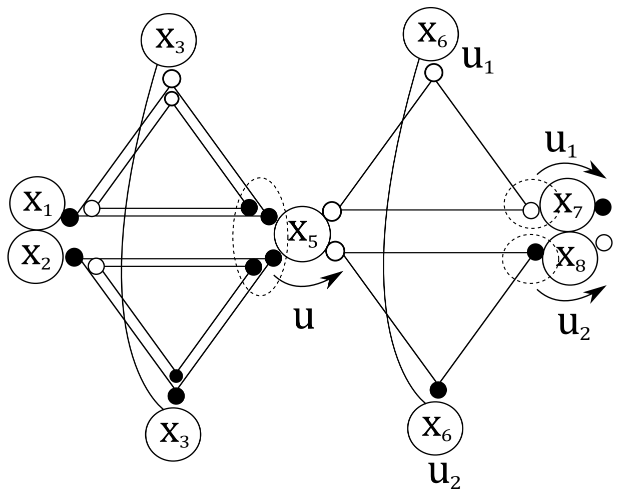

Figure 8: From the starting edge pattern, an upgrade from $\neg x_5$ to x_5 is performed. Then there is an intermediate pattern, because the upgrade u contains non-upgraded clauses containing the true and false representation of x_6 and also the two clauses have different connector literals x_7 and x_8.

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.

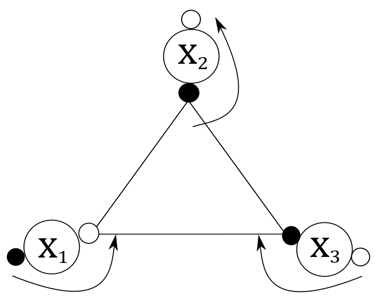

Figure 9: An intermediate clause. Only if there are two incoming upgrades, for example at x_1 and $\neg x_3$, an upgrade is sent via $\neg x_2$.

Definition 12 - 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.*

4 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.

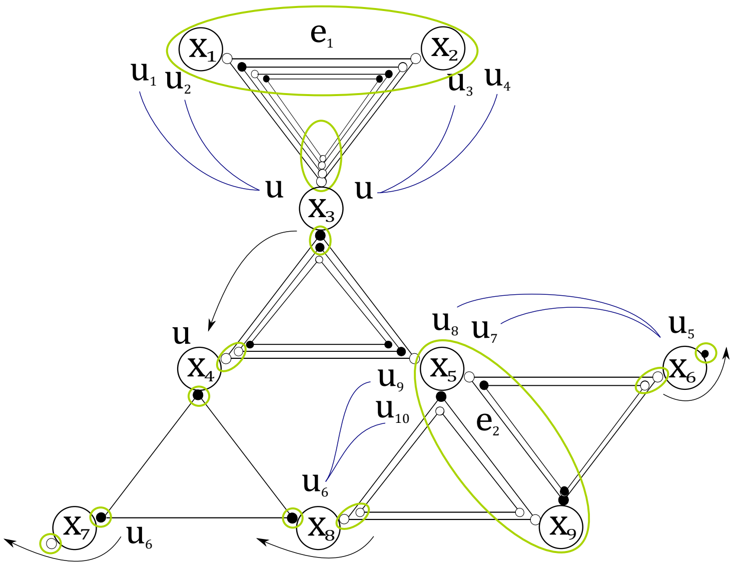

Figure 10: A sample execution of algorithm R. Starting at the edge pattern e_1, 4 start-upgrades $u_1, ..., u_4$ are created. They merge together at x_3 building the neutral upgrade u. u is sent via a border pattern from $\neg x_3$ via x_4 to $\neg x_4$ at the coalition $C_{(x_4,x_7,x_8)}$. There is another edge pattern e_2 creating the start upgrades $u_7, ..., u_{10}$. Two of the upgrades u_7 and u_8 merge into the following upgrade u_5 and u_9 and u_{10} merge into u_8, which is sent from x_8 to $\neg x_8$ to the intermediate clause $(\neg x_7 \vee \neg x_8 \vee \neg x_4)$ in $C_{(x_4,x_7,x_8)}$. Because u being a neutral upgrade and u_6 having the longer path according to the rules of upgrade connection and because the intermediate clause now received two upgrades from a common path, u_6 it is passed on from $\neg x_7$ to x_7.

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.

Now the algorithm can be described in pseudo-code:

```
Data: 3-SAT expression $e$
Result: Solution to the 3-SAT problem
Create the 3-SAT graph $G$ for $e$;
for Every edge pattern in $G$ do
    Create the 4 start-upgrade $U_s = u_{s_1}, ..., u_{s_4}$ and add $U_s$ to $U$;
    Send the start-upgrades via their corresponding connector literals;
    while Upgrade steps are possible do
        while There are edge patterns or circles in upgrades $u \in U$ do
            Mark their previous upgrades as circling;
            if Also the corresponding upgrade to one or more previous $u_p$ upgrades
              circles then
                Jump back to the step of marking all of their previous upgrades as
                  circling for each occurance;
            end
            if Also all 4 start upgrades circle then
                Return $unsatisfiable$;
            end
        end
        for All upgrade $u \in U$ containing a border pattern do
            Send $u$ on via the common variable of the border-patterns to neighbor
              coalitions, if possible;
            Jump back to the loop evaluating edge patterns;
        end
        if There is an intermediate pattern then
            Create for all upgrades $u$ with an intermediate pattern new upgrades
              $u_{true}$ and $u_{false}$ and send them on via the third literals of the clause;
            Replace $u$ in $U$ by $u_{true}$ and $u_{false}$;
            Jump back to the loop evaluating edge patterns;
        end
    end
end
Return $satisfiable$;
```

Algorithm 1: Algorithm R in pseudocode

5 Algorithm R is correct, within polynomial runtime and terminates

Now, that algorithm R is defined, its runtime, termination and correctness will be proven.

5.1 Proof of correctness

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

proof. Let E_{sol} be a solution for the 3-SAT expression e that contains a $full$ coalition $\mathtt{C}_{(\mathtt{x_1,x_2,x_3})}$. E_{sol} assigns every variable $x_1, ..., x_n$ 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 $\mathtt{C}_{(\mathtt{x_1,x_2,x_3})}$. Per definition of $\mathtt{C}_{(\mathtt{x_1,x_2,x_3})}$ 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. □

Theorem 2 *The algorithm* R *returns the correct result of the decision problem for every graph G.*

First it is proven, that for each different pattern, the algorithm applies the correct rules due to propositional logic.

Lemma 2.1 - intermediate pattern *If an upgrade u contains an intermediate pattern, then the algorithm* R *applies the implication rule correctly.*

proof. Let u be an upgrade. If u contains an intermediate pattern, it means, that the upgrade contains two or more literals of unique intermediate clauses containing the negated and non-negated form of a literal l, that is not yet on the upgrade path of u. Let this two clauses be c_1 and c_2 with $c_1 = (l_u \vee l_2 \vee l_3)$ and $c_2 = (l_{u2} \vee \neg l_2 \vee l_4)$ and let l_u and l_{u_2} be literals, that are already within the upgrade u, and l_2, l_3 and l_4 be literals, that are not yet within u. R would create two new upgrades. Let them be u_{2_t} for the non-negated literal l_2 and u_{2_f} for the negated literal $\neg l_2$. The upgrades are correspondent to each other and both have u as a previous upgrade. Afterwards R will sent on the upgrades via l_3 and l_4 to neighbor clauses, if there are such clauses with literals $\neg l_3$ and $\neg l_4$, that do not already contain u or an upgrade from its path P_u. Lets assume, that one of these neighbor clauses is $c_3 = (\neg l_3 \vee l_5 \vee l_6)$. Then we can write the clause $c_1 = (l_u \vee l_{u_2} \vee l_3)$ also as $(u \vee u_{2_t} \vee l_3)$ or$(u_{2_t} \vee l_3)$, because the literals are now represented by the variables of the received upgrades and their paths and u_{2_t} is the latest upgrade on the path . Then we get the following equation applying the implication rule:

$$(u_{2_t} \vee l_3) \wedge (\neg l_3 \vee l_5 \vee l_6) \Leftrightarrow (u_{2_t} \vee l_5 \vee l_6) \tag{2}$$

Also u_{2_f} could be sent on to another neighbor clause $c_4 = (u_{2_f} \vee l_7 \vee l_8)$

$$(u_{2_f} \vee l_4) \wedge (\neg l_4 \vee l_7 \vee l_8) \Leftrightarrow (u_{2_f} \vee l_7 \vee l_8) \tag{3}$$

The sending displays applications of the implications rules. Also if both upgrades circle, then also all of their common previous upgrades circle, because the negated and non-negated form of a literal cannot be satisfiable at the same time. So if both corresponding upgrades circle, then also the common previous upgrade cannot satisfy the expression and circles. If the circling upgrade has also a corresponding circling upgrade, the information is further backtracked throughout the upgrade path.

Lemma 2.2 *For a border pattern with two literals of the same form, the algorithm* R *applies the implication rule correctly.*

proof. Let $c_1 = (l_u \vee l_2 \vee l_3)$ and $c_2 = (l_{u_2} \vee \neg l_2 \vee l_3)$ be clauses, with $l_u, l_{u_2} \in L_u$, meaning l_u is within the set of literals of u and $l_2, l_3 \notin L_u$. According to Definition 5, c_1 and c_2 build a border pattern, because besides of the two connector literals from the upgrade, they share the common literal l_3 and contain also inverse literals l_2 and $\neg l_2$. Then the algorithm R will pass upgrade u on via neighbor clauses containing the negated form of the connector literal l_3, if the rules of upgrade connection are applied correctly. This is correct, because according to propositional logic:

$$\begin{aligned} (l_u \vee l_2 \vee l_3) \wedge (l_{u_2} \vee \neg l_2 \vee l_3) &\Leftrightarrow \\ (u \vee l_2 \vee l_3) \wedge (u \vee \neg l_2 \vee l_3) &\Leftrightarrow \\ (u \vee l_3) & \end{aligned} \tag{4}$$

In this equation, the literals within the upgrade were replaced by the upgrade, which is just a matter of notation. Of course the upgrade carries all the information about the literals containing it. By dissolving the border pattern, the algorithm R simply makes use of the implication rule, stating, that l_2 and $\neg l_2$ can both be erased, because it is not possible to fulfill both literals at the same time. So the literals do not need to be considered for the upgrade path. Only l_3, the common literal, remains. It could become another connector literal, if there are clauses with $\neg l_3$, that do not contain u or an upgrade from P_u. □

Lemma 2.3 *If an upgrade u contains an edge pattern, then u cannot be satisfied within P_u. This is applied by* R *correctly.*

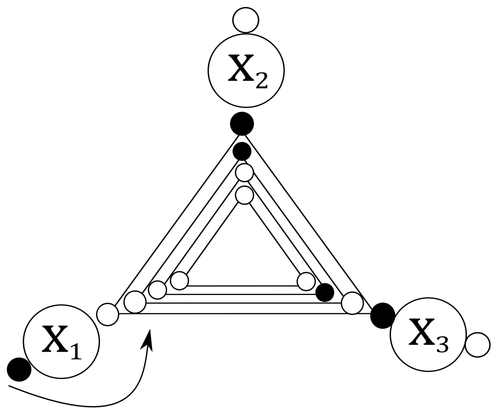

Figure 11: No upgrade will be passed on, because there are circles at x_2 and x_3.

proof. Assuming, that receiving an upgrade means, that the implication rule was applied correctly in previous steps and that the upgrade contains the edge pattern, as described within Definition 11.2, then it contains two border patterns. Let them be p_1 and p_2. Also the definition states, that the connector literal of one pattern has to be inverse to the connector literal of the other pattern. In the proof of Lemma 2.2. it is shown, that a border pattern like p_1 can be formulated also as $p_1 = (l_{u_1} \vee l_c)$, where l_{u_1} is a literal within an upgrade u and l_c is the connector literal. Per definition the second patterns can be written as $p_1 = (l_{u_2} \vee \neg l_c)$ with l_{u_2} being another literal within u:

$$(l_{u_1} \vee l_c) \wedge (l_{u_2} \vee \neg l_c) \Leftrightarrow l_{u_1} \wedge l_{u_2} \tag{5}$$

That means, that the upgrade literals from the patterns have to be both satisfied, which is impossible due to implications. So the upgrade circles, meaning it cannot be satisfied. However, this does not mean, that the whole path P_u is yet unsatisfiable. But it will later be proven, that edge patterns are necessary for an upgrade path to start dissolving and eventually be unsatisfiable.

Lemma 2.4 *For an intermediate clause,* `R` *applies the implication rules correctly.*

proof. For an intermediate clause `R` does not perform an upgrade, as soon as not at least two of its literals received upgrades and only circle, if it receives an upgrade on all of its three connector literals. Let $c_1 = (x_1 \vee x_2 \vee x_3)$ be an intermediate clause. If it receives an upgrade u at x_1, there are still two other choices to fulfill c_1, x_2 and x_3. The algorithm will not transport the upgrade u on via x_2 or x_3 in this case. This is because, if there are still two options open to fulfill the clause, the decision for one literal has not yet to be made. Only if the clause receives an upgrade from P_u or u at another connector literal, an upgrade will be sent on eventually to neighbor clauses. □

This proves Lemma 2, because it is proven, for every kind of pattern.

Lemma 3 Every unsatisfiable upgrade sequence begins and ends with edge patterns.

proof. An edge pattern is defined as a combination of two border patterns with the common literal of one pattern being inverse to the common literal of the other pattern. Because of that every clause of the pattern has two of its literals already eliminated from being able to satisfy the expression, because their inverse counterpart is already within the pattern. The upgrades from an edge pattern get only transported on via one literal per clause, if the sequence is unsatisfiable, because the connector literals of each clause also need to have an inverse counterpart within the graph.
With no other pattern, this is possible. Let $c = (x_1 \vee x_2 \vee x_3)$ and $\acute{c} = (\neg x_3 \vee x_4 \vee x_5)$ be clauses that are not within an edge pattern. By eliminating x_3 using the implication rule, one literal from c would be dissolved, but the literals x_4 and x_5 would be added for x_3 creating $c_3 = (x_1 \vee x_2 \vee x_4 \vee x_5)$. So the number of literals, that would need to be eliminated for the expression to be unsatisfiable has then increased by 1.
The only way to break even or reduce the number of literals in total is, if two clauses share one or two common literals and one clause contains a literal, that the other contains in its inverse form. For example if $c = (x_1 \vee x_2 \vee x_3)$ and $\acute{c} = (\neg x_3 \vee x_2 \vee x_5)$. The clauses would build a border pattern. If connector literals x_5 and x_1 have also implications with other clauses, also x_3 would be eliminated, because it would then exist in the negated and non-negated form within the path. But this would not resolve x_2.
To also resolve x_2 another border pattern is needed, demanding $\neg x$ as a common literal, because an intermediate clause for example, would bring two new literal arguments into play, not just one, that also needs to be satisfied, so we can only work with an already reduced pattern, another border pattern. But this is the definition of an edge pattern.
Only an edge pattern has only one connector per clause for a starting pattern and thus could be, if dissolved by following circling upgrades, mark the beginning of the unsatisfiable upgrade sequence, meaning, that it would contain the start upgrades, because the two first literals, that would also be the last to be dissolved are within the edge pattern. The same goes for the ending. If the ending would not be an edge pattern, then the upgrade sequence could always be extended by upgrading via another connector literal. The only way for it to run into a corner would be an another edge pattern. But it has to be noted, that the same edge pattern could be beginning and ending of a sequence theoretically. For example, if the upgrades from that pattern target all connector literals of another pattern. □

Lemma 4 *The rules of upgrade connection are applied correctly by* `R`

proof. The rule of upgrade connection states, that:

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 created a new leading upgrade u_l with u_c and u as previous upgrades.
(2) If $P_{u_c} \subseteq P_u$, then there is no need to perform the upgrade step.
(3) If $P_u \subseteq P_{u_c}$, then u will be assigned to l and be passed on.

(4) If an upgrade u and its corresponding upgrade $\acute{u}$ attempt to upgrade to a literal l, then their common previous upgrade u_{prev} will be passed on to l.
In the case (1) the idea is, that if multiple upgrades u_1 and u_2 with disjunct paths P_{u_1} and P_{u_2} are sent to the same literal l, then they connect to an upgrade u_c, that has u_1 and u_2 as previous upgrades. P_{u_1} and P_{u_2} connect to P_{u_c}. This makes no different for the satisfiability, because if $l \in L_{u_1}, L_{u_2}$, u_1 and u_2 will only behave differently, if they encounter another upgrade and eventually merge, because then the upgrade paths will be evaluated. But because P_{u_1} and P_{u_2} are still integrated within P_{u_c}, this makes no difference. Also because of the other rules, if a path P_{u_c} from a new upgrade u_n is integrated within another path P_{u_n} $P_{u_c} \subseteq P_{u_n}$, then the upgrade with the longer path will be passed on and if a leading upgrade is encountered while backtracking, the corresponding upgrade does not need to circle for the information to be backtracked.
The second case (2) will not happen, because assuming the case would occur, then the current upgrade u would have already been sent to the receiving literal l of u_c in the step, were the upgrade u, that is also part of P_{u_c} per definition reached the receiving literal and this contradicts u attempting to upgrade to l again.
For the third case (3), the rule of upgrade connection states, that only the upgrade with the longer path, u, is passed on. This is correct, because the longer path already contains the upgrade path of the upgrade with the shorter path, u_c, and the shorter path P_{u_c} would sooner or later also adapt to the longer path anyway.
The fourth rule (4) states, that an upgrades u can merge with its corresponding upgrade $\acute{u}$. This is because an upgrade u and its corresponding upgrade $\acute{u}$ contain upgrade paths P_u and $P_{\acute{u}}$, that only differ in one literal l, that is negated on one path and non-negated on the other path. If the paths merge, the negated and non-negated literal would dissolve, because P_u and $P_{\acute{u}}$ could never both satisfy l. So the literal can be eliminated from the common re-connected path and only the common previous upgrade needs to be passed on. $\square$

Lemma 5 *The backtracking steps of* `R` *run correctly*

proof. If an upgrade u circles, then it is not satisfiable, because all literals from a pattern p, that are not already present in its negated and non-negated forms within the pattern, have been excluded due to implications. So the upgrade is in conflict with p. When P_u cannot satisfy u, then the definition of circling states, that if the upgrade corresponding to u, $\acute{u}$, also circles, also all common previous upgrades circle. This is correct, because the paths of P_u and $P_{\acute{u}}$ only contain one different upgrade, u on P_u and $\acute{u}$ on $P_{\acute{u}}$, that have per definition of being correspondent inverse base literals, that could not be satisfied at the same time. If two corresponding upgrades cannot be satisfied, then all of their common previous upgrades circle. That means, they now build the current tip of the upgrade path. If also the upgrade correspondent to the previous upgrade circles, then the step will of course be repeated, until it is not possible anymore or corresponding start upgrades circle. If corresponding start upgrades circle, then P_u is an unsatisfiable sequence, because no assignment could satisfy a subset of the expression e and if an expression contains an unsatisfiable subset, it could not be satisfied, because for a solution e_{sol} for e, there has to be an assignment for every literal $l \in e$ solving e, and if there is no correct assignment for a subset L_s of all literals within the expression L_e, with $L_s \subseteq L_e$, there also will not be a correct solution for e. This is why the algorithm `R` has to start sequentially at all edge patterns, before returning satisfiable. □

Theorem 3 *The algorithm* `R` *returns a correct solution to the 3-SAT problem.*

Beginning of induction With Theorem 1 it was already proven, that `R` runs correctly with a single coalition C_1. If C_1 is $full$, then the algorithm `R` returns unsatisfiable and otherwise satisfiable.
Also a $full$ coalition consists of 2 contradicting edge patterns, with the beginning edge pattern being in a circle with the ending edge pattern. It is the smallest possible kind of unsatisfiable sequence. □

Induction step With already proven, that each unsatisfiable sequence begins and ends with edge patterns in Lemma 3, it can be assumed, that the beginning of the sequence has been an edge pattern and that already n patterns have been visited by the algorithm. For the pattern $n + 1$, it could be an edge pattern, intermediate pattern, border pattern or an intermediate clause. Within Theorem 2, it was already proven, that for all those cases, the algorithm `R` runs correctly and is able to identify all circles correctly.
The fact, that the algorithm starts from every edge pattern within the graph G, if not already visited, makes sure, that no edge pattern is left out.
And that also in case of multiple upgrades in place, the rules of upgrade connection are applied correctly, which is proven in Lemma 4.
In Lemma 5 it is proven, that in case of a circle, the backtracking process runs correctly.
So the correctness of every following step is also proven, proving the correctness of `R`. □

Theorem 4 *The algorithm* `R` *has a runtime of* $O(n^3)$.

proof. The 3-SAT graph can be built up in a runtime of $O(n^2)$, because for every clause, that connects to neighbor clauses, the algorithm has to go through every existing clause in the worst case.
The algorithm `R` runs through the graph making use of implications starting at edge pattern. Finding the edge patterns might also take $O(n^2)$ time. Then sending upgrades trough the graph can be done in $O(n^3)$, because every literal only gets visited once at most, but maybe by more than one upgrade. The complexity of the upgrades is within $O(n^2)$, because the set of all upgrade U is a subset of G and there is at most one new upgrade per literal, because they either merge together to one upgrade, if disjunct, or a single new upgrade is created as a following upgrade. If two or more upgrades are sent to the same literal, the algorithm has to run through both paths eventually to check, if they are equal, disjunct or one is a subset of the other. That costs time of $O(n)$. All in all the runtime of this step will be $O(n^3)$.

If an upgrade and its corresponding upgrade circle, then the algorithm might also backtrack through the upgrade path, a subgraph of the 3-SAT graph, having also a complexity of $O(n)$. This could happen in the worst case at every step, making the runtime of this step also cubic in $O(n^3)$ time. $\square$

Theorem 5 *The algorithm* `R` *terminates.*

proof. The algorithm visits in every execution step one literal of the graph G, that has a complexity of $O(n)$, in which it in the worst case backtracks trough two upgrade paths, at most one normal or already connected leading upgrade and the incoming upgrade. The upgrade paths are defined as a subgraphs of G, have also a complexity of $O(n)$ and are disjunct. If every literal was visited, then the algorithm terminates, which proves Theorem 5. $\square$

All in all, the correctness, the polynomial runtime of $O(n^3)$ and the termination of `R` are proven $\square$

6 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] Stephen A. Cook. The complexity of theorem-proving procedures. *Symposium on Theory of Computing*, 1971.

[2] Shamir A. Even S., Itai A. On the complexity of time table and multi-commodity flow problems. *SIAM Journal on Computing*, 5, 1976.

[3] Edward A Hirsch Evgeny Dantsin, Andreas Goerdt. A deterministic (2-2/(k+1))n algorithm for k-sat based on local search. *Theoretical Computer Science*, 8, 1979.

[4] Melven R. Krom. The decision problem for a class of first-order formulas in which all disjunctions are binary. *Mathematical Logic Quarterly*, 13, 1967.

[5] H. Putnam M. Davis. A probabilistic algorithm for k-sat and constraint satisfaction problems. *Foundations of Computer Science*, 1999.

[6] Uwe Sch"oning. A probabilistic algorithm for k-sat and constraint satisfaction problems. *Foundations of Computer Science*, 1999.