Polynomial Exact-3-SAT-Solving Algorithm

Matthias Michael Mueller
louis@louis-coder.com

Sat, 2018-11-17
Version DM-2.1

Abstract

This article describes an algorithm which is capable of solving any instance of a 3-SAT CNF in maximal $O(n^{18})$, whereby n is the literal index range within the 3-SAT CNF to solve. A special matrix, called clause table, will be introduced. The elements of this clause table are 3-SAT clauses which do each get assigned a Boolean value that depends on the occurrence of the corresponding 3-SAT clause in the 3-SAT CNF. It will be shown that if and only if all clauses of at least one line of the clause table matrix are true, the related 3-SAT CNF is solvable. Although the clause table matrix has 2^n many lines, the presented algorithm can determine its expected evaluation result in polynomial time and space. On the supposition the algorithm is correct, the P-NP-Problem would be solved with the result that the complexity classes NP and P are equal.

1 Introduction

Problems denoted as "NP-complete" are those algorithms which need an amount of computing time or space which grows exponentially with the problem size n. If it should be accomplished to solve in general at least one of those problems in polynomial time and space, all NP-complete problems out of the complexity class NP could from then on be solvable much more efficiently. Finding the answer to the open question if such a faster computation is possible at all is called the P versus NP problem.[6] The present article describes and proves the correctness of an algorithm which is supposed by its author to solve the NP-complete problem "exact-3-SAT" in polynomial time and space. In case all crucial statements made in this document are correct, the P versus NP problem would be solved with the conclusion that NP-complete problems *can* be solved in polynomial time and space. In this case, NP-complete problems would all lie in the complexity class P and thus P = NP was proven.

2 Definitions

The following definitions will be used to describe and analyze the polynomial algorithm. Their purpose will become accessible in the course of the document.

2.1 3-SAT CNF

Given is a formula in conjunctive normal form:

$$CNF = \bigwedge_{i=1}^{\gamma} (\epsilon_{i1} x_{i1} \vee \epsilon_{i2} x_{i2} \vee \epsilon_{i3} x_{i3})$$

This 3-SAT CNF, synonymously called SAT CNF or just CNF, consists of γ many conjugated clauses. Within each clause, $x_{i1}, x_{i2}, x_{i3} \mid i_1, i_2, i_3 \in \{1, ..., n\}$ are disjunctive Boolean variables called literals. There are always exactly three literals in each clause. Each literal has assigned a second variable epsilon: $\epsilon_{i1}, \epsilon_{i2}, \epsilon_{i3} \in \{0, 1\}$. If this $\epsilon_{ia} \mid a \in \{1, 2, 3\}$ has the value 1, then the value of x_{ia} is negated when evaluating the CNF. If $\epsilon_{ia} \mid a \in \{1, 2, 3\}$ has the value 0, no negation is done. The literal indices are chosen out of a set of natural numbers $\{1, ..., n\}$. The literal indices are pair-wise distinct: $i_2 \neq i_1, i_3 \neq i_1, i_3 \neq i_2$. n will be used as description of the problem size in the upcoming analysis of the solver's complexity.

The task of the presented polynomial solver is to find out if there is an assignment of either true or false to each literal so that a given 3-SAT CNF as a whole evaluates to true. Then we say this assignment, synonymously called solution, satisfies the CNF. If the CNF evaluates to true, we say the 3-SAT CNF is solvable. If it is not possible to satisfy the CNF, we say the 3-SAT CNF is unsatisfiable. The solvability must be determined by the solver in polynomial time and space. The solver only says *if* there is a solution or not, it does not output an assignment. This is still sufficient to solve the P versus NP problem.[6]

2.2 Notation Convention

Within this document, a fixed notation convention for 3-SAT clauses is used:

- The variables of the literal indices of any 3-SAT clause are noted in lowercase letters. For instance, i_1, i_2 and i_3 are the literal indices of the 3-SAT clause $(\epsilon_{i1} x_{i1} \vee \epsilon_{i2} x_{i2} \vee \epsilon_{i3} x_{i3})$. The literal indices as a whole, i.e. letter (here: i) plus the numeration (1, 2, 3), contain a natural number, which is out of the range $\{1, ..., n\}$. This means for the example: $i_1 \in \{1, ..., n\}$, $i_2 \in \{1, ..., n\}$ and $i_3 \in \{1, ..., n\}$.
- The related 3-SAT clause is identified by an uppercase letter. This letter does always match the letter of the literal indices (except the case, as already pointed out). In the current example, it would be $I = (\epsilon_{i1} x_{i1} \vee \epsilon_{i2} x_{i2} \vee \epsilon_{i3} x_{i3})$.
- This means that e.g. j_1, j_2, j_3 belong to a clause J which is typically mentioned in the same context. The same applies to e.g. k_1, k_2, k_3, which belong to a clause K, and so on. Whenever there are, in the nearby document section, some lowercase literal indices and an uppercase clause identifier, then both relate to the same clause. This relationship will be of importance especially in the upcoming proof of correctness.

2.3 Possible Clauses

The set PC of possible clauses is defined as:

$$PC = \{(\epsilon_{i1} x_{i1} \vee \epsilon_{i2} x_{i2} \vee \epsilon_{i3} x_{i3})\}$$

with $\epsilon_{i1}, \epsilon_{i2}, \epsilon_{i3} \in \{0, 1\}, i_1, i_2, i_3 \in \{1, ..., n\}, i_2 \neq i_1, i_3 \neq i_1, i_3 \neq i_2$.

$|PC| = (2^3 \times \binom{n}{3})$, because there are 2^3 epsilon combinations and $\binom{n}{3}$ possibilities to choose 3 distinct literal indices out of $\{1, ..., n\}$. If a clause $C \in PC$ appears in the 3-SAT CNF to solve we say C is an initially false clause and write $\tau(C) = 0$. If a clause $C \in PC$ does not appear in the 3-SAT CNF to solve we say C is an initially true clause and write $\tau(C) = 1$. A clause $C = (\epsilon_{c1}x_{c1} \vee \epsilon_{c2}x_{c2} \vee \epsilon_{c3}x_{c3}) \in PC$ appears in the 3-SAT CNF if there's a clause $S = (\epsilon_{s1}x_{s1} \vee \epsilon_{s2}x_{s2} \vee \epsilon_{s3}x_{s3})$ in the 3-SAT CNF so that $\forall x \in \{1, 2, 3\} : (c_x = s_x \wedge \epsilon_{cx} = \epsilon_{sx})$.

2.4 Underlying Solutions

The set of underlying solutions is the set of all 2^n many producible $0, 1$ progressions:

$$US = \{U \in \{0, 1\}^n\}$$

US_x is the x-th underlying solution in the set. U_x is the x-th element in the single underlying solution U.

2.5 In Conflict

Two clauses $J, K \in PC$ are said to be "in conflict" if they have at least one literal index in common and the ϵ's concerned are not equal:

$$(J \not\equiv K) \Leftrightarrow \exists(i \in \{1, 2, 3\} \mid ((\epsilon_{j_i} = 0 \wedge \epsilon_{ki} = 1) \vee (\epsilon_{j_i} = 1 \wedge \epsilon_{ki} = 0)) \wedge (j_i = k_i))$$

Similar is defined for a clause $J \in PC$ and an underlying solution $U \in US$:

$$(J \not\equiv U) \Leftrightarrow \exists(i \in \{1, 2, 3\} \mid ((\epsilon_{j_i} = 0 \wedge U_{j_i} = 1) \vee (\epsilon_{j_i} = 1 \wedge U_{j_i} = 0)))$$

We define "not in conflict" as $(J \equiv K) \Leftrightarrow \neg(J \not\equiv K)$ resp. $(J \equiv U) \Leftrightarrow \neg(J \not\equiv U)$.

Furthermore it is defined for a clause $C \in PC$ and an underlying solution $U \in US$: $C \in U \Rightarrow C \equiv U$.

2.6 Clause Table

The clause table, abbreviated CT, is a matrix with 2^n lines and $\binom{n}{3}$ columns. A CT line index is to be denoted with l, and a CT column index is to be denoted with c. Each CT line CT_l contains all possible clauses not in conflict with the l-th underlying solution out of the set US: $CT_l = \{C \in PC \mid (C \equiv US_l)\}$.

For better understanding, here an excerpt from the clause table for $n = 4$.

$$CT = \begin{pmatrix} (0x_1 \vee 0x_2 \vee 0x_3) & (0x_1 \vee 0x_2 \vee 0x_4) & (0x_1 \vee 0x_3 \vee 0x_4) & (0x_2 \vee 0x_3 \vee 0x_4) \\ (0x_1 \vee 0x_2 \vee 0x_3) & (0x_1 \vee 0x_2 \vee 1x_4) & (0x_1 \vee 0x_3 \vee 1x_4) & (0x_2 \vee 0x_3 \vee 1x_4) \\ (0x_1 \vee 0x_2 \vee 1x_3) & (0x_1 \vee 0x_2 \vee 0x_4) & (0x_1 \vee 1x_3 \vee 0x_4) & (0x_2 \vee 1x_3 \vee 0x_4) \\ (0x_1 \vee 0x_2 \vee 1x_3) & (0x_1 \vee 0x_2 \vee 1x_4) & (0x_1 \vee 1x_3 \vee 1x_4) & (0x_2 \vee 1x_3 \vee 1x_4) \\ \vdots & & & \end{pmatrix}$$

The underlying solution for CT line 1 is $\{0, 0, 0, 0\}$, for CT line 2 $\{0, 0, 0, 1\}$, for CT line 3 $\{0, 0, 1, 0\}$, for CT line 4 $\{0, 0, 1, 1\}$ and so on.

Claim 2.6.1 *If and only if there is a CT line with clauses which are all absent from the 3-SAT CNF, then this 3-SAT CNF is solvable:*

$$\exists(l \mid \forall c = \{1, ..., \binom{n}{3}\} : (C_c \equiv US_l \wedge \tau(C_c) = 1)) \Leftrightarrow \texttt{SAT CNF is solvable}.$$

Proof: We define for $S, S' \in US$: $((S' = neg(S)) := (\forall x \in \{1, ..., n\} : S'_x = \neg S_x$ [1] $))$. Furthermore we define that a clause $C = (\epsilon_{c1} x_{c1} \vee \epsilon_{c2} x_{c2} \vee \epsilon_{c3} x_{c3})$ is satisfied by an underlying solution $U = \{0, 1\}^n$ if $\exists x \in \{1, 2, 3\} : \epsilon_{cx} = U_{cx}$. Notice that here merely one x is enough.

In case the SAT CNF is solvable: We know there is at least one solution $S \in US$ which satisfies the SAT CNF. The SAT CNF does not contain any clause from the CT line l with $US_l = neg(S)$ because such a clause would not be satisfied by S and thus S would not satisfy *all* clauses of the SAT CNF. So at least all clauses from the CT line l do not appear in the SAT CNF.

In case the SAT CNF is unsatisfiable: For each $S \in US$ there must be at least one clause C in the SAT CNF for which applies: $\forall x \in \{1, 2, 3\} : \epsilon_{cx} = \neg(S_{cx})$. Otherwise S would be a solution, as mentioned in the previous paragraph, and the SAT CNF would be solvable, what is not the case by the basic assumption of this proof. This means: $\forall(neg(S) \in US)\exists(C \equiv neg(S) \wedge \tau(C) = 0)$. Because for absolutely every underlying solution $US_x \mid x \in \{1, ..., 2^n\}$ there's at least one clause $\tau(C) = 0$ not in conflict with US_x, it is for sure we can conclude: $\forall(S \in US)\exists(C \equiv S \wedge \tau(C) = 0)$. This implies: $\forall(US_l \mid l \in \{1, ..., 2^n\})\exists(C \equiv US_l \wedge \tau(C) = 0)$. This does <u>not</u> fulfill $\exists(l \mid \forall c = \{1, ..., \binom{n}{3}\} : (C_c \equiv US_l \wedge \tau(C_c) = 1))$. This is the result we need, because it was claimed the latter condition is only fulfilled if there is a CT line with only initially true clauses. $\square$

2.7 Being Contained

We regard the clauses $I = (\epsilon_{i1} x_{i1} \vee \epsilon_{i2} x_{i2} \vee \epsilon_{i3} x_{i3})$, $H = (\epsilon_{h1} x_{h1} \vee \epsilon_{h2} x_{h2} \vee \epsilon_{h3} x_{h3})$, $J = (\epsilon_{j1} x_{j1} \vee \epsilon_{j2} x_{j2} \vee \epsilon_{j3} x_{j3})$ and $K = (\epsilon_{k1} x_{k1} \vee \epsilon_{k2} x_{k2} \vee \epsilon_{k3} x_{k3})$.

We define the set D of tuples:

$$D = \{\{(j_a, \epsilon_{j\,a})\} \cup \{(k_b, \epsilon_{kb})\} \mid a, b \in \{1, 2, 3\}\}$$

We say I is contained within (J, K) if:

$$(I \equiv J \wedge I \equiv K \wedge J \equiv K) \wedge$$

$$(\forall x \in \{1, 2, 3\} : (i_x, \epsilon_{ix}) \in D)$$

This means that each literal of I appears with same literal index and ϵ value in J or K or both. Also the three clauses are not in conflict pair-wise.

For instance, some $I = (0x_1 \vee 0x_2 \vee 0x_4)$ is contained within $J = (0x_1 \vee 0x_2 \vee 1x_5)$ and $K = (0x_2 \vee 0x_3 \vee 0x_4)$. This is the case because all literal indices of I, which are 1, 2 and 4, match literal indices in J and K. Also the ϵ values of I, which are three times 0, match the corresponding ϵ values in J and K. In contrast, some $I = (1x_1 \vee 0x_2 \vee 0x_4)$ is *not* contained within $J = (0x_1 \vee 0x_2 \vee 1x_5)$ and $K = (0x_2 \vee 0x_3 \vee 0x_4)$, because ϵ_{i1}, which is 1, does not match ϵ_{j1}, which is 0. Moreover, some $I = (0x_1 \vee 0x_2 \vee 0x_6)$ is *not* contained within $J = (0x_1 \vee 0x_2 \vee 1x_5)$ and $K = (0x_2 \vee 0x_3 \vee 0x_4)$, because x_6 from I appears in neither J nor K.

A non-formal visualization shall illustrate "I is contained within (J, K)":

$$J = (A \vee B \vee C)$$

$$K = (D \vee E \vee F)$$

$$I = ((A \vee B \vee C \vee D \vee E \vee F) \vee (A \vee B \vee C \vee D \vee E \vee F) \vee (A \vee B \vee C \vee D \vee E \vee F))$$

Here A, B, C, D, E, F each represent a clause literal with preceded epsilon value. Exactly one letter must be chosen from each "$(A \vee B \vee C \vee D \vee E \vee F)$". I is contained within (J, K) if all of I's A, B, C, D, E or F appear in J or K or both. I must not contain any A, B, C, D, E or F more than once. This is in the visualization not explicitly defined to keep a simple notation. J or K or both might contain literals which

[1] $\neg 0 = 1$ and $\neg 1 = 0$

do not appear in I. But I must only contain literals which appear in J or K or both. Please notice that this illustration is given only to improve the reader's understanding of being contained, it's of course not a correct mathematical formulation.

We say I and H are contained within (J, K) if:

$$(I \equiv J \wedge I \equiv K \wedge H \equiv J \wedge H \equiv K \wedge J \equiv K) \wedge$$

$$(\forall (a, b, c, d, e, f \in \{1, 2, 3\}, a \neq b, a \neq c, b \neq c, d \neq e, d \neq f, e \neq f, p \in \{1, ..., n\}) : ($$
$$((i_a, \epsilon_{ia}) \in D) \wedge ((i_b, \epsilon_{ib}) \in D) \wedge ((i_c, \epsilon_{ic}) = (p, 0)) \wedge$$
$$((h_d, \epsilon_{hd}) \in D) \wedge ((h_e, \epsilon_{he}) \in D) \wedge ((h_f, \epsilon_{hf}) = (p, 1))))$$

This means that two literals of each I and H appear each with the same index and ϵ value in J or K or both. One literal index of I does not appear in J or K but is equal to some value p. The corresponding ϵ_{ic} is always 0. Similarly, one literal index of H does not appear in J or K but is equal to the value p, the same p as in the I case. The corresponding ϵ_{hf} is always 1.

The literal index p is to be called the conflict literal index.

Any p out of $\{1, ..., n\}$ is valid, as long as p is not equal to any literal index in J or K. Formally, it must apply: $p \in \{1, ..., n\} \mid (p \notin \{j_1, j_2, j_3\} \wedge p \notin \{k_1, k_2, k_3\})$. During the rest of the document, many example cases will be shown where two clauses are contained within another two clauses. Mostly only one choice of p will be presented, although there might be *several* suitable p. This is done to save space and to simplify the examples. Each example does work with the one stated p in any case, so that restriction is not fatal. Usually the p containing the smallest possible number is chosen.

For instance, some $I = (0x_1 \vee 0x_2 \vee 0x_7)$ and $H = (0x_2 \vee 0x_3 \vee 1x_7)$ are contained within $J = (0x_1 \vee 0x_2 \vee 1x_5)$ and $K = (0x_2 \vee 0x_3 \vee 0x_4)$. This is the case because all literal indices and corresponding ϵ values of I and H appear in J or K or both. The only exception is the literal index $p = 7$, which does not appear in J or K. The corresponding ϵ_{i3} is 0 and the corresponding ϵ_{h3} is 1 [2]. Therewith the conditions of I and H being contained within (J, K) are fulfilled.

A non-formal visualization shall illustrate "I and H are contained within (J, K)":

$$J = (A \vee B \vee C)$$

$$K = (D \vee E \vee F)$$

$$I = ((A \vee B \vee C \vee D \vee E \vee F) \vee (A \vee B \vee C \vee D \vee E \vee F) \vee 0x_p)$$

$$H = ((A \vee B \vee C \vee D \vee E \vee F) \vee (A \vee B \vee C \vee D \vee E \vee F) \vee 1x_p)$$

Again, A, B, C, D, E, F each represent a clause literal with preceded epsilon value. Exactly one letter must be chosen from each "$(A \vee B \vee C \vee D \vee E \vee F)$". I and H are contained within (J, K) if all of I's A, B, C, D, E or F appear in J or K or both. The only exception is one literal whose epsilon value is 0 and whose literal index is p. The same applies for H, except that the one literal's epsilon value is 1, but the literal index is also p. I must not contain any A, B, C, D, E or F more than once. The same applies to H. It is allowed that I contains other choices of A, B, C, D, E or F than H. It is also allowed that I contains one or more letters which do also appear in H, as long as letters do not repeat within the very same clause. The illustration suggests the p literal must be the third literal within the clause. This is not required, the p literal can also be at second or first location within the clause. This is not explicitly shown in the illustration to keep a simple notation. J or K or both might contain literals which do not appear in I or H or both. But each I and H must only contain literals (except the p literal) which appear in J or K or both.

Please notice "being contained" is ambiguous, it can mean one clause I, or two clauses I and H are contained within (J, K). The two ambiguous cases can be distinguished by the count of contained clauses (one or two, as just mentioned).

[2] Here the deepest literal index is 3 because the regarded epsilons belong to the third literal in the clauses I resp. H.

Finally, please notice that the upcoming statement "a clause table (CT) line contains one or more clauses" just means the clauses appear in the clause table line. This has nothing to do with the property of being contained as defined here in 2.7. So "to contain" is in this document sometimes used also in its common meaning. The reader should distinguish the meaning depending on the context.

2.8 Enabled/Disabled Clause Tuple

We will make use of the terms "enabled clause tuple(s)" and "disabled clause tuple(s)". An enabled tuple of two possible clauses J and K is noted as $\pi(J,K) = 1$. A disabled tuple of two possible clauses J and K is noted as $\pi(J,K) = 0$.

We say a tuple (J,K) is or gets disabled when the solver sets $\pi(J,K) := 0$. It is not important if the tuple was enabled before, the crucial point is that it does (from then on) apply $\pi(J,K) = 0$.

The enabled state of any tuple is non-volatile, it keeps its recently set value during the whole solving process of a SAT CNF.

The meaning of a disabled tuple is the following: at the very end of the solving process, exactly those tuples (J,K) will be disabled which appear exclusively in CT lines which contain at least one false clause.

2.9 Possible Clause Locations in CT

Claim 2.9.1 *If $I \in PC$ is contained within $(J,K) \mid J,K \in PC$, then I appears in every CT line l containing (J,K).*

Proof: From definition 2.6 we know all those possible clauses appear in a CT line l which are not in conflict with US_l. So I could only be in an other CT line than J and K if I was in conflict with US_l and J and K were not: $I \not\equiv US_l \wedge J \equiv US_l \wedge K \equiv US_l$. This is only possible if it applied: $\exists x \in \{1,2,3\} : \epsilon_{ix} \neq US_{l_{i_x}} \wedge \epsilon_{j_x} = US_{l_{j_x}} \wedge \epsilon_{kx} = US_{l_{k_x}}$. But from definition 2.7 we can derive that it does always apply: $\forall x \in \{1,2,3\} \exists y \in \{1,2,3\} : (\epsilon_{ix} = \epsilon_{j_y} \wedge \epsilon_{j_y} = US_{l_{j_y}}) \vee (\epsilon_{ix} = \epsilon_{ky} \wedge \epsilon_{ky} = US_{l_{k_y}})$. So we know each of I's literal indices and the corresponding epsilon value is always equal to one in J or K. We also know that J and K are not in conflict with US_l. This means that I is never in conflict with US_l if I is contained within (J,K). □

Claim 2.9.2 *If $I \in PC$ and $H \in PC$ is contained within $(J,K) \mid J,K \in PC$, then either I or H, but not both, appears in a CT line l containing (J,K).*

Proof: The situation is similar as in the previous proof, with the difference that I and H are in conflict at a literal index p, see definition 2.7. Because U_p can either be 0 or 1, not both I and H can be within the same CT line. □

3 The Polynomial Exact-3-SAT Solving Algorithm

The polynomial solver decides in polynomial time and space if any given 3-SAT CNF is solvable or not. The polynomial solving algorithm consists of an initialization phase, followed by the iterated application of two rules. The algorithm is stated in C-like pseudo code. If several lines after an `if ()` or `while` statement are equally indented, these lines belong to one and the same block. In this case *all* equally indented lines get executed if the prior condition is true. Two slashes `//` do always introduce a comment, here in the pseudo-code and later also in mathematical formulas.

INITIALIZATION The solver regards the entire set of tuples $\{(J,K) \mid J,K \in PC\}$. Only those tuples (J,K) get initially enabled whose J and K are not in conflict and if it does apply $\tau(J)=1$ and $\tau(K)=1$.

```
foreach J ∈ PC
  foreach K ∈ PC
    if ((τ(J) = 1) ∧ (τ(K) = 1) ∧ (J ≡ K))
      π(J,K) := 1
    if ((τ(J) = 0) ∨ (τ(K) = 0) ∨ (J ≢ K))
      π(J,K) := 0
```

RULE 1 Any tuple (J,K) gets disabled if there is some *contained* I and it does apply $\pi(I,J)=0 \vee \pi(I,K)=0$.

```
foreach J ∈ PC
  foreach K ∈ PC
    if (π(J,K) = 1)
      foreach I ∈ PC
        if (I is contained within (J,K))
          if (π(I,J) = 0 ∨ π(I,K) = 0)
            π(J,K) := 0
            Changed := true
```

The solver finds a contained I by just testing *every* $I \in PC$. This is implemented in the way the `foreach` loop iterates through all possible clauses. The second `if ()` condition works as a kind of 'filter' which lets pass through only I being *contained* within (J,K). If also the third `if ()` condition is fulfilled, the tuple (J,K) gets disabled and the Boolean flag *Changed* is set to *true*. Only those tuples (J,K) are processed which are not yet disabled (first `if ()` condition).

RULE 2 Any tuple (J,K) gets disabled if there is some *contained* I and H and it does apply $(\pi(I,J) = 0 \vee \pi(I,K)=0) \wedge (\pi(H,J)=0 \vee \pi(H,K)=0)$.

```
foreach J ∈ PC
  foreach K ∈ PC
    if (π(J,K) = 1)
      foreach I ∈ PC
        foreach H ∈ PC
          if (I and H are contained within (J,K))
            if ((π(I,J) = 0 ∨ π(I,K) = 0) ∧ (π(H,J) = 0 ∨ π(H,K) = 0))
              π(J,K) := 0
              Changed := true
```

This part of the solver works similar to the part which implements RULE 1. The main difference is that *two* clauses I and H need to be contained within (J,K).

RULE 1 and RULE 2 are applied repeatedly until all $I,H,J,K \in PC$ combinations have been regarded once in RULE 1 and RULE 2 and no tuple (J,K) has been disabled any more:

```
while (true)
  bool Changed := false // can be set to true by RULE 1 or RULE 2
  do RULE 1
  do RULE 2
  if (Changed == false) exit while
```

If any tuple (J,K) got disabled by RULE 1 or RULE 2, *Changed* is set to *true*. If no tuple (J,K) has been disabled by RULE 1 or RULE 2, *Changed* kept its initial state $false$ [3]. Then the `while (true)`-loop is left and the final result of the solving process is determined as follows:

[3] *Changed* is re-initialized to $false$ with *each* new iteration of the `while (true)`-loop.

$$\exists(J,K \in PC \mid \pi(J,K)=1) \Rightarrow \texttt{SAT CNF is solvable}.$$
$$\neg\exists(J,K \in PC \mid \pi(J,K)=1) \Rightarrow \texttt{SAT CNF is unsatisfiable}.$$

In words, if at least one enabled tuple rests, the SAT CNF is solvable. If all tuples were disabled, the SAT CNF is unsatisfiable.

There is the following idea behind the three rules: in 2.8 it was mentioned that a clause tuple (J,K) is disabled if there is at least one initially false clause $C \mid \tau(C)=0$ in each CT line the clause tuple (J,K) appears in. The INITIALIZATION rule implements this directly: we know that J and K appear in each CT line (J,K) appears in. If J or K (or both) is initially false, we know that there's an initially false clause, namely J or K, in each CT line where (J,K) is in. So $\pi(J,K)=0$ means that either J or K is initially false. RULE 1: if $\pi(I,J)=0$ or $\pi(I,K)=0$ then either I, J or K is initially false. We suppose I is the initially false clause, else the INITIALIZATION rule applies. If I is contained within (J,K), what RULE 1 demands, then I appears in every CT line l containing (J,K). This was proven in 2.9. This means that in each CT line containing (J,K) there's also the initially false I. Similar applies to RULE 2: if I and H are contained within (J,K), then either I or H appears in each CT line l containing (J,K). This was proven in 2.9. This means that in each CT line containing (J,K) there's also the initially false I or the initially false H. The finding that there is an initially false clause in each CT line (J,K) appears in is recursively forwarded by the repetitive application of RULE 1 and RULE 2. A deeper examination of how the knowledge of the existence of initially false clauses is passed on is not presented here. Instead, the proof of correctness will be given which does in a formal way make clear that the algorithm determines the satisfiability of any inputted SAT CNF correctly.

4 Proof of Correctness

4.1 Why Solvable Detection is Reliable

Given: At least one CT line l containing only initially true clauses. This CT line l is to be called the active CT line.

$$\exists(l \mid \forall c = \{1,...,\binom{n}{3}\} : (C_c \equiv US_l \wedge \tau(C_c)=1))$$

From 2.6 we know in this case the SAT CNF is solvable.

It will now be shown: None of the 3 solver rules will disable a tuple (J,K) with $J,K \in PC$ which appears in the active CT line:

INITIALIZATION There is no (J,K) with $(\tau(J)=0) \vee (\tau(K)=0) \vee (J \not\equiv K)$ for any J, K out of the active CT line l. Instead, it applies: $\forall J,K \in US_l : \tau(J)=1 \wedge \tau(K)=1 \wedge J \equiv K$. So $\forall J,K \in US_l : \pi(J,K) := 1$.

RULE 1 When I is contained within (J,K) then I appears in any CT line l containing (J,K). This has been shown in 2.9. As it is assumed in this proof that $\tau(C)=1$ if $C \in US_l$, it must apply: I contained within (J,K) with $J,K \in US_l \Rightarrow I \in US_l \Rightarrow \tau(I)=1$. This means $\forall I,J,K \in US_l : (\pi(I,J) = 1 \wedge \pi(I,K)=1)$. But this does not fulfill RULE 1 to disable (J,K), because RULE 1 demands $\exists((\pi(I,J)=0 \vee \pi(I,K)=0) \mid I$ is contained within $(J,K))$. From this contradiction we can deduce that (J,K) will stay enabled.

RULE 2 When I and H are contained within (J,K) then either I or H, but not both, appears in a CT line l containing (J,K). This has been shown in 2.9. This means it applies: $\tau(I)=1 \vee \tau(H)=1$ and therewith $(\pi(I,J)=1 \wedge \pi(I,K)=1) \vee (\pi(H,J)=1 \wedge \pi(H,K)=1)$. But this does not fulfill RULE 2 to disable (J,K), because RULE 2 demands $(\pi(I,J)=0 \vee \pi(I,K)=0) \wedge (\pi(H,J)=0 \vee \pi(H,K)=0)$. We can again conclude that (J,K) will stay enabled.

Because $(\forall(J, K \mid J \equiv US_l \wedge K \equiv US_l)) : \pi(J, K) = 1$, the solver determines "solvable".

4.2 Why Unsatisfiable Detection is Reliable

4.2.1 Preliminary Considerations

4.2.1.1 Initially False Clause in each CT Line

The basic premise of this proof is that the 3-SAT CNF is not solvable. In the definition of the clause table (2.6) it was shown that then there is *not* any clause table line whose clauses are all absent from the 3-SAT CNF. This means in return that there is in *every* clause table line at least one clause which does appear in the 3-SAT CNF. So it applies:

$$\forall l : \exists c \mid (C_c \equiv US_l \wedge \tau(C_c) = 0)$$

Here $C_c \equiv US_l$ means the clause C_c appears in CT line l. This is the case because if some clause is not in conflict with the underlying solution of some CT line, then the clause appears in that CT line. This has been defined in 2.6. Furthermore $\tau(C_c) = 0$ means the clause C_c is initially false because it appears in the 3-SAT CNF. Please recall 2.3.

For the following considerations C_c is denoted by F. In addition, US_l is denoted by U. This is done to simplify the notation and improve understandability.

So far we know in each CT line l there is some clause F which does not appear in the 3-SAT CNF and which is not in conflict with the CT line's underlying solution U. So each of the three epsilon values $\epsilon_{f_1}, \epsilon_{f_2}, \epsilon_{f_3}$ of $F = (\epsilon_{f_1} x_{f_1} \vee \epsilon_{f_2} x_{f_2} \vee \epsilon_{f_3} x_{f_3})$ is equal to the value of the underlying solution U at the position described by F's literal index:

$$\forall x \in \{1, 2, 3\} : \epsilon_{f_x} = U_{f_x}$$

Now comes an important observation: As derivable from the just shown formula, we do know for three literal indices that the epsilon values match (in the way just described). But what we do *not* know which value the literal indices have. This means, we do not know which natural number is inside f_x, $\forall x \in \{1, 2, 3\}$. The definition of F is not in conflict with U only requests the epsilon values between F and U match, but the definition of F is not in conflict with U does not make any restriction of the value in the literal indices f_x, $\forall x \in \{1, 2, 3\}$.

To clarify what this means in practice, an example: We assume $n = 6$. In addition, we assume $U = \{0, 1, 0, 1, 0, 1\}$. $\forall x \in \{1, 2, 3\} : \epsilon_{f_x} = U_{f_x}$ is fulfilled for multiple clauses F. Here are three different, suitable F:

$$F_{example1} = (0x_{f_1} \vee 1x_{f_2} \vee 0x_{f_3}) \; with \; f_1 = 1, f_2 = 2, f_3 = 3$$
$$F_{example2} = (0x_{f_1} \vee 0x_{f_2} \vee 0x_{f_3}) \; with \; f_1 = 1, f_2 = 3, f_3 = 5$$
$$F_{example3} = (1x_{f_1} \vee 1x_{f_2} \vee 1x_{f_3}) \; with \; f_1 = 2, f_2 = 4, f_3 = 6$$

Each of these F has the property that each of its three epsilon values match the corresponding one of U. However, figuratively spoken, the locations of the matches in U vary.

For better understanding, the matched epsilons within U are underlined for all three example F:

$$U_{example1} = \{\underline{0}, \underline{1}, \underline{0}, 1, 0, 1\}$$
$$U_{example2} = \{\underline{0}, 1, \underline{0}, 1, \underline{0}, 1\}$$
$$U_{example3} = \{0, \underline{1}, 0, \underline{1}, 0, \underline{1}\}$$

The shown three F are only an incomplete example. There are $\binom{n}{3}$ many F which fulfill the requirement of F is not in conflict with U. This is the case because it is possible to choose the three distinct variable indices f_1, f_2, f_3 out of $\{1, ..., n\}$.

Summarized, it has been pointed out that the condition F is not in conflict with U is fulfilled for $\binom{n}{3}$ many F. The basic premise in this proof is that for each CT line l there's at least one clause which is not in conflict with the CT line's underlying solution US_l. So the discovery of the past paragraphs applies to *all* CT lines.

The polynomial solver must reliably detect that the 3-SAT CNF is unsatisfiable no matter which of the $\binom{n}{3}$ many clauses in each CT line is the initially false clause $F \mid \tau(F) = 0$.

For better understanding, the kind of visualization used for $U_{example1}, U_{example2}, U_{example3}$ is applied on the first eight underlying solutions of the CT for $n = 6$. The following visualization is of course only an incomplete example, but should contribute to the reader's comprehension:

$\mathtt{US}_1 = \{\underline{0}, 0, \underline{0}, \underline{0}, 0, 0\}$
$\mathtt{US}_2 = \{\underline{0}, 0, \underline{0}, \underline{0}, 0, 1\}$
$\mathtt{US}_3 = \{\underline{0}, 0, \underline{0}, \underline{0}, 1, 0\}$
$\mathtt{US}_4 = \{\underline{0}, 0, \underline{0}, \underline{0}, 1, 1\}$
$\mathtt{US}_5 = \{0, 0, \underline{0}, \underline{1}, \underline{0}, 0\}$
$\mathtt{US}_6 = \{0, 0, \underline{0}, \underline{1}, \underline{0}, 1\}$
$\mathtt{US}_7 = \{\underline{0}, 0, 0, \underline{1}, \underline{1}, 0\}$
$\mathtt{US}_8 = \{\underline{0}, 0, 0, \underline{1}, \underline{1}, 1\}$
$\vdots$

$$CNF_{unsatisfiable} = (0x_1 \vee 0x_3 \vee 0x_4) \wedge (0x_3 \vee 1x_4 \vee 0x_5) \wedge (0x_1 \vee 1x_4 \vee 1x_5) \wedge ...$$

Another example of how each CT line's initially false clause F can be chosen is:

$\mathtt{US}_1 = \{\underline{0}, \underline{0}, 0, \underline{0}, 0, 0\}$
$\mathtt{US}_2 = \{\underline{0}, \underline{0}, 0, \underline{0}, 0, 1\}$
$\mathtt{US}_3 = \{\underline{0}, \underline{0}, 0, \underline{0}, 1, 0\}$
$\mathtt{US}_4 = \{\underline{0}, \underline{0}, 0, \underline{0}, 1, 1\}$
$\mathtt{US}_5 = \{\underline{0}, 0, \underline{0}, \underline{1}, 0, 0\}$
$\mathtt{US}_6 = \{\underline{0}, 0, \underline{0}, \underline{1}, 0, 1\}$
$\mathtt{US}_7 = \{\underline{0}, 0, \underline{0}, \underline{1}, 1, 0\}$
$\mathtt{US}_8 = \{\underline{0}, 0, \underline{0}, \underline{1}, 1, 1\}$
$\vdots$

$$CNF_{unsatisfiable} = (0x_1 \vee 0x_2 \vee 0x_4) \wedge (0x_1 \vee 0x_3 \vee 1x_4) \wedge ...$$

In both example cases, there is one clause of the 3-SAT CNF to solve which is not in conflict with each CT line's underlying solution. This means the polynomial solver must in both example cases output 'unsatisfiable'.

4.2.1.2 Idea of the Proof: Find Initially False Clauses Doing Recursion and Extension

We know there is at least one initially false clause F in every CT line.

Imagine we wanted to check *if* there's at least one initially false clause in each CT line. We could do this as follows:

Definition 4.2.1

```
function recursive_F_search(tuple_set S_F, int λ, int_set p())
  if λ >= |p()|     // is λ larger than the count of elements in p()?
    return FAILED   // won't happen as there's an F in each CT line [4]
  S_F0 = (S_F ∪ (p(λ), 0))
  S_F1 = (S_F ∪ (p(λ), 1))
  if
    ((
    there is any initially false clause F0 with τ(F0) = 0 and
    ((f0_1, ε_f0_1), (f0_2, ε_f0_2), (f0_3, ε_f0_3) ∈ S_F0)
    ∨
    recursive_F_search(S_F0, λ + 1, p()) == F_IN_EVERY_CT_LINE
    )
    ∧
    (
    there is any initially false clause F1 with τ(F1) = 0 and
    ((f1_1, ε_f1_1), (f1_2, ε_f1_2), (f1_3, ε_f1_3) ∈ S_F1)
    ∨
    recursive_F_search(S_F1, λ + 1, p()) == F_IN_EVERY_CT_LINE
    ))
    return F_IN_EVERY_CT_LINE
  else
    return FAILED
```

`recursive_F_search()` is meant to be initially called with the parameters $S_{Finit} = \{\}$, $\lambda = 0$, $p() = \{1, ..., n\}$ [5]. If there is at least one initially false clause in each CT line, then `recursive_F_search()` will return `F_IN_EVERY_CT_LINE`.

This becomes obvious when examining which S_{F0} and S_{F1} sets `recursive_F_search()` does search for initially false clauses:

At first, `recursive_F_search()` checks if there are two initially false clauses $F0$ and $F1$ within:

$\texttt{US}_1 = \{0\}$
$\texttt{US}_2 = \{1\}$

This is not yet useful because we need at least three U_x values in each CT line. Therefore `recursive_F_search()` must do further recursions. This is the case because *three* SAT clauses with three distinct literals must 'fit' into the CT lines. Therefore the polynomial solver does in practice start with a non-empty S_{Finit}, as it will be shown.

After one recursion, `recursive_F_search()` checks if there is an initially false clause in each of the CT lines belonging to these underlying solutions US_1 to US_4:

$\texttt{US}_1 = \{0, 0\}$
$\texttt{US}_2 = \{0, 1\}$
$\texttt{US}_3 = \{1, 0\}$
$\texttt{US}_4 = \{1, 1\}$

[4] This is a basic supposition of this proof.

[5] The index of $p()$ is 0-based. This means the first element of $p()$ is accessed via the index 0: $p(0) = 1$. This is done for compatibility with an upcoming, later definition of $p()$ used by the polynomial solver. Furthermore, n is the 3-SAT CNF's literal index range, as defined in 2.1.

After two recursions the same check is done for:

$\texttt{US}_1 = \{0, 0, 0\}$
$\texttt{US}_2 = \{0, 0, 1\}$
$\texttt{US}_3 = \{0, 1, 0\}$
$\texttt{US}_4 = \{0, 1, 1\}$
$\texttt{US}_5 = \{1, 0, 0\}$
$\texttt{US}_6 = \{1, 0, 1\}$
$\texttt{US}_7 = \{1, 1, 0\}$
$\texttt{US}_8 = \{1, 1, 1\}$

After three recursions:

$\texttt{US}_1 = \{0, 0, 0, 0\}$
$\texttt{US}_2 = \{0, 0, 0, 1\}$
$\texttt{US}_3 = \{0, 0, 1, 0\}$
$\texttt{US}_4 = \{0, 0, 1, 1\}$
$\texttt{US}_5 = \{0, 1, 0, 0\}$
$\texttt{US}_6 = \{0, 1, 0, 1\}$
$\texttt{US}_7 = \{0, 1, 1, 0\}$
$\texttt{US}_8 = \{0, 1, 1, 1\}$
$\texttt{US}_9 = \{1, 0, 0, 0\}$
$\texttt{US}_{10} = \{1, 0, 0, 1\}$
$\texttt{US}_{11} = \{1, 0, 1, 0\}$
$\texttt{US}_{12} = \{1, 0, 1, 1\}$
$\texttt{US}_{13} = \{1, 1, 0, 0\}$
$\texttt{US}_{14} = \{1, 1, 0, 1\}$
$\texttt{US}_{15} = \{1, 1, 1, 0\}$
$\texttt{US}_{16} = \{1, 1, 1, 1\}$

After four recursions:

$\texttt{US}_1 = \{0, 0, 0, 0, 0\}$
$\texttt{US}_2 = \{0, 0, 0, 0, 1\}$
$\texttt{US}_3 = \{0, 0, 0, 1, 0\}$
$\texttt{US}_4 = \{0, 0, 0, 1, 1\}$
$\texttt{US}_5 = \{0, 0, 1, 0, 0\}$
$\texttt{US}_6 = \{0, 0, 1, 0, 1\}$
$\texttt{US}_7 = \{0, 0, 1, 1, 0\}$
$\texttt{US}_8 = \{0, 0, 1, 1, 1\}$
$\texttt{US}_9 = \{0, 1, 0, 0, 0\}$
$\texttt{US}_{10} = \{0, 1, 0, 0, 1\}$
$\texttt{US}_{11} = \{0, 1, 0, 1, 0\}$
$\texttt{US}_{12} = \{0, 1, 0, 1, 1\}$
$\texttt{US}_{13} = \{0, 1, 1, 0, 0\}$
$\texttt{US}_{14} = \{0, 1, 1, 0, 1\}$
$\texttt{US}_{15} = \{0, 1, 1, 1, 0\}$
$\texttt{US}_{16} = \{0, 1, 1, 1, 1\}$
$\texttt{US}_{17} = \{1, 0, 0, 0, 0\}$
$\texttt{US}_{18} = \{1, 0, 0, 0, 1\}$
$\texttt{US}_{19} = \{1, 0, 0, 1, 0\}$
$\texttt{US}_{20} = \{1, 0, 0, 1, 1\}$
$\vdots$

And so on.

It is recognizable that at maximal recursion depth, the S_{F0}'s and S_{F1}'s in all sub calls of `recursive_F_search()` contain taken together all 2^n underlying solutions. At the latest then an initially false F_0 and F_1 will be found in each recursive sub call. Each recursive sub call will return `F_IN_EVERY_CT_LINE` and finally, when all recursive sub calls returned, the initial, first call of `recursive_F_search()` will return `F_IN_EVERY_CT_LINE`.

The upcoming proof has to show that the polynomial solver, as defined in 3, has the following crucial properties of `recursive_F_search()`:

recursion) If $F0$ was not found, `recursive_F_search()` performs one recursive call of itself. Similarly, if $F1$ was not found, `recursive_F_search()` performs one recursive call of itself. This working mechanism of `recursive_F_search()` is to be called *the recursion*.

extension) In the recursive calls, S_F has become S_{F0} respectively S_{F1}. S_{F0} has been built out of S_F by adding a 0. S_{F1} has been built out of S_F by adding a 1. This means $F0$ must be out of $S_{F0} = S_F \cup (p(\lambda), 0)$. Similarly, $F1$ must be out of $S_{F1} = S_F \cup (p(\lambda), 1)$. This working mechanism of `recursive_F_search()` is to be called *the extension*. It means the set S_F is extended for each recursion.

Beyond that, the polynomial solver needs an additional feature: It was presented that `recursive_F_search()` starts with $S_{Finit} = \{\}$ and $p() = \{1, ..., n\}$. The polynomial solver starts with $S_{Finit} = \{(j_{01}, \epsilon_{j_0 1}), (j_{02}, \epsilon_{j_0 2}), (j_{03}, \epsilon_{j_0 3}), (k_{01}, \epsilon_{k_0 1}), (k_{02}, \epsilon_{k_0 2}), (k_{03}, \epsilon_{k_0 3})\}$ and some special $p()$ which will be defined in the upcoming passage 4.2.2.2.

Here two conventions are important. First, j_{0x} and k_{0y} with $x, y \in \{1, 2, 3\}$ designate the literals of the such called *basis tuple* (J_0, K_0). The basis tuple is the currently regarded tuple the polynomial solver must disable in this proof (see also next paragraph). The clauses and their literals of the basis tuple do always get the (top-level) index 0, as just shown. The two clauses of the basis tuple are called the basis J and the basis K, or just the basis clauses.

Secondly, the polynomial solver starts with $S_{Finit} = \{(j_{01}, \epsilon_{j_0 1}), (j_{02}, \epsilon_{j_0 2}), (j_{03}, \epsilon_{j_0 3}), (k_{01}, \epsilon_{k_0 1}), (k_{02}, \epsilon_{k_0 2}), (k_{03}, \epsilon_{k_0 3})\}$ because only the CT lines containing the basis J and K will be searched for initially false clauses. This is the case because the polynomial solver finally determines "solvable" or "unsatisfiable" by checking the enabled state of tuples. This has been defined in 3. Therefore this proof will show the polynomial solver will disable any basis tuple (J_0, K_0) if there is at least one initially false clause in each CT line containing J_0 and K_0. It will be observable that this is no fatal restriction, the polynomial solver will disable any (J_0, K_0) also when using the stated non-empty S_{Finit}.

An example involving S_{Finit}: We assume $J_0 = (0x_1 \vee 0x_2 \vee 1x_4)$ and $K_0 = (0x_1 \vee 0x_2 \vee 1x_4)$ and $n = 5$. Then $S_{Finit} = \{(1, 0), (2, 0), (4, 1), (1, 0), (2, 0), (4, 1)\}$. The CT lines related to the following underlying solutions US_x must be searched for initially false clauses F:

$\texttt{US}_3 = \{0, 0, 0, 1, 0\}$
$\texttt{US}_4 = \{0, 0, 0, 1, 1\}$
$\texttt{US}_7 = \{0, 0, 1, 1, 0\}$
$\texttt{US}_8 = \{0, 0, 1, 1, 1\}$

In this example, the polynomial solver has to use an S_{Finit} which contains doubled elements, which are $(1, 0)$, $(2, 0)$, $(4, 1)$. In practice this does not cause any problems because the count of elements in S_{Finit} is never checked by the polynomial solver, only *if* some literal index- and epsilon tuple is in S_{Finit} or not will be of importance.

The goal of the following proof is to show that the polynomial solver, as defined in 3, implements the functionality of `recursive_F_search()`. In particular, it must be shown that if there is no initially false F, then the polynomial solver seeks recursively for $F0$ in S_{F0} and $F1$ in S_{F1}. S_{F0} must here contain all literal index- and epsilon tuples of S_F, plus $(p(\lambda), 0)$. S_{F1} must here contain all literal index- and epsilon tuples of S_F, plus $(p(\lambda), 1)$.

Because we know `recursive_F_search()` reliably detects that there is at least one initially false clause in each CT line, we can then derive that also the polynomial solver has this ability. This means the polynomial solver reliably detects that there is at least one initially false clause in each CT line containing the basis tuple (J_0, K_0). So `recursive_F_search()` is here in this proof used as a preferably simple auxiliary function to prove the correctness of the polynomial solver.

4.2.1.3 How the Polynomial Solver Implements Recursion and Extension in General

In this topic an overview is given how the polynomial solver does implement the recursion and the extension in general. By now, merely the basic concepts will be explained. A detailed analysis and especially a comparison between `recursive_F_search()` and the polynomial solver will follow later on.

The basic mechanisms of the polynomial solver to implement the recursion and the extension are the following ones:

recursion) It has been defined in the prior topic 4.2.1.2 that `recursive_F_search()` implements the recursion by this means: "If $F0$ was not found, `recursive_F_search()` performs one recursive call of itself. Similarly, if $F1$ was not found, `recursive_F_search()` performs one recursive call of itself. This working mechanism of `recursive_F_search()` is to be called *the recursion*."

While the polynomial solver does not use recursive procedure calls, it can disable tuples recursively. This means, it disables any tuple (J, K) if there are at least two further disabled tuples, of which each could have been disabled by further disabled tuples, and so on. How this recursive tuple disabling is implemented will now be explained.

We suppose RULE 1 cannot disable (J, K) as there's no single initially false I being contained within (J, K). Furthermore we suppose also RULE 2 cannot disable (J, K) as there are no initially false I and H being contained within (J, K). Even in this case $\pi(J, K) := 0$ is possible, even though all possibilities of RULE 1 and RULE 2 were exhausted. The reason is that one or more of the tuples (I, J), (I, K), (H, J), (H, K) regarded by RULE 2 might get disabled in the same way as (J, K). So we must take into consideration a *recursive* usage of RULE 2. Is is very important to notice that the recursion is *not* purposely implemented but happens in practice because clause tuples depend on each other in what concerns their enabled state. Within this proof, merely a recursive *examination* of tuple dependencies will be performed. The polynomial solver itself does *not* use recursive procedure calling in its source code. Therefore this 'recursion' supported by the polynomial solver is here called the *practical recursion*.

The recursive dependencies of tuple enabled states can be derived from the third `if ()` condition of RULE 2. This condition requests that $(\pi(I, J) = 0 \vee \pi(I, K) = 0) \wedge (\pi(H, J) = 0 \vee \pi(H, K) = 0)$. The disabling of each of these four tuples (I, J), (I, K), (H, J), (H, K) needs further initially false clauses. In the following consideration, these further initially false clauses are called $I1$, $H1$, $I2$, $H2$, $I3$, $H3$, $I4$, $H4$. The disabling operations can be performed on the following conditions:

$\pi(\texttt{I}, \texttt{J}) := \texttt{0}$ `if` $(\pi(\texttt{I1}, \texttt{I}) = \texttt{0} \vee \pi(\texttt{I1}, \texttt{J}) = \texttt{0}) \wedge (\pi(\texttt{H1}, \texttt{I}) = \texttt{0} \vee \pi(\texttt{H1}, \texttt{J}) = \texttt{0})$
$\pi(\texttt{I}, \texttt{K}) := \texttt{0}$ `if` $(\pi(\texttt{I2}, \texttt{I}) = \texttt{0} \vee \pi(\texttt{I2}, \texttt{K}) = \texttt{0}) \wedge (\pi(\texttt{H2}, \texttt{I}) = \texttt{0} \vee \pi(\texttt{H2}, \texttt{K}) = \texttt{0})$
$\pi(\texttt{H}, \texttt{J}) := \texttt{0}$ `if` $(\pi(\texttt{I3}, \texttt{H}) = \texttt{0} \vee \pi(\texttt{I3}, \texttt{J}) = \texttt{0}) \wedge (\pi(\texttt{H3}, \texttt{H}) = \texttt{0} \vee \pi(\texttt{H3}, \texttt{J}) = \texttt{0})$
$\pi(\texttt{H}, \texttt{K}) := \texttt{0}$ `if` $(\pi(\texttt{I4}, \texttt{H}) = \texttt{0} \vee \pi(\texttt{I4}, \texttt{K}) = \texttt{0}) \wedge (\pi(\texttt{H4}, \texttt{H}) = \texttt{0} \vee \pi(\texttt{H4}, \texttt{K}) = \texttt{0})$

These conditions were gained just by replacing the variable names in RULE 2's pseudo code line "`if` $((\pi(I, J) = 0 \vee \pi(I, K) = 0) \wedge (\pi(H, J) = 0 \vee \pi(H, K) = 0))$ `[then]` $(J, K) := 0$" and placing the consequent before the `if` keyword.

The dependencies, i.e. which tuple gets disable by which further tuples, can be visualized as follows:

```
(J,K) can be disabled by
      (I,J) [with I and H being contained within (J,K)] can be disabled by
         (I1,I) [with I1 and H1 being contained within (I,J)]
         and
         (H1,I) [with I1 and H1 being contained within (I,J)]
      or
      (I,K) [with I and H being contained within (J,K)] can be disabled by
         (I2,I) [with I2 and H2 being contained within (I,K)]
         and
         (H2,I) [with I2 and H2 being contained within (I,K)]
   and
      (H,J) [with I and H being contained within (J,K)] can be disabled by
         (I3,H) [with I3 and H3 being contained within (H,J)]
         and
         (H3,H) [with I3 and H3 being contained within (H,J)]
      or
      (H,K) [with I and H being contained within (J,K)] can be disabled by
         (I4,H) [with I4 and H4 being contained within (H,K)]
         and
         (H4,H) [with I4 and H4 being contained within (H,K)]
```

This recursion can of course be continued, it does in practice not have to end after one practical recursion step like just shown in the visualization.

Figuratively, it can be determined: While `recursive_F_search()` checks if the return values of its two recursive procedure calls are both `F_IN_EVERY_CT_LINE`, the polynomial solver's analog is to check if tuples out of (I, J), (I, K), (H, J), (H, K) have been disabled.

The depth of the practical recursion will be denoted as λ. This λ grows by one each time one clause tuple out of $(\pi(I, J) = 0 \vee \pi(I, K) = 0) \wedge (\pi(H, J) = 0 \vee \pi(H, K) = 0)$ is turned into the tuple to be disabled in the next deeper practical recursion layer [6].

A tuple to be disabled by RULE 2 in a practical recursion depth λ will often be called (J_λ, K_λ). So clauses can have an index showing in which practical recursion depth the clauses are valid resp. examined. Details on clause index-ing will follow.

extension) The extension requests that S_{F0} gets extended by $(p(\lambda), 0)$. Similarly, the extension requests that S_{F1} gets extended by $(p(\lambda), 1)$. This is the formal description of the extension given in 4.2.1.2.

In the upcoming passage "Formulas Describing Procedures within Polynomial Solver" (4.2.3), it will be shown that the set the literal index values of $F0_\lambda$ and $F1_\lambda$ (which are the $F0$ and $F1$ at some recursion depth λ) can be chosen from is the corresponding set provided in $\lambda - 1$, extended by $p(\lambda)$. This comes from the fact that RULE 2 turns each of the I_λ and H_λ into the new $J_{\lambda+1}$, as described in the prior topic "recursion)". Because I_λ and H_λ are contained within (J_λ, K_λ), they may have an additional literal index at some position p. This is allowed by the definition of being contained, please see 2.7. This position p will, figuratively spoken, be 'built-in' into the set the literal index values of the initially false $F0_{\lambda+1}$ and $F1_{\lambda+1}$ can be chosen from. So the literal with index p requested by RULE 2 does finally implement the extension. Again, the exact working mechanism will become clear in the proof passage "Formulas Describing Procedures within Polynomial Solver", which is soon to follow.

[6] In this proof, counting the practical recursion depth is the only meaning of λ, regardless of how λ is used in common mathematical literature.

4.2.2 Artifacts for the Proof

At next, some definitions and observations are presented which will be used in the actual proof.

4.2.2.1 Literal Indices a, b, c, d, e, f

Definition 4.2.2 *The literal indices a, b, c, d, e, f are defined as follows:*

$a, b, c, d, e, f \in \{1, 2, 3\} \mid (a \neq b) \wedge (a \neq c) \wedge (b \neq c) \wedge (d \neq e) \wedge (d \neq f) \wedge (e \neq f)$

In the course of the proof, clauses will be characterized by defining sets the clauses' literal index- and epsilon values can be selected from. Sometimes two of a clause's literal index- and epsilon values are to be selected from an other set than the third literal's values. The variables a, b, c, d, e, f do here allow a flexible definition. With their use it is possible to allow all three possible assignments of one of the three clause literals to the one set, and the resting two literals to the other set. For instance:

$\mathtt{I}_1 \mid ((\mathtt{i}_{1\mathtt{a}}, \epsilon_{\mathtt{i}_{1\mathtt{a}}}), (\mathtt{i}_{1\mathtt{b}}, \epsilon_{\mathtt{i}_{1\mathtt{b}}}) \in$
$\{(\mathtt{j}_{0_1}, \epsilon_{\mathtt{j}_{0_1}}), (\mathtt{j}_{0_2}, \epsilon_{\mathtt{j}_{0_2}}), (\mathtt{j}_{0_3}, \epsilon_{\mathtt{j}_{0_3}}), (\mathtt{k}_{0_1}, \epsilon_{\mathtt{k}_{0_1}}), (\mathtt{k}_{0_2}, \epsilon_{\mathtt{k}_{0_2}}), (\mathtt{k}_{0_3}, \epsilon_{\mathtt{k}_{0_3}}), (\mathtt{p}(0), \epsilon_{\mathtt{p}(0)})\} \wedge$
$(\mathtt{i}_{1\mathtt{c}}, \epsilon_{\mathtt{i}_{1\mathtt{c}}}) = (\mathtt{p}(1), 0))$

This means exactly one of these situations applies: $i_{11} = p(1) \vee i_{12} = p(1) \vee i_{13} = p(1)$. The resting two $i_{1\ldots}$ are chosen out of $\{j_{01}, j_{02}, j_{03}, k_{01}, k_{02}, k_{03}, p(0)\}$. The epsilons are chosen accordingly.

The scope of the values of a, b, c, d, e, f is the current clause only. In the following proof, large formulas will be introduced which contain a, b, c and d, e, f repetitively within several clauses. Each assignment to the six variables can be re-chosen for each clause. Mostly a, b, c is assigned to I or $F0$ and d, e, f is assigned to H or $F1$, which appear nearby. The six variables are just repeated and reused to avoid a confusingly excessive variable count.

4.2.2.2 p(Lambda)

Definition 4.2.3 *$p()$ is defined as set which contains the values of all literal indices which do not appear in the basis J_0 and do not appear in the basis K_0:*

$p = \{l \in \{1, ..., n\} \mid (l \neq j_{01} \wedge l \neq j_{02} \wedge l \neq j_{03} \wedge l \neq k_{01} \wedge l \neq k_{02} \wedge l \neq k_{03})\}$

There are $m = |p|$ many elements in $p()$.

Example: We suppose $J_0 = (0x_2 \vee 0x_3 \vee 0x_5)$ and $K_0 = (0x_3 \vee 0x_5 \vee 1x_7)$ and $n = 10$. Then $p = \{1, 4, 6, 8, 9, 10\}$ and $m = 6$.

$p(x)$ shall represent the (x+1)-th element from the set $p()$. This means the index of $p()$ is 0-based. Like this, the practical recursion depth λ can directly be used as index, because λ is also 0-based.

$p()$ has the purpose to avoid that the polynomial solver tries, in the context of the extension, to add literal index- and epsilon values which are already part of the basis J_0 or K_0.

4.2.2.3 Clauses, Literals and their Indices

During the proof, letters are used to denote clauses. These clause variables are mostly of the form $C\#_\lambda$. Here C is a placeholder for the clause name, e.g. I, H, J, K, F. Clause names are typically noted in capital letters and # is occasionally used as a numeration. That # numeration can be 0 or 1, e.g. in $F0$ and $F1$. This means $F0$ has been extended by an epsilon value of 0. Similarly, $F1$ has been

extended by an epsilon value of 1. Else the # numeration is just a serial number, like in $I1$, $I2$, $I3$, $I4$, which have been used in the definition of the practical recursion. λ is the practical recursion depth the clause is used in. λ can be 0 for the basis clauses, or 1, or 2, or it is just "λ" for some practical recursion depth not specified.

As mentioned in 2.1, each clause consists of literals with related epsilons. Literals as a whole are index-ed "x" variables. In this document, also the term literal *indices* is used. A literal index embraces all indices attaches to a literal's "x", but without the "x". The literal indices belonging to the literals of a clause C are typically of the form c_y or $c_{\lambda y}$. These literal indices are noted in non-capital letters and they can also have a practical recursion depth index λ, like the related clause. $\lambda = 0$ does here denote the index of a literal of a basis clause. In any case literal indices have a sub-index y out of $\{1, 2, 3, a, b, c, d, e, f\}$. This sub-index is either the location of the literal in the clause (1, 2 or 3) or it is an index as introduced in 4.2.2.1. This means the literal indices of $C = (0x_1 \vee 0x_3 \vee 1x_5)$ are 1, 3, 5 and the literals of C are x_1, x_3, x_5. It is important to not confuse the literal as a whole with its literal index, which does not include the "x". Each clause literal may be combined with its corresponding epsilon value. For instance, $\epsilon_{j_{\lambda y}}$ is the epsilon value (0 or 1) which stands, within the clause J_λ, in front of the literal $x_{j_{\lambda y}}$.

The literal indices of the clauses of the basis tuple (J_0, K_0) are of great importance for the proof. These literal indices have the form j_{0_y} resp. k_{0_y}, with $y \in \{1, 2, 3, a, b, c, d, e, f\}$. Recall the basis tuple is the tuple the polynomial solver needs to disable throughout this proof. The basis J_0 is written-out: $J_0 = (\epsilon_{j_{01}} x_{j_{01}} \vee \epsilon_{j_{02}} x_{j_{02}} \vee \epsilon_{j_{03}} x_{j_{03}})$. It contains the epsilon values $\epsilon_{j_{01}}, \epsilon_{j_{02}}, \epsilon_{j_{03}}$ and the literal indices j_{01}, j_{02}, j_{03}. Similarly, the basis K_0 is written-out: $K_0 = (\epsilon_{k01} x_{k01} \vee \epsilon_{k02} x_{k02} \vee \epsilon_{k03} x_{k03})$. It contains the epsilon values $\epsilon_{k_{01}}, \epsilon_{k_{02}}, \epsilon_{k_{03}}$ and the literal indices k_{01}, k_{02}, k_{03}.

4.2.2.4 Being Contained

The following two sub proofs will be utilized in the actual proof.

Please notice the denotation convention used exclusively in this "Being Contained" passage: The mere expression "literal" is used for the entirety of literal index and epsilon value. So "one literal of I_1 is equal to one literal of J_1" means $\exists x, y \in \{1, 2, 3\} : i_{1x} = j_{1y} \wedge \epsilon_{i1x} = \epsilon_{j1y}$.

Claim 4.2.4

$$
\begin{aligned}
&\texttt{Any}\\
&\mathtt{I}_1 \mid ((\mathtt{i}_{1\mathtt{a}}, \epsilon_{\mathtt{i}_{1\mathtt{a}}}), (\mathtt{i}_{1\mathtt{b}}, \epsilon_{\mathtt{i}_{1\mathtt{b}}}) \in\\
&\{(\mathtt{j}_{0_1}, \epsilon_{\mathtt{j}_{0_1}}), (\mathtt{j}_{0_2}, \epsilon_{\mathtt{j}_{0_2}}), (\mathtt{j}_{0_3}, \epsilon_{\mathtt{j}_{0_3}}), (\mathtt{k}_{0_1}, \epsilon_{\mathtt{k}_{0_1}}), (\mathtt{k}_{0_2}, \epsilon_{\mathtt{k}_{0_2}}), (\mathtt{k}_{0_3}, \epsilon_{\mathtt{k}_{0_3}}), (\mathtt{p}(0), \epsilon_{\mathtt{p}(0)})\} \wedge\\
&(\mathtt{i}_{1\mathtt{c}}, \epsilon_{\mathtt{i}_{1\mathtt{c}}}) = (\mathtt{p}(1), 0))\\
&\texttt{and}\\
&\mathtt{H}_1 \mid ((\mathtt{h}_{1\mathtt{d}}, \epsilon_{\mathtt{h}_{1\mathtt{d}}}), (\mathtt{h}_{1\mathtt{e}}, \epsilon_{\mathtt{h}_{1\mathtt{e}}}) \in\\
&\{(\mathtt{j}_{0_1}, \epsilon_{\mathtt{j}_{0_1}}), (\mathtt{j}_{0_2}, \epsilon_{\mathtt{j}_{0_2}}), (\mathtt{j}_{0_3}, \epsilon_{\mathtt{j}_{0_3}}), (\mathtt{k}_{0_1}, \epsilon_{\mathtt{k}_{0_1}}), (\mathtt{k}_{0_2}, \epsilon_{\mathtt{k}_{0_2}}), (\mathtt{k}_{0_3}, \epsilon_{\mathtt{k}_{0_3}}), (\mathtt{p}(0), \epsilon_{\mathtt{p}(0)})\} \wedge\\
&(\mathtt{h}_{1\mathtt{f}}, \epsilon_{\mathtt{h}_{1\mathtt{f}}}) = (\mathtt{p}(1), 1))\\
&\texttt{are contained within}\\
&\texttt{either at least one}\\
&\quad \mathtt{J}_1 \mid (\mathtt{j}_{1_1}, \epsilon_{\mathtt{j}_{1_1}}), (\mathtt{j}_{1_2}, \epsilon_{\mathtt{j}_{1_2}}), (\mathtt{j}_{1_3}, \epsilon_{\mathtt{j}_{1_3}}) \in\\
&\quad \{(\mathtt{j}_{0_1}, \epsilon_{\mathtt{j}_{0_1}}), (\mathtt{j}_{0_2}, \epsilon_{\mathtt{j}_{0_2}}), (\mathtt{j}_{0_3}, \epsilon_{\mathtt{j}_{0_3}}), (\mathtt{k}_{0_1}, \epsilon_{\mathtt{k}_{0_1}}), (\mathtt{k}_{0_2}, \epsilon_{\mathtt{k}_{0_2}}), (\mathtt{k}_{0_3}, \epsilon_{\mathtt{k}_{0_3}}), (\mathtt{p}(0), \epsilon_{\mathtt{p}(0)})\}\\
&\quad \texttt{and}\\
&\quad \mathtt{K1}_1 \mid (\mathtt{k1}_{1_1}, \epsilon_{\mathtt{k1}_{1_1}}), (\mathtt{k1}_{1_2}, \epsilon_{\mathtt{k1}_{1_2}}), (\mathtt{k1}_{1_3}, \epsilon_{\mathtt{k1}_{1_3}}) \in\\
&\quad \{(\mathtt{j}_{0_1}, \epsilon_{\mathtt{j}_{0_1}}), (\mathtt{j}_{0_2}, \epsilon_{\mathtt{j}_{0_2}}), (\mathtt{j}_{0_3}, \epsilon_{\mathtt{j}_{0_3}})\}\\
&\texttt{or at least one}\\
&\quad \mathtt{J}_1 \mid (\mathtt{j}_{1_1}, \epsilon_{\mathtt{j}_{1_1}}), (\mathtt{j}_{1_2}, \epsilon_{\mathtt{j}_{1_2}}), (\mathtt{j}_{1_3}, \epsilon_{\mathtt{j}_{1_3}}) \in\\
&\quad \{(\mathtt{j}_{0_1}, \epsilon_{\mathtt{j}_{0_1}}), (\mathtt{j}_{0_2}, \epsilon_{\mathtt{j}_{0_2}}), (\mathtt{j}_{0_3}, \epsilon_{\mathtt{j}_{0_3}}), (\mathtt{k}_{0_1}, \epsilon_{\mathtt{k}_{0_1}}), (\mathtt{k}_{0_2}, \epsilon_{\mathtt{k}_{0_2}}), (\mathtt{k}_{0_3}, \epsilon_{\mathtt{k}_{0_3}}), (\mathtt{p}(0), \epsilon_{\mathtt{p}(0)})\}\\
&\quad \texttt{and}\\
&\quad \mathtt{K2}_1 \mid (\mathtt{k2}_{1_1}, \epsilon_{\mathtt{k2}_{1_1}}), (\mathtt{k2}_{1_2}, \epsilon_{\mathtt{k2}_{1_2}}), (\mathtt{k2}_{1_3}, \epsilon_{\mathtt{k2}_{1_3}}) \in\\
&\quad \{(\mathtt{k}_{0_1}, \epsilon_{\mathtt{k}_{0_1}}), (\mathtt{k}_{0_2}, \epsilon_{\mathtt{k}_{0_2}}), (\mathtt{k}_{0_3}, \epsilon_{\mathtt{k}_{0_3}})\}
\end{aligned}
$$

Proof:

- First, it is important to recognize i_{1a}, i_{1b}, i_{1c} and h_{1d}, h_{1e}, h_{1f} will not all be out of $\{j_{0_1}, j_{0_2}, j_{0_3}, k_{01}, k_{02}, k_{03}, p(0)\}$. This is the case because it is explicitly assumed $i_{1_c} = p(1)$ and $h_{1_f} = p(1)$. Therefore it is not required to assign i_{1_c} or h_{1_f} a literal index out of $\{j_{0_1}, j_{0_2}, j_{0_3}, k_{01}, k_{02}, k_{03}, p(0)\}$. So there are only two literal indices of I_1 and two literal indices of H_1 which must be equal to literal indices out of J_1 and ($K1_1$ or $K2_1$).
- The polynomial solver will detect I_1 and H_1 being contained within J_1 and ($K1_1$ or $K2_1$) if:
  - two literals of I_1 are equal to two literals of J_1.
  - one literal of H_1 is equal to one literal of J_1.
  - one literal of H_1 is equal to one literal of either $K1_1$ or $K2_1$.

  This is an excerpt of the possible cases. The same situation with I_1 and H_1 swapped would be accepted as being contained as well. However, it suffices to take into consideration the described sub case.

  The crucial point is that it has been shown it is *not* required that *both* $K1_1$ and $K2_1$ need to have literals equal to I_1 or H_1. This frugal property will be of importance in upcoming proof passages. □

Claim 4.2.5 *At any practical recursion depth $\lambda \geq 2$ it applies:*

$$
\begin{aligned}
&\texttt{Any}\\
&\mathtt{I}_\lambda \mid ((\mathtt{i}_{\lambda_a}, \epsilon_{\mathtt{i}\lambda_a}), (\mathtt{i}_{\lambda_b}, \epsilon_{\mathtt{i}\lambda_b}) \in\\
&\{(\mathtt{j}_{0_1}, \epsilon_{\mathtt{j}_{0_1}}), (\mathtt{j}_{0_2}, \epsilon_{\mathtt{j}_{0_2}}), (\mathtt{j}_{0_3}, \epsilon_{\mathtt{j}_{0_3}}), (\mathtt{k}_{0_1}, \epsilon_{\mathtt{k}_{0_1}}), (\mathtt{k}_{0_2}, \epsilon_{\mathtt{k}_{0_2}}), (\mathtt{k}_{0_3}, \epsilon_{\mathtt{k}_{0_3}}), (\mathtt{p}(0), \epsilon_{\mathtt{p}}(0)), ..., (\mathtt{p}(\lambda-1), \epsilon_{\mathtt{p}(\lambda-1)})\} \wedge\\
&(\mathtt{i}_{\lambda_c}, \epsilon_{\mathtt{i}\lambda_c}) = (\mathtt{p}(\lambda), 0))\\
&\texttt{and}\\
&\mathtt{H}_\lambda \mid ((\mathtt{h}_{\lambda_d}, \epsilon_{\mathtt{h}\lambda_d}), (\mathtt{h}_{\lambda_e}, \epsilon_{\mathtt{h}\lambda_e}) \in\\
&\{(\mathtt{j}_{0_1}, \epsilon_{\mathtt{j}_{0_1}}), (\mathtt{j}_{0_2}, \epsilon_{\mathtt{j}_{0_2}}), (\mathtt{j}_{0_3}, \epsilon_{\mathtt{j}_{0_3}}), (\mathtt{k}_{0_1}, \epsilon_{\mathtt{k}_{0_1}}), (\mathtt{k}_{0_2}, \epsilon_{\mathtt{k}_{0_2}}), (\mathtt{k}_{0_3}, \epsilon_{\mathtt{k}_{0_3}}), (\mathtt{p}(0), \epsilon_{\mathtt{p}}(0)), ..., (\mathtt{p}(\lambda-1), \epsilon_{\mathtt{p}(\lambda-1)})\} \wedge\\
&(\mathtt{h}_{\lambda_f}, \epsilon_{\mathtt{h}\lambda_f}) = (\mathtt{p}(\lambda), 1))\\
&\texttt{are contained within at least one}\\
&\mathtt{J}_\lambda \mid (\mathtt{j}_{\lambda_1}, \epsilon_{\mathtt{j}_{\lambda_1}}), (\mathtt{j}_{\lambda_2}, \epsilon_{\mathtt{j}_{\lambda_2}}), (\mathtt{j}_{\lambda_3}, \epsilon_{\mathtt{j}_{\lambda_3}}) \in\\
&\{(\mathtt{j}_{0_1}, \epsilon_{\mathtt{j}_{0_1}}), (\mathtt{j}_{0_2}, \epsilon_{\mathtt{j}_{0_2}}), (\mathtt{j}_{0_3}, \epsilon_{\mathtt{j}_{0_3}}), (\mathtt{k}_{0_1}, \epsilon_{\mathtt{k}_{0_1}}), (\mathtt{k}_{0_2}, \epsilon_{\mathtt{k}_{0_2}}), (\mathtt{k}_{0_3}, \epsilon_{\mathtt{k}_{0_3}}), (\mathtt{p}(0), \epsilon_{\mathtt{p}}(0)), ..., (\mathtt{p}(\lambda-1), \epsilon_{\mathtt{p}(\lambda-1)})\}\\
&\texttt{and}\\
&\mathtt{K}_\lambda \mid (\mathtt{k}_{\lambda_1}, \epsilon_{\mathtt{k}\lambda_1}), (\mathtt{k}_{\lambda_2}, \epsilon_{\mathtt{k}\lambda_2}), (\mathtt{k}_{\lambda_3}, \epsilon_{\mathtt{k}\lambda_3}) \in\\
&\{(\mathtt{j}_{0_1}, \epsilon_{\mathtt{j}_{0_1}}), (\mathtt{j}_{0_2}, \epsilon_{\mathtt{j}_{0_2}}), (\mathtt{j}_{0_3}, \epsilon_{\mathtt{j}_{0_3}}), (\mathtt{k}_{0_1}, \epsilon_{\mathtt{k}_{0_1}}), (\mathtt{k}_{0_2}, \epsilon_{\mathtt{k}_{0_2}}), (\mathtt{k}_{0_3}, \epsilon_{\mathtt{k}_{0_3}}), (\mathtt{p}(0), \epsilon_{\mathtt{p}}(0)), ..., (\mathtt{p}(\lambda-2), \epsilon_{\mathtt{p}(\lambda-2)})\}
\end{aligned}
$$

The dots "..." represent a succession of $(p(x), \epsilon_{p(x)})$ with $0<x<[\lambda - 1$ resp. $\lambda - 2]$.

Proof: The goal of this proof is to show that it is for any of the stated I_λ and H_λ possible to choose at least one J_λ and K_λ with literal indices and epsilon values from the sets stated in the claim, whereby J_λ and K_λ contain I_λ and H_λ. To do so, first it must be heeded we cannot assign more than summarized three literals of I_λ and H_λ to each J_λ and K_λ. This is because all clauses are exact-3-SAT clauses and thus have exactly 3 literals. So it is important to regard the count of I_λ and H_λ literals being assigned. Additionally it must be recognized K_λ cannot contain any literal index equal to $p(\lambda - 1)$. So a literal of J_λ must be used therefore. K_λ's literal index range goes up to $p(\lambda - 2)$ only, as visible in the formula above. Finally it must be noted that one literal index of I_λ *must* be $p(\lambda)$. The same applies to one literal index of H_λ, which *must* be $p(\lambda)$.

Keeping this in mind it can be gathered that J_λ and K_λ must be chosen as follows to contain I_λ and H_λ:

- If one I_λ literal and one H_λ literal has index $p(\lambda - 1)$
  - Those I_λ and H_λ literals (because non-distinct, effectively one) must be equal to some literal in J_λ.
  - The resting one I_λ literal must be equal to some literal in J_λ and
  - the resting one H_λ literal must be equal to some literal in K_λ.
- If one I_λ literal and no H_λ literal has index $p(\lambda - 1)$
  - This one I_λ literal must be equal to to some literal in J_λ.
  - The resting one I_λ literal must be equal to some literal in J_λ and
  - the resting two H_λ literals must be equal to some literals in K_λ.
- If no I_λ literal and one H_λ literal has index $p(\lambda - 1)$
  - This one H_λ literal must be equal to some literal in J_λ.
  - The resting one H_λ literal must be equal to some literal in J_λ and
  - the resting two I_λ literals must be equal to some literals in K_λ.
- If no I_λ literal and no H_λ literal has index $p(\lambda - 1)$
  - The resting two I_λ literals must be equal to some literals in J_λ and
  - the resting two H_λ literals must be equal to some literals in K_λ.

This is an excerpt of the situations in which the polynomial solver accepts I_λ and H_λ being contained within J_λ and K_λ. However, the just stated situations are in practice sufficient to make the polynomial solver work. This is forecasted by theory and I could also not find any errors by computer-aided verification using self-written computer programs. □

4.2.3 Formulas Describing Procedures within Polynomial Solver

Hereafter several "formulas" will be presented. These formulas consist of tuple sets noted using mathematical notation. These tuple sets describe literal index- and epsilon values which specific possible clauses may contain. The superior idea is to describe precisely which initially false clauses or which disabled tuples are accepted by the polynomial solver to disable further tuples.

The formulas are categorized by the practical recursion depth (see 4.2.1.3) in which they are valid.

At first sight it might seem to the reader that the formulas are hard to understand. But they are the easiest way I found to analyze in detail how the polynomial solver internally works. Besides, the formulas are all similar. When having understood one of them, the sense of all will be accessible. After the formulas have been presented, detailed instructions will follow on how to interpret the formulas and how they are used to prove the correctness of the polynomial solver.

4.2.3.1 Lambda = 0

Claim 4.2.6 *The polynomial solver disables at least one tuple (J_0, K_0) with J_0 and K_0 having their literal index- and epsilon values out of the stated sets if the conditions stated in the following formula are fulfilled.*

Notice that (I_0, J_0) at practical recursion depth $\lambda = 0$ is equal to (J_1, K_1) at practical recursion depth $\lambda = 1$. This is just a naming convention, because the tuple being disabled by RULE 1 or RULE 2 is always named (J, K). Please see 3. When one step of the practical recursion is done, in this next deeper

practical recursion layer the tuple which gets disabled shall again be named using the clause names J and K. The same applies to (I_0, K_0), which is equal to another (J_1, K_1). To be able to distinguish (I_0, J_0) and (I_0, K_0), those tuples are here always noted together with (J_1, K_1). This holds true for greater λ as well. Also the comments introduced by `//` apply to the corresponding locations in the formulas for greater λ, too. They will just not be re-added there to save page space.

```
// === At least one of these tuples gets disabled: ===
```
$\pi($
$J_0 \mid ((j_{0\,1}, \epsilon_{j_0\,1}), (j_{0\,2}, \epsilon_{j_0\,2}), (j_{0\,3}, \epsilon_{j_0\,3})) \in$
$\{(j_{0\,1}, \epsilon_{j_0\,1}), (j_{0\,2}, \epsilon_{j_0\,2}), (j_{0\,3}, \epsilon_{j_0\,3})\}$,
$K_0 \mid ((k_{0\,1}, \epsilon_{k_0\,1}), (k_{0\,2}, \epsilon_{k_0\,2}), (k_{0\,3}, \epsilon_{k_0\,3})) \in$
$\{(k_{0\,1}, \epsilon_{k_0\,1}), (k_{0\,2}, \epsilon_{k_0\,2}), (k_{0\,3}, \epsilon_{k_0\,3})\}$
$) := 0$
`if`
`(`
`// === if there is at least one of the following initially false clauses: ===`
`// "case F":`
$(\exists F_0 \mid (\tau(F_0) = 0 \wedge (((f_{0\,1}, \epsilon_{f_0\,1}), (f_{0\,2}, \epsilon_{f_0\,2}), (f_{0\,3}, \epsilon_{f_0\,3})) \in$
$\{(j_{0\,1}, \epsilon_{j_0\,1}), (j_{0\,2}, \epsilon_{j_0\,2}), (j_{0\,3}, \epsilon_{j_0\,3}), (k_{0\,1}, \epsilon_{k_0\,1}), (k_{0\,2}, \epsilon_{k_0\,2}), (k_{0\,3}, \epsilon_{k_0\,3})\})))$
`// ` F_0 `is equal to` I_0 `of "case I"`
$\vee$
`// === or/and at least one of the following disabled tuples: ===`
`// "case I":`
$\pi($ `// some of these tuples can be used as` (I, J) `in RULE 1`
$I_0 = J_1 \mid ((j_{1\,a}, \epsilon_{j_1\,a}), (j_{1\,b}, \epsilon_{j_1\,b}), (j_{1\,c}, \epsilon_{j_1\,c})) \in$
$\{(j_{0\,1}, \epsilon_{j_0\,1}), (j_{0\,2}, \epsilon_{j_0\,2}), (j_{0\,3}, \epsilon_{j_0\,3}), (k_{0\,1}, \epsilon_{k_0\,1}), (k_{0\,2}, \epsilon_{k_0\,2}), (k_{0\,3}, \epsilon_{k_0\,3})\}$,
$J_0 = K_1 \mid ((k_{1\,1}, \epsilon_{k_1\,1}), (k_{1\,2}, \epsilon_{k_1\,2}), (k_{1\,3}, \epsilon_{k_1\,3})) \in$
$\{(j_{0\,1}, \epsilon_{j_0\,1}), (j_{0\,2}, \epsilon_{j_0\,2}), (j_{0\,3}, \epsilon_{j_0\,3})\}$ `// this is the` J_0 `from the tuple` (J_0, K_0) `which gets disabled`
$) = 0$
$\vee$
`// "case I":`
$\pi($ `// some of these tuples can be used as` (I, K) `in RULE 1`
$I_0 = J_1 \mid ((j_{1\,a}, \epsilon_{j_1\,a}), (j_{1\,b}, \epsilon_{j_1\,b}), (j_{1\,c}, \epsilon_{j_1\,c})) \in$
$\{(j_{0\,1}, \epsilon_{j_0\,1}), (j_{0\,2}, \epsilon_{j_0\,2}), (j_{0\,3}, \epsilon_{j_0\,3}), (k_{0\,1}, \epsilon_{k_0\,1}), (k_{0\,2}, \epsilon_{k_0\,2}), (k_{0\,3}, \epsilon_{k_0\,3})\}$,
$K_0 = K_1 \mid ((k_{1\,1}, \epsilon_{k_1\,1}), (k_{1\,2}, \epsilon_{k_1\,2}), (k_{1\,3}, \epsilon_{k_1\,3})) \in$
$\{(k_{0\,1}, \epsilon_{k_0\,1}), (k_{0\,2}, \epsilon_{k_0\,2}), (k_{0\,3}, \epsilon_{k_0\,3})\}$ `// this is the` K_0 `from the tuple` (J_0, K_0) `which gets disabled`
$) = 0$
$\vee$
`// === or if there is one of these disabled tuples ===`
`// ===  or initially false clauses:  ===`
`((`
`// "case I+H":`
$\pi($ `// RECURSION` $\lambda = 0, S_{F0}$`; some of these tuples can be used as` (I, J) `in RULE 2`
$I_0 = J_1 \mid (((j_{1\,a}, \epsilon_{j_1\,a}), (j_{1\,b}, \epsilon_{j_1\,b})) \in$
$\{(j_{0\,1}, \epsilon_{j_0\,1}), (j_{0\,2}, \epsilon_{j_0\,2}), (j_{0\,3}, \epsilon_{j_0\,3}), (k_{0\,1}, \epsilon_{k_0\,1}), (k_{0\,2}, \epsilon_{k_0\,2}), (k_{0\,3}, \epsilon_{k_0\,3})\} \wedge$
$(j_{1\,c}, \epsilon_{j_1\,c}) = (p(0), 0))$, `// EXTENSION` $\lambda = 0, S_{F0}$
$J_0 = K_1 \mid ((k_{1\,1}, \epsilon_{k_1\,1}), (k_{1\,2}, \epsilon_{k_1\,2}), (k_{1\,3}, \epsilon_{k_1\,3})) \in$
$\{(j_{0\,1}, \epsilon_{j_0\,1}), (j_{0\,2}, \epsilon_{j_0\,2}), (j_{0\,3}, \epsilon_{j_0\,3})\}$ `// this is the` J_0 `from the tuple` (J_0, K_0) `which gets disabled`
$) = 0$

$\vee$
// "case I+H":
$\pi($ // RECURSION $\lambda = 0, S_{F0}$; some of these tuples can be used as (I, K) in RULE 2
$I_0 = J_1 \mid (((j_{1a}, \epsilon_{j_1 a}), (j_{1b}, \epsilon_{j_1 b})) \in$
$\{(j_{01}, \epsilon_{j_0 1}), (j_{02}, \epsilon_{j_0 2}), (j_{03}, \epsilon_{j_0 3}), (k_{01}, \epsilon_{k_0 1}), (k_{02}, \epsilon_{k_0 2}), (k_{03}, \epsilon_{k_0 3})\} \wedge$
$(j_{1c}, \epsilon_{j_1 c}) = (p(0), 0)),$ // EXTENSION $\lambda = 0, S_{F0}$
$K_0 = K_1 \mid ((k_{11}, \epsilon_{k_1 1}), (k_{12}, \epsilon_{k_1 2}), (k_{13}, \epsilon_{k_1 3})) \in$
$\{(k_{01}, \epsilon_{k_0 1}), (k_{02}, \epsilon_{k_0 2}), (k_{03}, \epsilon_{k_0 3})\}$ // this is the K_0 from the tuple (J_0, K_0) which gets disabled
$) = 0$
$\vee$
// "case F0/F1":
$(\exists F0_0 \mid (\tau(F0_0) = 0 \wedge (((f0_{0a}, \epsilon_{f0_0 a}), (f0_{0b}, \epsilon_{f0_0 b})) \in$
$\{(j_{01}, \epsilon_{j_0 1}), (j_{02}, \epsilon_{j_0 2}), (j_{03}, \epsilon_{j_0 3}), (k_{01}, \epsilon_{k_0 1}), (k_{02}, \epsilon_{k_0 2}), (k_{03}, \epsilon_{k_0 3})\} \wedge$
$(f0_{0c}, \epsilon_{f0_0 c}) = (p(0), 0))))$ // equal to I_0 of "case I+H"
)
$\wedge$
// === and additionally one of these disabled tuples ===
// === or initially false clauses: ===
(
// "case I+H":
$\pi($ // RECURSION $\lambda = 0, S_{F1}$; some of these tuples can be used as (H, J) in RULE 2
$H_0 = J_1 \mid (((j_{1d}, \epsilon_{j_1 d}), (j_{1e}, \epsilon_{j_1 e})) \in$
$\{(j_{01}, \epsilon_{j_0 1}), (j_{02}, \epsilon_{j_0 2}), (j_{03}, \epsilon_{j_0 3}), (k_{01}, \epsilon_{k_0 1}), (k_{02}, \epsilon_{k_0 2}), (k_{03}, \epsilon_{k_0 3})\} \wedge$
$(j_{1f}, \epsilon_{j_1 f}) = (p(0), 1)),$ // EXTENSION $\lambda = 0, S_{F1}$
$J_0 = K_1 \mid ((k_{11}, \epsilon_{k_1 1}), (k_{12}, \epsilon_{k_1 2}), (k_{13}, \epsilon_{k_1 3})) \in$
$\{(j_{01}, \epsilon_{j_0 1}), (j_{02}, \epsilon_{j_0 2}), (j_{03}, \epsilon_{j_0 3})\}$ // this is the J_0 from the tuple (J_0, K_0) which gets disabled
$) = 0$
$\vee$
// "case I+H":
$\pi($ // RECURSION $\lambda = 0, S_{F1}$; some of these tuples can be used as (H, K) in RULE 2
$H_0 = J_1 \mid (((j_{1d}, \epsilon_{j_1 d}), (j_{1e}, \epsilon_{j_1 e})) \in$
$\{(j_{01}, \epsilon_{j_0 1}), (j_{02}, \epsilon_{j_0 2}), (j_{03}, \epsilon_{j_0 3}), (k_{01}, \epsilon_{k_0 1}), (k_{02}, \epsilon_{k_0 2}), (k_{03}, \epsilon_{k_0 3})\} \wedge$
$(j_{1f}, \epsilon_{j_1 f}) = (p(0), 1)),$ // EXTENSION $\lambda = 0, S_{F1}$
$K_0 = K_1 \mid ((k_{11}, \epsilon_{k_1 1}), (k_{12}, \epsilon_{k_1 2}), (k_{13}, \epsilon_{k_1 3})) \in$
$\{(k_{01}, \epsilon_{k_0 1}), (k_{02}, \epsilon_{k_0 2}), (k_{03}, \epsilon_{k_0 3})\}$ // this is the K_0 from the tuple (J_0, K_0) which gets disabled
$) = 0$
$\vee$
// "case F0/F1":
$(\exists F1_0 \mid (\tau(F1_0) = 0 \wedge (((f1_{0d}, \epsilon_{f1_0 d}), (f1_{0e}, \epsilon_{f1_0 e})) \in$
$\{(j_{01}, \epsilon_{j_0 1}), (j_{02}, \epsilon_{j_0 2}), (j_{03}, \epsilon_{j_0 3}), (k_{01}, \epsilon_{k_0 1}), (k_{02}, \epsilon_{k_0 2}), (k_{03}, \epsilon_{k_0 3})\} \wedge$
$(f1_{0f}, \epsilon_{f1_0 f}) = (p(0), 1))))$ // equal to H_0 of "case I+H"
))
)

Proof:

"case I+H") Disabled tuple(s) contribute to setting $\pi(J_0, K_0) := 0$

It is supposed that $\pi(J_0, K_0) = 1$, that means the basis tuple has not been disabled yet. Otherwise the whole proof consideration here would be irrelevant.

The stated I_0 and H_0 are contained within (J_0, K_0), as defined in 2.7. The reason is that all literal index- and epsilon value tuples of I_0 and H_0 appear in J_0 or K_0 or both, except $(p(0), 0)$ resp. $(p(0), 1)$. These two literal indices $p(0)$ are the position p as defined in 2.7 [7].

[7] The $p(0)$ is an other variable than the p used in the definition of being contained. Please do not get confused about this

Furthermore the formula from this proof's claim defines for "case I+H" that it is given: $(\pi(I_0, J_0) = 0 \wedge \pi(H_0, J_0) = 0) \vee (\pi(I_0, K_0) = 0 \wedge \pi(H_0, K_0) = 0)$.

This fulfills the three `if ()` conditions of RULE 2. Therefore RULE 2 will disable (J_0, K_0).

How RULE 2 does this in practice is now explained.

We look at the exact definition of RULE 2, as given in 3:

```
foreach J ∈ PC
  foreach K ∈ PC
    if (π(J,K) = 1)
      foreach I ∈ PC
        foreach H ∈ PC
          if (I and H are contained within (J,K))
            if ((π(I,J) = 0 ∨ π(I,K) = 0) ∧ (π(H,J) = 0 ∨ π(H,K) = 0))
              π(J,K) := 0
              Changed := true
```

The four `foreach` loops each iterate through absolutely all possible clauses. Therefore it will happen at least once that all four loops point to the J_0, K_0, I_0, H_0 regarded in this proof. As already mentioned, then the three `if ()` conditions are fulfilled and the tuple (J, K), which is (J_0, K_0) in this consideration [8], gets disabled.

The reader might have noticed the pseudo code of the polynomial solver does not contain any $p(\lambda)$. The formulas whereas do. This is no contradiction, for the following reason: the $p(\lambda)$'s shown in this document are always a part of some clauses defined in the formulas of 4.2.6, 4.2.7, 4.2.8, 4.2.9. These clauses are, even when containing any $p(\lambda)$, all out of the set of possible clauses. This means that each of the `foreach` loops of the polynomial solver will 'sooner or later' point to the 'right' clause which does contain the stated $p(\lambda)$. For clarification, an example: The formula of 4.2.6 states that some $F0_0$ and $F1_0$ is accepted to disable (J_0, K_0). The formula also states $F0_0$ and $F1_0$ might contain a literal with index $p(0)$. When the `foreach` $I \in PC$ and `foreach` $H \in PC$ loops of RULE 2 point to these $F0_0$ and $F1_0$, then the three `if ()` conditions are fulfilled and the code line $\pi(J, K) := 0$ will be executed.

Please notice that it *must* apply $(j_{1c}, \epsilon_{j_{1c}}) = (p(0), 0)$ *and* $(j_{1f}, \epsilon_{j_{1f}}) = (p(0), 1)$. So one literal index of I_0 *and* one literal index of H_0 *must* be equal to $p(0)$. The corresponding epsilon value of I_0 must be 0 and the corresponding epsilon value of H_0 must be 1. This is requested in the definition of "being contained", please see 2.7. Because I_0 and H_0 are the two clauses being contained within (J_0, K_0), the described situation must apply.

But what if there is an "asynchrony", meaning either I_0 or H_0 contains $(p(0), 0)$ resp. $(p(0), 1)$, but not both? This does also not cause problems, because then "case I" (see below) applies. If the literal indices and epsilon values of e.g. I_0 (or H_0) do not contain $(p(0), 0)$ (resp. $(p(0), 1)$), then they are all out of $\{(j_{01}, \epsilon_{j_{01}}), (j_{02}, \epsilon_{j_{02}}), (j_{03}, \epsilon_{j_{03}}), (k_{01}, \epsilon_{k_{01}}), (k_{02}, \epsilon_{k_{02}}), (k_{03}, \epsilon_{k_{03}})\}$. These requirements fulfill "case I". If only I_0 contains $(p(0), 0)$, then I_0 can in practice be seen as surplus, because H_0 fulfills "case I". The same applies for the opposite case where only H_0 contains $(p(0), 1)$, then H_0 can be seen as surplus. An initially false I_0 is then 'enough' to disable (J_0, K_0).

"case I") Disabled tuple(s) contribute to setting $\pi(J_0, K_0) := 0$

"Case I" is similar to "case I+H".

It is supposed that $\pi(J_0, K_0) = 1$, that means the basis tuple has not been disabled yet. Otherwise the whole proof consideration here would be irrelevant.

The stated I_0 is contained within the stated (J_0, K_0), as defined in 2.7. The reason is that all literal index- and epsilon value tuples of I_0 appear in J_0 or K_0 or both.

Furthermore the formula 4.2.6 defines for "case I" that it is given: $\pi(I_0, J_0) = 0$ or/and $\pi(I_0, K_0) = 0$.

This fulfills the three `if ()` conditions of RULE 1. Therefore RULE 1 will disable (J_0, K_0).

random accordance.

[8] Clause names can be replaced as long as their content is the same, like for any mathematical variable in general.

How RULE 1 does this in practice is now explained.

We look at the exact definition of RULE 1, as given in 3:

```
foreach J ∈ PC
  foreach K ∈ PC
    if (π(J,K) = 1)
      foreach I ∈ PC
        if (I is contained within (J,K))
          if (π(I,J) = 0 ∨ π(I,K) = 0)
            π(J,K) := 0
            Changed := true
```

The three `foreach` loops each iterate through absolutely all possible clauses. Therefore it will happen at least once that all three loops point to the J_0, K_0, I_0 regarded in this proof. As already mentioned, then the three `if ()` conditions are fulfilled and the tuple (J, K), which is (J_0, K_0) in this consideration (clause names can be substituted), gets disabled.

The reason why here in "case I" of this $\lambda = 0$ formula it is checked for *two* disabled tuples $\pi(I_0, J_0) = 0$ or/and $\pi(I_0, K_0) = 0$ is that at $\lambda = 1$ only one of both could have gotten disabled. Further notes will be given as part of the upcoming formulas for $\lambda = 1$ (4.2.7 and 4.2.8).

"case F0/F1") Initially false clause(s) contribute to setting $\pi(J_0, K_0) := 0$

For "case F0/F1" it is assumed that there are initially false clauses $F0_0$ or/and $F1_0$ instead of disabled tuples.

Because $F0_0$ is initially false, it applies $\tau(F0_0) = 0$. Similarly, because $F1_0$ is initially false, it applies $\tau(F1_0) = 0$.

Then the INITIALIZATION rule has disabled any tuple which holds the clause $F0_0$. This means: $\forall X \in PC : \pi(F0_0, X) = 0$. Similarly, the INITIALIZATION rule has disabled any tuple which holds the clause $F1_0$. This means: $\forall X \in PC : \pi(F1_0, X) = 0$.

How the INITIALIZATION rule does this in practice is now explained.

We look at the exact definition of the INITIALIZATION rule, as given in 3:

```
foreach J ∈ PC
  foreach K ∈ PC
    if ((τ(J) = 1) ∧ (τ(K) = 1) ∧ (J ≡ K))
      π(J,K) := 1
    if ((τ(J) = 0) ∨ (τ(K) = 0) ∨ (J ≢ K))
      π(J,K) := 0
```

The two `foreach` loops each iterate through absolutely all possible clauses. Therefore it will happen at least once that both loops point to the ($F0_0$ or $F1_0$) and X regarded in this proof. Then the second `if ()` condition is fulfilled and the tuple (J, K), which is $(F0_0, X)$ or $(F1_0, X)$ in this consideration, gets disabled.

When regarding carefully how the literal index- and epsilon values of $F0_0$ can be chosen (see formula of claim 4.2.6), one can determine this allows the very same selection as for I_0 in "case I+H". Similarly, when regarding carefully how the literal index- and epsilon values of $F1_0$ can be chosen (again see formula), one can determine this allows the very same selection as for H_0 in "case I+H".

This means there is always some I_0 from "case I+H" which is equal to $F0_0$. Also, there is always some H_0 from "case I+H" which is equal to $F1_0$.

Because we already found out $\forall X \in PC : \pi(F0_0, X) = 0$, we can conclude: $\forall X \in PC : \pi(I_0, X) = 0$ and finally $\pi(I_0, J_0) = 0$, because there is one X equal to J_0. Similarly, because we already know $\forall X \in PC : \pi(F1_0, X) = 0$, we can conclude: $\forall X \in PC : \pi(H_0, X) = 0$ and finally $\pi(H_0, J_0) = 0$, because there is one X equal to J_0.

Now there is the same situation as in "case I+H". There are I_0 and H_0 being contained within (J_0, K_0) and it applies $\pi(I_0, J_0) = 0 \wedge \pi(H_0, J_0) = 0$. Then the same procedure as explained in "case I+H" is performed to disable the tuple (J_0, K_0).

But what if there is an "asynchrony", meaning either $F0_0$ or $F1_0$ contains $(p(0), 0)$ resp. $(p(0), 1)$, but not both? This does also not cause problems, because then "case F" (see below) applies. If the literal indices and epsilon values of e.g. $F0_0$ (or $F1_0$) do not contain $(p(0), 0)$ (resp. $(p(0), 1)$), then they are all out of $\{(j_{0_1}, \epsilon_{j_{0_1}}), (j_{0_2}, \epsilon_{j_{0_2}}), (j_{0_3}, \epsilon_{j_{0_3}}), (k_{0_1}, \epsilon_{k_{0_1}}), (k_{0_2}, \epsilon_{k_{0_2}}), (k_{0_3}, \epsilon_{k_{0_3}})\}$. These requirements fulfill "case I".

"case F") Initially false clause(s) contribute to setting $\pi(J_0, K_0) := 0$

"Case F" is similar to "case F0/F1".

Any of the F_0 stated by the formula matches some of the I_0 of "case I". Any such I_0 is then contained within (J_0, K_0), as pointed out in the previous "case I" text passage.

Because $\tau(F_0) = 0$ and thus $\tau(I_0) = 0$, the INITIALIZATION rule will have set $\pi(I_0, J_0) = 0$. Please compare to the prior proof of "case F0/F1".

Finally the same procedure as explained in "case I" is performed to disable the tuple (J_0, K_0).

Please notice that this "case F" disables (J_0, K_0) if the initially false clause of each CT line is completely contained within (J_0, K_0). This means "case F" covers the single case where no recursion and extension is to be done at all. Please keep this in mind, because the rest of the proof examines exclusively cases which require recursion and extension. □

4.2.3.2 Lambda = 1

Claim 4.2.7 *The polynomial solver disables at least one tuple (I_0, J_0) or (I_0, K_0) with I_0 and J_0 and K_0 having their literal index- and epsilon values out of the stated sets if the conditions stated in the following formula are fulfilled.*

It is not predictable if it is (I_0, J_0) or (I_0, K_0) which gets disabled, but it is for sure that at least one of both gets disabled.

$$
\begin{aligned}
&\pi(\\
&I_0 = J_1 \mid ((j_{1_1}, \epsilon_{j_{1_1}}), (j_{1_2}, \epsilon_{j_{1_2}}), (j_{1_3}, \epsilon_{j_{1_3}})) \in \\
&\{(j_{0_1}, \epsilon_{j_{0_1}}), (j_{0_2}, \epsilon_{j_{0_2}}), (j_{0_3}, \epsilon_{j_{0_3}}), (k_{0_1}, \epsilon_{k_{0_1}}), (k_{0_2}, \epsilon_{k_{0_2}}), (k_{0_3}, \epsilon_{k_{0_3}}), (p(0), 0)\}, \\
&J_0 = K_1 \mid ((k_{1_1}, \epsilon_{k_{1_1}}), (k_{1_2}, \epsilon_{k_{1_2}}), (k_{1_3}, \epsilon_{k_{1_3}})) \in \\
&\{(j_{0_1}, \epsilon_{j_{0_1}}), (j_{0_2}, \epsilon_{j_{0_2}}), (j_{0_3}, \epsilon_{j_{0_3}})\} \\
&) := 0 \\
&\vee \\
&\pi(\\
&I_0 = J_1 \mid ((j_{1_1}, \epsilon_{j_{1_1}}), (j_{1_2}, \epsilon_{j_{1_2}}), (j_{1_3}, \epsilon_{j_{1_3}})) \in \\
&\{(j_{0_1}, \epsilon_{j_{0_1}}), (j_{0_2}, \epsilon_{j_{0_2}}), (j_{0_3}, \epsilon_{j_{0_3}}), (k_{0_1}, \epsilon_{k_{0_1}}), (k_{0_2}, \epsilon_{k_{0_2}}), (k_{0_3}, \epsilon_{k_{0_3}}), (p(0), 0)\}, \\
&K_0 = K_1 \mid ((k_{1_1}, \epsilon_{k_{1_1}}), (k_{1_2}, \epsilon_{k_{1_2}}), (k_{1_3}, \epsilon_{k_{1_3}})) \in \\
&\{(k_{0_1}, \epsilon_{k_{0_1}}), (k_{0_2}, \epsilon_{k_{0_2}}), (k_{0_3}, \epsilon_{k_{0_3}})\} \\
&) := 0 \\
&\texttt{if} \\
&\texttt{(} \\
&\quad \texttt{// "case F":} \\
&\quad (\exists F_1 \mid (\tau(F_1) = 0 \wedge (((f_{1_1}, \epsilon_{f_{1_1}}), (f_{1_2}, \epsilon_{f_{1_2}}), (f_{1_3}, \epsilon_{f_{1_3}})) \in \\
&\quad \{(j_{0_1}, \epsilon_{j_{0_1}}), (j_{0_2}, \epsilon_{j_{0_2}}), (j_{0_3}, \epsilon_{j_{0_3}}), (k_{0_1}, \epsilon_{k_{0_1}}), (k_{0_2}, \epsilon_{k_{0_2}}), (k_{0_3}, \epsilon_{k_{0_3}}), (p(0), 0)\}))) \\
&\quad \vee \\
&\quad \texttt{// "case I":} \\
&\quad \pi(\\
&\quad I_1 = J_2 \mid ((j_{2_a}, \epsilon_{j_{2_a}}), (j_{2_b}, \epsilon_{j_{2_b}}), (j_{2_c}, \epsilon_{j_{2_c}})) \in \\
&\quad \{(j_{0_1}, \epsilon_{j_{0_1}}), (j_{0_2}, \epsilon_{j_{0_2}}), (j_{0_3}, \epsilon_{j_{0_3}}), (k_{0_1}, \epsilon_{k_{0_1}}), (k_{0_2}, \epsilon_{k_{0_2}}), (k_{0_3}, \epsilon_{k_{0_3}}), (p(0), 0)\}, \\
&\quad J_1 = K_2 \mid ((k_{2_1}, \epsilon_{k_{2_1}}), (k_{2_2}, \epsilon_{k_{2_2}}), (k_{2_3}, \epsilon_{k_{2_3}})) \in \\
&\quad \{(j_{0_1}, \epsilon_{j_{0_1}}), (j_{0_2}, \epsilon_{j_{0_2}}), (j_{0_3}, \epsilon_{j_{0_3}}), (k_{0_1}, \epsilon_{k_{0_1}}), (k_{0_2}, \epsilon_{k_{0_2}}), (k_{0_3}, \epsilon_{k_{0_3}}), (p(0), 0)\} \\
&\quad) = 0
\end{aligned}
$$

$\vee$
((
// "case I+H":
$\pi($ // RECURSION $\lambda = 1, S_{F0}$
$I_1 = J_2 \mid (((j_{2a}, \epsilon_{j_{2}a}), (j_{2b}, \epsilon_{j_{2}b})) \in$
$\{(j_{01}, \epsilon_{j_{0}1}), (j_{02}, \epsilon_{j_{0}2}), (j_{03}, \epsilon_{j_{0}3}), (k_{01}, \epsilon_{k_{0}1}), (k_{02}, \epsilon_{k_{0}2}), (k_{03}, \epsilon_{k_{0}3}), (p(0), 0)\} \wedge$
$(j_{2c}, \epsilon_{j_{2}c}) = (p(1), 0))$, // EXTENSION $\lambda = 1, S_{F0}$
$J_1 = K_2 \mid ((k_{21}, \epsilon_{k_{2}1}), (k_{22}, \epsilon_{k_{2}2}), (k_{23}, \epsilon_{k_{2}3})) \in$
$\{(j_{01}, \epsilon_{j_{0}1}), (j_{02}, \epsilon_{j_{0}2}), (j_{03}, \epsilon_{j_{0}3}), (k_{01}, \epsilon_{k_{0}1}), (k_{02}, \epsilon_{k_{0}2}), (k_{03}, \epsilon_{k_{0}3}), (p(0), 0)\}$
$) = 0$
$\vee$
// "case F0/F1":
$(\exists F0_1 \mid (\tau(F0_1) = 0 \wedge (((f0_{1a}, \epsilon_{f0_{1}a}), (f0_{1b}, \epsilon_{f0_{1}b})) \in$
$\{(j_{01}, \epsilon_{j_{0}1}), (j_{02}, \epsilon_{j_{0}2}), (j_{03}, \epsilon_{j_{0}3}), (k_{01}, \epsilon_{k_{0}1}), (k_{02}, \epsilon_{k_{0}2}), (k_{03}, \epsilon_{k_{0}3}), (p(0), 0)\} \wedge$
$(f0_{1c}, \epsilon_{f0_{1}c}) = (p(1), 0))))$
)
$\wedge$
(
// "case I+H":
$\pi($ // RECURSION $\lambda = 1, S_{F1}$
$H_1 = J_2 \mid (((j_{2d}, \epsilon_{j_{2}d}), (j_{2e}, \epsilon_{j_{2}e})) \in$
$\{(j_{01}, \epsilon_{j_{0}1}), (j_{02}, \epsilon_{j_{0}2}), (j_{03}, \epsilon_{j_{0}3}), (k_{01}, \epsilon_{k_{0}1}), (k_{02}, \epsilon_{k_{0}2}), (k_{03}, \epsilon_{k_{0}3}), (p(0), 0)\} \wedge$
$(j_{2f}, \epsilon_{j_{2}f}) = (p(1), 1))$, // EXTENSION $\lambda = 1, S_{F1}$
$J_1 = K_2 \mid ((k_{21}, \epsilon_{k_{2}1}), (k_{22}, \epsilon_{k_{2}2}), (k_{23}, \epsilon_{k_{2}3})) \in$
$\{(j_{01}, \epsilon_{j_{0}1}), (j_{02}, \epsilon_{j_{0}2}), (j_{03}, \epsilon_{j_{0}3}), (k_{01}, \epsilon_{k_{0}1}), (k_{02}, \epsilon_{k_{0}2}), (k_{03}, \epsilon_{k_{0}3}), (p(0), 0)\}$
$) = 0$
$\vee$
// "case F0/F1":
$(\exists F1_1 \mid (\tau(F1_1) = 0 \wedge (((f1_{1d}, \epsilon_{f1_{1}d}), (f1_{1e}, \epsilon_{f1_{1}e})) \in$
$\{(j_{01}, \epsilon_{j_{0}1}), (j_{02}, \epsilon_{j_{0}2}), (j_{03}, \epsilon_{j_{0}3}), (k_{01}, \epsilon_{k_{0}1}), (k_{02}, \epsilon_{k_{0}2}), (k_{03}, \epsilon_{k_{0}3}), (p(0), 0)\} \wedge$
$(f1_{1f}, \epsilon_{f1_{1}f}) = (p(1), 1))))$
))
)

Proof: The four cases can be proven just like done in the $\lambda = 0$ case. Please see 4.2.6. There is only one difference concerning "case I+H": To show that I_1 and H_1 are contained within (J_1, K_1) now 4.2.4 is to be used. Please notice that 4.2.4 cannot forecast if I_1 and H_1 are contained within (J_1, K_1)=(I_0, J_0) or if I_1 and H_1 are contained within (J_1, K_1)=(I_0, K_0). For this reason, this formula 4.2.7 leaves open if (I_0, J_0) or if (I_0, K_0) is the tuple which gets disabled. $\square$

Claim 4.2.8 *The polynomial solver disables at least one tuple (H_0, J_0) or (H_0, K_0) with H_0 and J_0 and K_0 having their literal index- and epsilon values out of the stated sets if the conditions stated in the following formula are fulfilled.*

It is not predictable if it is (H_0, J_0) or (H_0, K_0) which gets disabled, but it is for sure that at least one of both gets disabled.

$\pi($
$H_0 = J_1 \mid ((j_{11}, \epsilon_{j_{1}1}), (j_{12}, \epsilon_{j_{1}2}), (j_{13}, \epsilon_{j_{1}3})) \in$
$\{(j_{01}, \epsilon_{j_{0}1}), (j_{02}, \epsilon_{j_{0}2}), (j_{03}, \epsilon_{j_{0}3}), (k_{01}, \epsilon_{k_{0}1}), (k_{02}, \epsilon_{k_{0}2}), (k_{03}, \epsilon_{k_{0}3}), (p(0), 1)\}$,
$J_0 = K_1 \mid ((k_{11}, \epsilon_{k_{1}1}), (k_{12}, \epsilon_{k_{1}2}), (k_{13}, \epsilon_{k_{1}3})) \in$
$\{(j_{01}, \epsilon_{j_{0}1}), (j_{02}, \epsilon_{j_{0}2}), (j_{03}, \epsilon_{j_{0}3})\}$
$) := 0$

$\vee$
$\pi($
$H_0 = J_1 \mid ((j_{1\,1}, \epsilon_{j_1\,1}), (j_{1\,2}, \epsilon_{j_1\,2}), (j_{1\,3}, \epsilon_{j_1\,3})) \in$
$\{(j_{0\,1}, \epsilon_{j_0\,1}), (j_{0\,2}, \epsilon_{j_0\,2}), (j_{0\,3}, \epsilon_{j_0\,3}), (k_{0\,1}, \epsilon_{k_0\,1}), (k_{0\,2}, \epsilon_{k_0\,2}), (k_{0\,3}, \epsilon_{k_0\,3}), (p(0), 1)\}$,
$K_0 = K_1 \mid ((k_{1\,1}, \epsilon_{k_1\,1}), (k_{1\,2}, \epsilon_{k_1\,2}), (k_{1\,3}, \epsilon_{k_1\,3})) \in$
$\{(k_{0\,1}, \epsilon_{k_0\,1}), (k_{0\,2}, \epsilon_{k_0\,2}), (k_{0\,3}, \epsilon_{k_0\,3})\}$
$) := 0$
`if`
`(`
`// "case F":`
$(\exists F_1 \mid (\tau(F_1) = 0 \wedge (((f_{1\,1}, \epsilon_{f_1\,1}), (f_{1\,2}, \epsilon_{f_1\,2}), (f_{1\,3}, \epsilon_{f_1\,3})) \in$
$\{(j_{0\,1}, \epsilon_{j_0\,1}), (j_{0\,2}, \epsilon_{j_0\,2}), (j_{0\,3}, \epsilon_{j_0\,3}), (k_{0\,1}, \epsilon_{k_0\,1}), (k_{0\,2}, \epsilon_{k_0\,2}), (k_{0\,3}, \epsilon_{k_0\,3}), (p(0), 1)\})))$
$\vee$
`// "case I":`
$\pi($
$I_1 = J_2 \mid ((j_{2\,a}, \epsilon_{j_2\,a}), (j_{2\,b}, \epsilon_{j_2\,b}), (j_{2\,c}, \epsilon_{j_2\,c})) \in$
$\{(j_{0\,1}, \epsilon_{j_0\,1}), (j_{0\,2}, \epsilon_{j_0\,2}), (j_{0\,3}, \epsilon_{j_0\,3}), (k_{0\,1}, \epsilon_{k_0\,1}), (k_{0\,2}, \epsilon_{k_0\,2}), (k_{0\,3}, \epsilon_{k_0\,3}), (p(0), 1)\},$
$J_1 = K_2 \mid ((k_{2\,1}, \epsilon_{k_2\,1}), (k_{2\,2}, \epsilon_{k_2\,2}), (k_{2\,3}, \epsilon_{k_2\,3})) \in$
$\{(j_{0\,1}, \epsilon_{j_0\,1}), (j_{0\,2}, \epsilon_{j_0\,2}), (j_{0\,3}, \epsilon_{j_0\,3}), (k_{0\,1}, \epsilon_{k_0\,1}), (k_{0\,2}, \epsilon_{k_0\,2}), (k_{0\,3}, \epsilon_{k_0\,3}), (p(0), 1)\}$
$) = 0$
$\vee$
`((`
`// "case I+H":`
$\pi($ `// RECURSION` $\lambda = 1, S_{F0}$
$I_1 = J_2 \mid (((j_{2\,a}, \epsilon_{j_2\,a}), (j_{2\,b}, \epsilon_{j_2\,b})) \in$
$\{(j_{0\,1}, \epsilon_{j_0\,1}), (j_{0\,2}, \epsilon_{j_0\,2}), (j_{0\,3}, \epsilon_{j_0\,3}), (k_{0\,1}, \epsilon_{k_0\,1}), (k_{0\,2}, \epsilon_{k_0\,2}), (k_{0\,3}, \epsilon_{k_0\,3}), (p(0), 1)\} \wedge$
$(j_{2\,c}, \epsilon_{j_2\,c}) = (p(1), 0)),$ `// EXTENSION` $\lambda = 1, S_{F0}$
$J_1 = K_2 \mid ((k_{2\,1}, \epsilon_{k_2\,1}), (k_{2\,2}, \epsilon_{k_2\,2}), (k_{2\,3}, \epsilon_{k_2\,3})) \in$
$\{(j_{0\,1}, \epsilon_{j_0\,1}), (j_{0\,2}, \epsilon_{j_0\,2}), (j_{0\,3}, \epsilon_{j_0\,3}), (k_{0\,1}, \epsilon_{k_0\,1}), (k_{0\,2}, \epsilon_{k_0\,2}), (k_{0\,3}, \epsilon_{k_0\,3}), (p(0), 1)\}$
$) = 0$
$\vee$
`// "case F0/F1":`
$(\exists F0_1 \mid (\tau(F0_1) = 0 \wedge (((f0_{1_a}, \epsilon_{f0_{1\,a}}), (f0_{1_b}, \epsilon_{f0_{1\,b}})) \in$
$\{(j_{0\,1}, \epsilon_{j_0\,1}), (j_{0\,2}, \epsilon_{j_0\,2}), (j_{0\,3}, \epsilon_{j_0\,3}), (k_{0\,1}, \epsilon_{k_0\,1}), (k_{0\,2}, \epsilon_{k_0\,2}), (k_{0\,3}, \epsilon_{k_0\,3}), (p(0), 1)\} \wedge$
$(f0_{1_c}, \epsilon_{f0_{1\,c}}) = (p(1), 0))))$
`)`
$\wedge$
`(`
`// "case I+H":`
$\pi($ `// RECURSION` $\lambda = 1, S_{F1}$
$H_1 = J_2 \mid (((j_{2\,d}, \epsilon_{j_2\,d}), (j_{2\,e}, \epsilon_{j_2\,e})) \in$
$\{(j_{0\,1}, \epsilon_{j_0\,1}), (j_{0\,2}, \epsilon_{j_0\,2}), (j_{0\,3}, \epsilon_{j_0\,3}), (k_{0\,1}, \epsilon_{k_0\,1}), (k_{0\,2}, \epsilon_{k_0\,2}), (k_{0\,3}, \epsilon_{k_0\,3}), (p(0), 1)\} \wedge$
$(j_{2\,f}, \epsilon_{j_2\,f}) = (p(1), 1)),$ `// EXTENSION` $\lambda = 1, S_{F1}$
$J_1 = K_2 \mid ((k_{2\,1}, \epsilon_{k_2\,1}), (k_{2\,2}, \epsilon_{k_2\,2}), (k_{2\,3}, \epsilon_{k_2\,3})) \in$
$\{(j_{0\,1}, \epsilon_{j_0\,1}), (j_{0\,2}, \epsilon_{j_0\,2}), (j_{0\,3}, \epsilon_{j_0\,3}), (k_{0\,1}, \epsilon_{k_0\,1}), (k_{0\,2}, \epsilon_{k_0\,2}), (k_{0\,3}, \epsilon_{k_0\,3}), (p(0), 1)\}$
$) = 0$
$\vee$
`// "case F0/F1":`
$(\exists F1_1 \mid (\tau(F1_1) = 0 \wedge (((f1_{1_d}, \epsilon_{f1_{1\,d}}), (f1_{1_e}, \epsilon_{f1_{1\,e}})) \in$
$\{(j_{0\,1}, \epsilon_{j_0\,1}), (j_{0\,2}, \epsilon_{j_0\,2}), (j_{0\,3}, \epsilon_{j_0\,3}), (k_{0\,1}, \epsilon_{k_0\,1}), (k_{0\,2}, \epsilon_{k_0\,2}), (k_{0\,3}, \epsilon_{k_0\,3}), (p(0), 1)\} \wedge$
$(f1_{1\,f}, \epsilon_{f1_1\,f}) = (p(1), 1))))$
`))`
`)`

Proof: The four cases can be proven just like done in the $\lambda = 0$ case. Please see 4.2.6. There is only one difference concerning "case I+H": To show that I_1 and H_1 are contained within (J_1, K_1) now 4.2.4 is to be used. Please notice that 4.2.4 cannot forecast if I_1 and H_1 are contained within (J_1, K_1)=(H_0, J_0) or if I_1 and H_1 are contained within (J_1, K_1)=(H_0, K_0). For this reason, this formula 4.2.8 leaves open if (H_0, J_0) or if (H_0, K_0) is the tuple which gets disabled. □

4.2.3.3 Lambda >= 2

Claim 4.2.9 *The polynomial solver disables at least one tuple (J_λ, K_λ) with J_λ and K_λ having their literal index- and epsilon values out of the stated sets if the conditions stated in the following formula are fulfilled.*

The dots "..." stand for a succession of tuples $(p(\lambda), \epsilon_{p(\lambda)}) \mid 0 \leq \lambda \leq x$, where x is the smallest λ not written out in the corresponding formula line. This means $x = \lambda - 3$ within K_λ, else $x = \lambda - 2$. If λ is smaller than 3, the dots in K_λ are actually surplus and are to be ignored. I just didn't make a case analysis here to keep the formula notation simple.

$\pi($
$J_\lambda \mid ((j_{\lambda 1}, \epsilon_{j_{\lambda 1}}), (j_{\lambda 2}, \epsilon_{j_{\lambda 2}}), (j_{\lambda 3}, \epsilon_{j_{\lambda 3}})) \in$
$\{(j_{01}, \epsilon_{j_0 1}), (j_{02}, \epsilon_{j_0 2}), (j_{03}, \epsilon_{j_0 3}), (k_{01}, \epsilon_{k_0 1}), (k_{02}, \epsilon_{k_0 2}), (k_{03}, \epsilon_{k_0 3}), ..., (p(\lambda - 1), \epsilon_{p(\lambda-1)})\}$,
$K_\lambda \mid ((k_{\lambda 1}, \epsilon_{k_\lambda 1}), (k_{\lambda 2}, \epsilon_{k_\lambda 2}), (k_{\lambda 3}, \epsilon_{k_\lambda 3})) \in$
$\{(j_{01}, \epsilon_{j_0 1}), (j_{02}, \epsilon_{j_0 2}), (j_{03}, \epsilon_{j_0 3}), (k_{01}, \epsilon_{k_0 1}), (k_{02}, \epsilon_{k_0 2}), (k_{03}, \epsilon_{k_0 3}), ..., (p(\lambda - 2), \epsilon_{p(\lambda-2)})\}$
$) := 0$
`if`
`(`
`// "case F":`
$(\exists F_\lambda \mid (\tau(F_\lambda) = 0 \wedge (((f_{\lambda 1}, \epsilon_{f_\lambda 1}), (f_{\lambda 2}, \epsilon_{f_\lambda 2}), (f_{\lambda 3}, \epsilon_{f_\lambda 3})) \in$
$\{(j_{01}, \epsilon_{j_0 1}), (j_{02}, \epsilon_{j_0 2}), (j_{03}, \epsilon_{j_0 3}), (k_{01}, \epsilon_{k_0 1}), (k_{02}, \epsilon_{k_0 2}), (k_{03}, \epsilon_{k_0 3}), ..., (p(\lambda - 1), \epsilon_{p(\lambda-1)})\})))$
$\vee$
`// "case I":`
$\pi($
$I_\lambda = J_{\lambda+1} \mid ((j_{\lambda+1_a}, \epsilon_{j_{\lambda+1_a}}), (j_{\lambda+1_b}, \epsilon_{j_{\lambda+1_b}}), (j_{\lambda+1_c}, \epsilon_{j_{\lambda+1_c}})) \in$
$\{(j_{01}, \epsilon_{j_0 1}), (j_{02}, \epsilon_{j_0 2}), (j_{03}, \epsilon_{j_0 3}), (k_{01}, \epsilon_{k_0 1}), (k_{02}, \epsilon_{k_0 2}), (k_{03}, \epsilon_{k_0 3}), ..., (p(\lambda - 1), \epsilon_{p(\lambda-1)})\}$,
$J_\lambda = K_{\lambda+1} \mid ((k_{\lambda+1_1}, \epsilon_{k_{\lambda+1_1}}), (k_{\lambda+1_2}, \epsilon_{k_{\lambda+1_2}}), (k_{\lambda+1_3}, \epsilon_{k_{\lambda+1_3}})) \in$
$\{(j_{01}, \epsilon_{j_0 1}), (j_{02}, \epsilon_{j_0 2}), (j_{03}, \epsilon_{j_0 3}), (k_{01}, \epsilon_{k_0 1}), (k_{02}, \epsilon_{k_0 2}), (k_{03}, \epsilon_{k_0 3}), ..., (p(\lambda - 1), \epsilon_{p(\lambda-1)})\}$
$) = 0$
$\vee$
`((`
`// "case I+H":`
$\pi($ `// RECURSION` $\lambda \geq 2, S_{F0}$
$I_\lambda = J_{\lambda+1} \mid (((j_{\lambda+1_a}, \epsilon_{j_{\lambda+1_a}}), (j_{\lambda+1_b}, \epsilon_{j_{\lambda+1_b}})) \in$
$\{(j_{01}, \epsilon_{j_0 1}), (j_{02}, \epsilon_{j_0 2}), (j_{03}, \epsilon_{j_0 3}), (k_{01}, \epsilon_{k_0 1}), (k_{02}, \epsilon_{k_0 2}), (k_{03}, \epsilon_{k_0 3}), ..., (p(\lambda - 1), \epsilon_{p(\lambda-1)})\} \wedge$
$(j_{\lambda+1_c}, \epsilon_{j_{\lambda+1_c}}) = (p(\lambda), 0))$, `// EXTENSION` $\lambda \geq 2, S_{F0}$
$J_\lambda = K_{\lambda+1} \mid ((k_{\lambda+1_1}, \epsilon_{k_{\lambda+1_1}}), (k_{\lambda+1_2}, \epsilon_{k_{\lambda+1_2}}), (k_{\lambda+1_3}, \epsilon_{k_{\lambda+1_3}})) \in$
$\{(j_{01}, \epsilon_{j_0 1}), (j_{02}, \epsilon_{j_0 2}), (j_{03}, \epsilon_{j_0 3}), (k_{01}, \epsilon_{k_0 1}), (k_{02}, \epsilon_{k_0 2}), (k_{03}, \epsilon_{k_0 3}), ..., (p(\lambda - 1), \epsilon_{p(\lambda-1)})\}$
$) = 0$
$\vee$
`// "case F0/F1":`
$(\exists F0_\lambda \mid (\tau(F0_\lambda) = 0 \wedge (((f0_{\lambda_a}, \epsilon_{f0_{\lambda a}}), (f0_{\lambda_b}, \epsilon_{f0_{\lambda b}})) \in$
$\{(j_{01}, \epsilon_{j_0 1}), (j_{02}, \epsilon_{j_0 2}), (j_{03}, \epsilon_{j_0 3}), (k_{01}, \epsilon_{k_0 1}), (k_{02}, \epsilon_{k_0 2}), (k_{03}, \epsilon_{k_0 3}), ..., (p(\lambda - 1), \epsilon_{p(\lambda-1)})\} \wedge$
$(f0_{\lambda_c}, \epsilon_{f0_{\lambda c}}) = (p(\lambda), 0))))$
`)`

$\wedge$
(
// "case I+H":
π(// RECURSION $\lambda \geq 2, S_{F1}$
$H_\lambda = J_{\lambda+1} \mid (((j_{\lambda+1_d}, \epsilon_{j_{\lambda+1_d}}), (j_{\lambda+1_e}, \epsilon_{j_{\lambda+1_e}})) \in$
$\{(j_{0_1}, \epsilon_{j_{0_1}}), (j_{0_2}, \epsilon_{j_{0_2}}), (j_{0_3}, \epsilon_{j_{0_3}}), (k_{0_1}, \epsilon_{k_{0_1}}), (k_{0_2}, \epsilon_{k_{0_2}}), (k_{0_3}, \epsilon_{k_{0_3}}), ..., (p(\lambda-1), \epsilon_{p(\lambda-1)})\} \wedge$
$(j_{\lambda+1_f}, \epsilon_{j_{\lambda+1_f}}) = (p(\lambda), 1))$, // EXTENSION $\lambda \geq 2, S_{F1}$
$J_\lambda = K_{\lambda+1} \mid ((k_{\lambda+1_1}, \epsilon_{k_{\lambda+1_1}}), (k_{\lambda+1_2}, \epsilon_{k_{\lambda+1_2}}), (k_{\lambda+1_3}, \epsilon_{k_{\lambda+1_3}})) \in$
$\{(j_{0_1}, \epsilon_{j_{0_1}}), (j_{0_2}, \epsilon_{j_{0_2}}), (j_{0_3}, \epsilon_{j_{0_3}}), (k_{0_1}, \epsilon_{k_{0_1}}), (k_{0_2}, \epsilon_{k_{0_2}}), (k_{0_3}, \epsilon_{k_{0_3}}), ..., (p(\lambda-1), \epsilon_{p(\lambda-1)})\}$
$) = 0$
$\vee$
// "case F0/F1":
$(\exists F1_\lambda \mid (\tau(F1_\lambda) = 0 \wedge (((f1_{\lambda_d}, \epsilon_{f1_{\lambda_d}}), (f1_{\lambda_e}, \epsilon_{f1_{\lambda_e}})) \in$
$\{(j_{0_1}, \epsilon_{j_{0_1}}), (j_{0_2}, \epsilon_{j_{0_2}}), (j_{0_3}, \epsilon_{j_{0_3}}), (k_{0_1}, \epsilon_{k_{0_1}}), (k_{0_2}, \epsilon_{k_{0_2}}), (k_{0_3}, \epsilon_{k_{0_3}}), ..., (p(\lambda-1), \epsilon_{p(\lambda-1)})\} \wedge$
$(f1_{\lambda_f}, \epsilon_{f1_{\lambda_f}}) = (p(\lambda), 1))))$
))
)

Proof: The four cases can be proven just like done in the $\lambda = 0$ case. Please see 4.2.6. There is only one difference concerning "case I+H": To show that I_λ and H_λ are contained within (J_λ, K_λ) now 4.2.5 is to be used. $\square$

4.2.4 Interpretation of the Formulas

In the previous passage 4.2.3 four huge formulas were presented. It will now be explained in detail what these formulas express and how they help to prove the correctness of the polynomial solver.

The formulas all have the same structure: The formulas tell which tuples must be disabled or which initially false clauses must exist to disable some tuple (J_0, K_0) resp. (J_1, K_1) resp. (K_λ, J_λ). The tuple which gets disabled does in each formula always stand before the `if` keyword. The required disabled tuples and initially false clauses do in each formula stand behind the `if` keyword.

Generally, there is no statement of single suitable tuples and clauses but it is stated from which sets the involved clauses' literal index- and epsilon values can be chosen from. Hereto an excerpt taken from 4.2.9:

$\pi($
$J_\lambda \mid ((j_{\lambda 1}, \epsilon_{j_{\lambda 1}}), (j_{\lambda 2}, \epsilon_{j_{\lambda 2}}), (j_{\lambda 3}, \epsilon_{j_{\lambda 3}})) \in$
$\{(j_{0_1}, \epsilon_{j_{0_1}}), (j_{0_2}, \epsilon_{j_{0_2}}), (j_{0_3}, \epsilon_{j_{0_3}}), (k_{0_1}, \epsilon_{k_{0_1}}), (k_{0_2}, \epsilon_{k_{0_2}}), (k_{0_3}, \epsilon_{k_{0_3}}), ..., (p(\lambda-1), \epsilon_{p(\lambda-1)})\}$,
$K_\lambda \mid ((k_{\lambda 1}, \epsilon_{k_{\lambda 1}}), (k_{\lambda 2}, \epsilon_{k_{\lambda 2}}), (k_{\lambda 3}, \epsilon_{k_{\lambda 3}})) \in$
$\{(j_{0_1}, \epsilon_{j_{0_1}}), (j_{0_2}, \epsilon_{j_{0_2}}), (j_{0_3}, \epsilon_{j_{0_3}}), (k_{0_1}, \epsilon_{k_{0_1}}), (k_{0_2}, \epsilon_{k_{0_2}}), (k_{0_3}, \epsilon_{k_{0_3}}), ..., (p(\lambda-2), \epsilon_{p(\lambda-2)})\}$
$) := 0$
`if`
(
...
((
// "case I+H":
π(
$I_\lambda = J_{\lambda+1} \mid (((j_{\lambda+1_a}, \epsilon_{j_{\lambda+1_a}}), (j_{\lambda+1_b}, \epsilon_{j_{\lambda+1_b}})) \in$
$\{(j_{0_1}, \epsilon_{j_{0_1}}), (j_{0_2}, \epsilon_{j_{0_2}}), (j_{0_3}, \epsilon_{j_{0_3}}), (k_{0_1}, \epsilon_{k_{0_1}}), (k_{0_2}, \epsilon_{k_{0_2}}), (k_{0_3}, \epsilon_{k_{0_3}}), ..., (p(\lambda-1), \epsilon_{p(\lambda-1)})\} \wedge$
$(j_{\lambda+1_c}, \epsilon_{j_{\lambda+1_c}}) = (p(\lambda), 0))$,
$J_\lambda = K_{\lambda+1} \mid ((k_{\lambda+1_1}, \epsilon_{k_{\lambda+1_1}}), (k_{\lambda+1_2}, \epsilon_{k_{\lambda+1_2}}), (k_{\lambda+1_3}, \epsilon_{k_{\lambda+1_3}})) \in$
$\{(j_{0_1}, \epsilon_{j_{0_1}}), (j_{0_2}, \epsilon_{j_{0_2}}), (j_{0_3}, \epsilon_{j_{0_3}}), (k_{0_1}, \epsilon_{k_{0_1}}), (k_{0_2}, \epsilon_{k_{0_2}}), (k_{0_3}, \epsilon_{k_{0_3}}), ..., (p(\lambda-1), \epsilon_{p(\lambda-1)})\}$
$) = 0$

$\vee$
// "case F0/F1":
$(\exists F0_\lambda \mid (\tau(F0_\lambda) = 0 \wedge (((f0_{\lambda a}, \epsilon_{f0_\lambda a}), (f0_{\lambda b}, \epsilon_{f0_\lambda b})) \in$
$\{(j_{01}, \epsilon_{j_0 1}), (j_{02}, \epsilon_{j_0 2}), (j_{03}, \epsilon_{j_0 3}), (k_{01}, \epsilon_{k_0 1}), (k_{02}, \epsilon_{k_0 2}), (k_{03}, \epsilon_{k_0 3}), ..., (p(\lambda - 1), \epsilon_{p(\lambda-1)})\} \wedge$
$(f0_{\lambda c}, \epsilon_{f0_\lambda c}) = (p(\lambda), 0))))$
)
...

This formula states that at least one (J_λ, K_λ) gets disabled if there is, among others not shown in this example, some disabled tuple (I_λ, J_λ) or, among others, some initially false clause $F0_\lambda$.

As already mentioned, the formula does not state one exact tuple (J_λ, K_λ) which could get disabled. Instead, the formula describes a set of tuples (J_λ, K_λ). This tuple set is defined via the statement of which literal index- and epsilon values the tuples' clauses might consist. At least one tuple (J_λ, K_λ) will get disabled if the conditions behind the `if` keyword are fulfilled. Similar applies to the required initially false clauses and required disabled tuples behind the `if` keyword. They are not exactly described but any clause(s) out of the stated sets will fulfill the `if ()` condition. The reason why clauses are not exactly specified here is that it is (of course) not known in advance which initially false clauses the 3-SAT CNF to solve will contain. With other initially false clauses, also different tuples might be disabled in any depth of the practical recursion.

What is important to recognize is that at some practical recursion depth $\lambda + 1$, any tuple $(J_{\lambda+1}, K_{\lambda+1})$ which gets disabled is accepted in "case I" or "case I+H" in the next lower practical recursion depth λ. Put another way, this means that at some practical recursion depth $\lambda + 1$, any tuple $(J_{\lambda+1}, K_{\lambda+1})$ which gets disabled is also accepted by RULE 1 or/and RULE 2 as (I, J) or (I, K) or (H, J) or (H, K) (that's how the tuples are named in the definition of RULE 1 and RULE 2, see 3) in the next lower practical recursion depth λ.

For instance: At practical recursion depth $\lambda = 0$, among others, the following tuple can contribute to the disabling of (J_0, K_0):

// At $\lambda = 0$, one of the following tuples (J_0, K_0) gets disabled
$\pi($
$J_0 \mid ((j_{01}, \epsilon_{j_0 1}), (j_{02}, \epsilon_{j_0 2}), (j_{03}, \epsilon_{j_0 3})) \in$
$\{(j_{01}, \epsilon_{j_0 1}), (j_{02}, \epsilon_{j_0 2}), (j_{03}, \epsilon_{j_0 3})\}$,
$K_0 \mid ((k_{01}, \epsilon_{k_0 1}), (k_{02}, \epsilon_{k_0 2}), (k_{03}, \epsilon_{k_0 3})) \in$
$\{(k_{01}, \epsilon_{k_0 1}), (k_{02}, \epsilon_{k_0 2}), (k_{03}, \epsilon_{k_0 3})\}$
$) := 0$
if
(
...
// "case I":
$\pi($ // by one of these tuples (I_0, J_0)
$I_0 = J_1 \mid ((j_{1a}, \epsilon_{j_1 a}), (j_{1b}, \epsilon_{j_1 b}), (j_{1c}, \epsilon_{j_1 c})) \in$
$\{(j_{01}, \epsilon_{j_0 1}), (j_{02}, \epsilon_{j_0 2}), (j_{03}, \epsilon_{j_0 3}), (k_{01}, \epsilon_{k_0 1}), (k_{02}, \epsilon_{k_0 2}), (k_{03}, \epsilon_{k_0 3})\}$,
$J_0 = K_1 \mid ((k_{11}, \epsilon_{k_1 1}), (k_{12}, \epsilon_{k_1 2}), (k_{13}, \epsilon_{k_1 3})) \in$
$\{(j_{01}, \epsilon_{j_0 1}), (j_{02}, \epsilon_{j_0 2}), (j_{03}, \epsilon_{j_0 3})\}$
$) = 0$

$\vee$
...
((
// "case I+H":
π(// or one of these tuples (I_0, J_0)
$I_0 = J_1 \mid (((j_{1a}, \epsilon_{j_1 a}), (j_{1b}, \epsilon_{j_1 b})) \in$
$\{(j_{01}, \epsilon_{j_0 1}), (j_{02}, \epsilon_{j_0 2}), (j_{03}, \epsilon_{j_0 3}), (k_{01}, \epsilon_{k_0 1}), (k_{02}, \epsilon_{k_0 2}), (k_{03}, \epsilon_{k_0 3})\} \wedge$
$(j_{1c}, \epsilon_{j_1 c}) = (p(0), 0))$,
$J_0 = K_1 \mid ((k_{11}, \epsilon_{k_1 1}), (k_{12}, \epsilon_{k_1 2}), (k_{13}, \epsilon_{k_1 3})) \in$
$\{(j_{01}, \epsilon_{j_0 1}), (j_{02}, \epsilon_{j_0 2}), (j_{03}, \epsilon_{j_0 3})\}$
$) = 0$
...

This is an excerpt from 4.2.6.

At the next deeper practical recursion depth $\lambda = 1$, some (J_1, K_1) whose literal index- and epsilon values can be selected as follows gets disabled:

π(// At the next greater $\lambda = 1$, at least one of those (I_0, J_0) gets disabled
$I_0 = J_1 \mid ((j_{11}, \epsilon_{j_1 1}), (j_{12}, \epsilon_{j_1 2}), (j_{13}, \epsilon_{j_1 3})) \in$
$\{(j_{01}, \epsilon_{j_0 1}), (j_{02}, \epsilon_{j_0 2}), (j_{03}, \epsilon_{j_0 3}), (k_{01}, \epsilon_{k_0 1}), (k_{02}, \epsilon_{k_0 2}), (k_{03}, \epsilon_{k_0 3}), (p(0), 0)\}$,
$J_0 = K_1 \mid ((k_{11}, \epsilon_{k_1 1}), (k_{12}, \epsilon_{k_1 2}), (k_{13}, \epsilon_{k_1 3})) \in$
$\{(j_{01}, \epsilon_{j_0 1}), (j_{02}, \epsilon_{j_0 2}), (j_{03}, \epsilon_{j_0 3})\}$
$) := 0$
if
...

This is an excerpt from 4.2.7.

It can be realized that for *every* (J_1, K_1) which gets disabled there is the same (J_1, K_1) accepted to disable (J_0, K_0). This is the case for the following reasons:

- We compare the formulas for $\lambda = 1$ and $\lambda = 0$ (see above). We assume the tuple (I_0, J_0)=(J_1, K_1) being disabled at $\lambda = 1$ contains some literal index- and epsilon value tuple $(p(0), 0)$. Then this tuple (I_0, J_0)=(J_1, K_1) being disabled at $\lambda = 1$ is accepted at $\lambda = 0$ in "case I+H".
- We assume the tuple (I_0, J_0)=(J_1, K_1) being disabled at $\lambda = 1$ does *not* contain some literal index- and epsilon value tuple $(p(0), 0)$. Then this tuple (I_0, J_0)=(J_1, K_1) being disabled at $\lambda = 1$ is accepted at $\lambda = 0$ in "case I".

The analog observations can be done for any $\lambda + 1$ and λ. This is an additional evidence that the polynomial solver implements the recursion right. It does namely *not* happen that possibilities to select the initially false clauses 'get lost' among the steps of the practical recursion.

One might wonder where the tuples and clauses in the formulas of 4.2.6, 4.2.7, 4.2.8, 4.2.9 come from. In other words, why are the clauses choosable as stated? The answer is that the clause tuples (I_λ, J_λ), (I_λ, K_λ), (H_λ, J_λ), (H_λ, K_λ) with $\lambda \geq 0$ of "case I+H" and "case I" are exactly those tuples which are accepted in RULE 2 (case I+H) and RULE 1 (case I) to disable some (J_λ, K_λ). The initially false clauses $F0_\lambda$, $F1_\lambda$ of "case F0/F1" and F_λ of "case F" are those clauses which disable at least one tuple of "case I+H" and "case I". Finally the one or more tuples which get disabled (i.e. the (J_λ, K_λ) at formula top) are exactly those tuples which can be disabled by RULE 1 or/and RULE 2. It is very important for this proof to know how the tuples (I_λ, J_λ), (I_λ, K_λ), (H_λ, J_λ), (H_λ, K_λ) being accepted by RULE 2 look like, because RULE 2 is the feature of the polynomial solver which does finally implement the recursion and the extension. And the goal of this proof is to show that the polynomial solver does implement the recursion and extension right. Please recall 4.2.1.2. This was the hidden agenda by me when having set up the formulas.

The order of the formulas has been presented 'bottom-up'. This means first it was shown how the basis tuple (J_0, K_0) gets disabled by further tuples which must already have been disabled. In practice, the polynomial solver does the tuple disabling 'top-down' in the opposite order. This means the basis tuple (J_0, K_0) is the last one to be disabled. Through the polynomial solver's `while (true)`-loop in combination with the *Changed* flag check (see 3) it does not matter in which order tuples get disabled. The solver will operate until there is no more possibility to disable at least one tuple. So this proof works without regarding the chronology of the tuple disabling. The bottom up order has been chosen for this proof because also `recursive_F_search()` works some kind of bottom up, as it begins with an empty S_{Finit}.

4.2.5 Formulas Prove the Polynomial Solver does Recursion and Extension

We take for granted the formulas shown in "Formulas" (4.2.3) do correctly and completely describe the working mechanisms of the polynomial solver. This has been proven for each formula, in the "Proof: ..." passages which followed on each formula. Then we can apply the following proof schema: If it can be shown the formulas implement the recursion and the extension (as defined in 4.2.1.2), then we do also know the polynomial solver implements the recursion and the extension. This will now be examined.

recursion) It can be recognized the formulas of 4.2.6 and 4.2.7 and 4.2.8 and 4.2.9 implement the recursion.

In the passage "Idea of the Proof" (4.2.1.2) the recursion has been defined as follows: "If $F0$ was not found, `recursive_F_search()` performs one recursive call of itself. Similarly, if $F1$ was not found, `recursive_F_search()` performs one recursive call of itself. This working mechanism of `recursive_F_search()` is to be called *the recursion*."

Please regard again the formulas presented in 4.2.6, 4.2.7, 4.2.8, 4.2.9. It can be recognized that in each formula, the existence of an initially false $F0_\lambda$ and an initially false $F1_\lambda$ disables the tuple (J_λ, K_λ). This is implemented for $\lambda = 0$ by the formula lines $(\exists F0_0 \mid (\tau(F0_0) = 0 \wedge ...$ and $(\exists F1_0 \mid (\tau(F1_0) = 0 \wedge ....$ For higher λ's the same applies.

It is important to realize that if no such initially false $F0_0$ exists, a disabled tuple (I_0, J_0) or (I_0, K_0) can 'compensate' $F0_0$. 'Compensate' does here mean the disabled tuple has the same effect as the initially false $F0_0$ (in view of disabling the basis (J_0, K_0)). This is implemented by the formula lines marked with the comment `// RECURSION` $\lambda = 0, S_{F0}$. The formula lines $(\exists F0_0 \mid (\tau(F0_0) = 0 \wedge ...$ and the request for a disabled tuple (I_0, J_0) or (I_0, K_0) are linked by an $\vee$, so one of these three requirements is sufficient to contribute to the disabling of (J_0, K_0). Similarly, a disabled tuple (H_0, J_0) or (H_0, K_0) can 'compensate' $F1_0$. Again the related the formula lines are marked with the comment `// RECURSION` $\lambda = 0, S_{F1}$. For higher λ's the same applies, except that it is not required for higher λ's to accept a disabled (I_λ, K_λ) resp. (H_λ, K_λ). For $\lambda \geq 1$ it is sufficient to check for $\pi(I_\lambda, J_\lambda) = 0$ resp. $\pi(H_\lambda, J_\lambda) = 0$ only (the reason therefore will be given later on).

Here are the parallels between `recursive_F_search()` and the formulas describing the working of the polynomial solver:

If `recursive_F_search()` finds no $F0$, it checks if the return value of a recursive sub call to itself is `F_IN_EVERY_CT_LINE`. If the polynomial solver finds no $F0_\lambda$ $(\lambda \geq 0)$, it checks if the specified tuples are disabled. These tuples could have been disabled in a recursive manner, as explained in 4.2.1.3. The same applies for $F1$.

So, figuratively spoken, the analog to `recursive_F_search()`'s return value checks are the polynomial solver's $\pi(clause_1, clause_2) = 0$ checks.

extension) It can be recognized the formulas of 4.2.6 and 4.2.7 and 4.2.8 and 4.2.9 implement the extension.

In the passage "Idea of the Proof" (4.2.1.2) the recursion has been defined as follows: "In the recursive calls, S_F has become S_{F0} respectively S_{F1}. S_{F0} has been built out of S_F by adding a 0. S_{F1} has been built out of S_F by adding a 1. This means $F0$ must be out of $S_{F0} = S_F \cup (p(\lambda), 0)$. Similarly, $F1$ must be out of $S_{F1} = S_F \cup (p(\lambda), 1)$. This working mechanism of `recursive_F_search()` is to be called *the extension*."

Instead of arguing here how the polynomial solver implements this extension, we just regard the formulas, as described at the outset of this document section.

The following mathematical formulas are partially 1-to-1 excerpts from the formulas presented in 4.2.6, 4.2.7, 4.2.8, 4.2.9. The reader might refer to these previously given formulas and compare them to the excerpts and citations given in the following.

The following formula parts were taken from 4.2.6, 4.2.7, 4.2.8, 4.2.9:

1) The parts that define which tuple gets disabled;
2) The parts that define which two initially false clauses $F0_\lambda$ and $F1_\lambda$ (for any $\lambda \geq 0$) must exist to disable this tuple;
3) The parts that tell which disabled tuples (I_λ, J_λ), (H_λ, J_λ), (I_λ, K_λ), (H_λ, K_λ) (for any $\lambda \geq 0$) are accepted as replacement for non-existing $F0_\lambda$ or/and $F1_\lambda$.

The practical recursion has been 'unrolled' once for $0 \leq \lambda \leq 2$ and once for the general case $\lambda \to \lambda + 1$. This means for instance, if there is no initially false $F0_0$ at practical recursion depth $\lambda = 0$, then a disabled (I_0, J_0) or a disabled (I_0, K_0) is accepted as well. It is explicitly shown how these (I_0, J_0) or (I_0, K_0) can get disabled in return at $\lambda = 1$. This $\lambda = 1$ layer has been inserted into the formula at the appropriate position so that the reader has a good overview of the dependencies. Please compare to the dependency visualization shown in the definition of the practical recursion (4.2.1.3).

The overall motivation is to summarize which initially false clauses $F0_\lambda$ and $F1_\lambda$ (for any $\lambda \geq 0$) are required to disable the basis tuple (J_0, K_0). In particular, it is to be shown that the sets the literal index- and epsilon values of these initially false clauses can be chosen from is recursively extended as requested in 4.2.1.2.

Case $\lambda = 0 \to \lambda = 1 \to \lambda = 2$

The formula locations where the extension becomes visible are underlined.

// === taken from 4.2.6, top of formula: ===
$\pi(J_0, K_0) := 0$ if:
// === taken from 4.2.6, "case F0/F1": ===
$(\exists F0_0 \mid (\tau(F0_0) = 0 \wedge (((f0_{0_a}, \epsilon_{f0_{0\,a}}), (f0_{0_b}, \epsilon_{f0_{0\,b}})) \in$
$\{(j_{0_1}, \epsilon_{j_{0\,1}}), (j_{0_2}, \epsilon_{j_{0\,2}}), (j_{0_3}, \epsilon_{j_{0\,3}}), (k_{0_1}, \epsilon_{k_{0\,1}}), (k_{0_2}, \epsilon_{k_{0\,2}}), (k_{0_3}, \epsilon_{k_{0\,3}})\} \wedge (f0_{0_c}, \epsilon_{f0_{0\,c}}) = \underline{(p(0), 0)})))$
// === taken from 4.2.6, "case I+H": ===
If no such initially false $F0_0$ exists, $\pi(I_0, J_0) = 0 \vee \pi(I_0, K_0) = 0$ is accepted.
  // === taken from 4.2.7, top of formula: ===
  Either $\pi(I_0, J_0) := 0$ or $\pi(I_0, K_0) := 0$ if:
  // === taken from 4.2.7, "case F0/F1": ===
  $(\exists F0_1 \mid (\tau(F0_1) = 0 \wedge (((f0_{1_a}, \epsilon_{f0_{1\,a}}), (f0_{1_b}, \epsilon_{f0_{1\,b}})) \in$
  $\{(j_{0_1}, \epsilon_{j_{0\,1}}), (j_{0_2}, \epsilon_{j_{0\,2}}), (j_{0_3}, \epsilon_{j_{0\,3}}), (k_{0_1}, \epsilon_{k_{0\,1}}), (k_{0_2}, \epsilon_{k_{0\,2}}), (k_{0_3}, \epsilon_{k_{0\,3}}), (p(0), 0)\} \wedge (f0_{1_c}, \epsilon_{f0_{1\,c}}) = \underline{(p(1), 0)})))$
  // === taken from 4.2.7, "case I+H": ===
  If no such initially false $F0_1$ exists, $\pi(I_1, J_1) = 0$ is accepted.
    // === taken from 4.2.9, top of formula: ===
    $\pi(I_1, J_1) := 0$ if:
    // === taken from 4.2.9, "case F0/F1": ===
    $(\exists F0_2 \mid (\tau(F0_2) = 0 \wedge (((f0_{2_a}, \epsilon_{f0_{2\,a}}), (f0_{2_b}, \epsilon_{f0_{2\,b}})) \in$
    $\{(j_{0_1}, \epsilon_{j_{0\,1}}), (j_{0_2}, \epsilon_{j_{0\,2}}), (j_{0_3}, \epsilon_{j_{0\,3}}), (k_{0_1}, \epsilon_{k_{0\,1}}), (k_{0_2}, \epsilon_{k_{0\,2}}), (k_{0_3}, \epsilon_{k_{0\,3}}), ..., (p(2-1), \epsilon_{p(2-1)})\} \wedge$
    $(f0_{2_c}, \epsilon_{f0_{2\,c}}) = \underline{(p(2), 0)})))$
    // === taken from 4.2.9, "case I+H": ===
    If no such initially false $F0_2$ exists, $\pi(I_2, J_2) = 0$ is accepted.
    $\wedge$
    $(\exists F1_2 \mid (\tau(F1_2) = 0 \wedge (((f1_{2_d}, \epsilon_{f1_{2\,d}}), (f1_{2_e}, \epsilon_{f1_{2\,e}})) \in$
    $\{(j_{0_1}, \epsilon_{j_{0\,1}}), (j_{0_2}, \epsilon_{j_{0\,2}}), (j_{0_3}, \epsilon_{j_{0\,3}}), (k_{0_1}, \epsilon_{k_{0\,1}}), (k_{0_2}, \epsilon_{k_{0\,2}}), (k_{0_3}, \epsilon_{k_{0\,3}}), ..., (p(2-1), \epsilon_{p(2-1)})\} \wedge$
    $(f1_{2_f}, \epsilon_{f1_{2\,f}}) = \underline{(p(2), 1)})))$
    If no such initially false $F1_2$ exists, $\pi(H_2, J_2) = 0$ is accepted.

$\wedge$
$(\exists F1_1 \mid (\tau(F1_1) = 0 \wedge (((f1_{1_d}, \epsilon_{f1_{1_d}})(f1_{1_e}, \epsilon_{f1_{1_e}})) \in$
$\{(j_{0_1}, \epsilon_{j_{0_1}}), (j_{0_2}, \epsilon_{j_{0_2}}), (j_{0_3}, \epsilon_{j_{0_3}}), (k_{0_1}, \epsilon_{k_{0_1}}), (k_{0_2}, \epsilon_{k_{0_2}}), (k_{0_3}, \epsilon_{k_{0_3}}), (p(0), 0)\} \wedge (f1_{1_f}, \epsilon_{f1_{1_f}}) = \underline{(p(1), 1)})))$
`If no such initially false` $F1_1$ `exists,` $\pi(H_1, J_1) = 0$ `is accepted.`
$\pi(H_1, J_1) := 0$ `if:`
$(\exists F0_2 \mid (\tau(F0_2) = 0 \wedge (((f0_{2_a}, \epsilon_{f0_{2_a}}), (f0_{2_b}, \epsilon_{f0_{2_b}})) \in$
$\{(j_{0_1}, \epsilon_{j_{0_1}}), (j_{0_2}, \epsilon_{j_{0_2}}), (j_{0_3}, \epsilon_{j_{0_3}}), (k_{0_1}, \epsilon_{k_{0_1}}), (k_{0_2}, \epsilon_{k_{0_2}}), (k_{0_3}, \epsilon_{k_{0_3}}), ..., (p(2-1), \epsilon_{p(2-1)})\} \wedge$
$(f0_{2_c}, \epsilon_{f0_{2_c}}) = \underline{(p(2), 0)})))$
`If no such initially false` $F0_2$ `exists,` $\pi(I_2, J_2) = 0$ `is accepted.`
$\wedge$
$(\exists F1_2 \mid (\tau(F1_2) = 0 \wedge (((f1_{2_d}, \epsilon_{f1_{2_d}}), (f1_{2_e}, \epsilon_{f1_{2_e}})) \in$
$\{(j_{0_1}, \epsilon_{j_{0_1}}), (j_{0_2}, \epsilon_{j_{0_2}}), (j_{0_3}, \epsilon_{j_{0_3}}), (k_{0_1}, \epsilon_{k_{0_1}}), (k_{0_2}, \epsilon_{k_{0_2}}), (k_{0_3}, \epsilon_{k_{0_3}}), ..., (p(2-1), \epsilon_{p(2-1)})\} \wedge$
$(f1_{2_f}, \epsilon_{f1_{2_f}}) = \underline{(p(2), 1)})))$
`If no such initially false` $F1_2$ `exists,` $\pi(H_2, J_2) = 0$ `is accepted.`
$\wedge$
`// === taken from 4.2.6, "case F0/F1": ===`
$(\exists F1_0 \mid (\tau(F1_0) = 0 \wedge (((f1_{0_d}, \epsilon_{f1_{0_d}}), (f1_{0_e}, \epsilon_{f1_{0_e}})) \in$
$\{(j_{0_1}, \epsilon_{j_{0_1}}), (j_{0_2}, \epsilon_{j_{0_2}}), (j_{0_3}, \epsilon_{j_{0_3}}), (k_{0_1}, \epsilon_{k_{0_1}}), (k_{0_2}, \epsilon_{k_{0_2}}), (k_{0_3}, \epsilon_{k_{0_3}})\} \wedge (f1_{0_f}, \epsilon_{f1_{0_f}}) = \underline{(p(0), 1)})))$
`// === taken from 4.2.6, "case I+H": ===`
`If no such initially false` $F1_0$ `exists,` $\pi(H_0, J_0) = 0 \vee \pi(H_0, K_0) = 0$ `is accepted.`
`// === taken from 4.2.8, top of formula: ===`
`Either` $\pi(H_0, J_0) := 0$ `or` $\pi(H_0, K_0) := 0$ `if:`
`// === taken from 4.2.8, "case F0/F1": ===`
$(\exists F0_1 \mid (\tau(F0_1) = 0 \wedge (((f0_{1_a}, \epsilon_{f0_{1_a}}), (f0_{1_b}, \epsilon_{f0_{1_b}})) \in$
$\{(j_{0_1}, \epsilon_{j_{0_1}}), (j_{0_2}, \epsilon_{j_{0_2}}), (j_{0_3}, \epsilon_{j_{0_3}}), (k_{0_1}, \epsilon_{k_{0_1}}), (k_{0_2}, \epsilon_{k_{0_2}}), (k_{0_3}, \epsilon_{k_{0_3}}), (p(0), 1)\} \wedge (f0_{1_c}, \epsilon_{f0_{1_c}}) = \underline{(p(1), 0)})))$
`// === taken from 4.2.8, "case I+H": ===`
`If no such initially false` $F0_1$ `exists,` $\pi(I_1, J_1) = 0$ `is accepted.`
$\pi(I_1, J_1) := 0$ `if:`
$(\exists F0_2 \mid (\tau(F0_2) = 0 \wedge (((f0_{2_a}, \epsilon_{f0_{2_a}}), (f0_{2_b}, \epsilon_{f0_{2_b}})) \in$
$\{(j_{0_1}, \epsilon_{j_{0_1}}), (j_{0_2}, \epsilon_{j_{0_2}}), (j_{0_3}, \epsilon_{j_{0_3}}), (k_{0_1}, \epsilon_{k_{0_1}}), (k_{0_2}, \epsilon_{k_{0_2}}), (k_{0_3}, \epsilon_{k_{0_3}}), ..., (p(2-1), \epsilon_{p(2-1)})\} \wedge$
$(f0_{2_c}, \epsilon_{f0_{2_c}}) = \underline{(p(2), 0)})))$
`If no such initially false` $F0_2$ `exists,` $\pi(I_2, J_2) = 0$ `is accepted.`
$\wedge$
$(\exists F1_2 \mid (\tau(F1_2) = 0 \wedge (((f1_{2_d}, \epsilon_{f1_{2_d}}), (f1_{2_e}, \epsilon_{f1_{2_e}})) \in$
$\{(j_{0_1}, \epsilon_{j_{0_1}}), (j_{0_2}, \epsilon_{j_{0_2}}), (j_{0_3}, \epsilon_{j_{0_3}}), (k_{0_1}, \epsilon_{k_{0_1}}), (k_{0_2}, \epsilon_{k_{0_2}}), (k_{0_3}, \epsilon_{k_{0_3}}), ..., (p(2-1), \epsilon_{p(2-1)})\} \wedge$
$(f1_{2_f}, \epsilon_{f1_{2_f}}) = \underline{(p(2), 1)})))$
`If no such initially false` $F1_2$ `exists,` $\pi(H_2, J_2) = 0$ `is accepted.`
$\wedge$
$(\exists F1_1 \mid (\tau(F1_1) = 0 \wedge (((f1_{1_d}, \epsilon_{f1_{1_d}}), (f1_{1_e}, \epsilon_{f1_{1_e}})) \in$
$\{(j_{0_1}, \epsilon_{j_{0_1}}), (j_{0_2}, \epsilon_{j_{0_2}}), (j_{0_3}, \epsilon_{j_{0_3}}), (k_{0_1}, \epsilon_{k_{0_1}}), (k_{0_2}, \epsilon_{k_{0_2}}), (k_{0_3}, \epsilon_{k_{0_3}}), (p(0), 1)\} \wedge (f1_{1_f}, \epsilon_{f1_{1_f}}) = \underline{(p(1), 1)})))$
`If no such initially false` $F1_1$ `exists,` $\pi(H_1, J_1) = 0$ `is accepted.`
$\pi(H_1, J_1) := 0$ `if:`
$(\exists F0_2 \mid (\tau(F0_2) = 0 \wedge (((f0_{2_a}, \epsilon_{f0_{2_a}}), (f0_{2_b}, \epsilon_{f0_{2_b}})) \in$
$\{(j_{0_1}, \epsilon_{j_{0_1}}), (j_{0_2}, \epsilon_{j_{0_2}}), (j_{0_3}, \epsilon_{j_{0_3}}), (k_{0_1}, \epsilon_{k_{0_1}}), (k_{0_2}, \epsilon_{k_{0_2}}), (k_{0_3}, \epsilon_{k_{0_3}}), ..., (p(2-1), \epsilon_{p(2-1)})\} \wedge$
$(f0_{2_c}, \epsilon_{f0_{2_c}}) = \underline{(p(2), 0)})))$
`If no such initially false` $F0_2$ `exists,` $\pi(I_2, J_2) = 0$ `is accepted.`
$\wedge$
$(\exists F1_2 \mid (\tau(F1_2) = 0 \wedge (((f1_{2_d}, \epsilon_{f1_{2_d}}), (f1_{2_e}, \epsilon_{f1_{2_e}})) \in$
$\{(j_{0_1}, \epsilon_{j_{0_1}}), (j_{0_2}, \epsilon_{j_{0_2}}), (j_{0_3}, \epsilon_{j_{0_3}}), (k_{0_1}, \epsilon_{k_{0_1}}), (k_{0_2}, \epsilon_{k_{0_2}}), (k_{0_3}, \epsilon_{k_{0_3}}), ..., (p(2-1), \epsilon_{p(2-1)})\} \wedge$
$(f1_{2_f}, \epsilon_{f1_{2_f}}) = \underline{(p(2), 1)})))$
`If no such initially false` $F1_2$ `exists,` $\pi(H_2, J_2) = 0$ `is accepted.`

From the just shown formula, the following information can be derived:

- If no initially false $F0_0$ exists, an initially false $F0_1$ and an initially false $F1_1$ can take over its task to disable (J_0, K_0).

  The sets the literal index values of $F0_1$ can be chosen from is equal to the set the literal index values of $F0_0$ can be chosen from, extended by $p(1)$. Equally, the corresponding epsilon value set was extended by a 0.

  The sets the literal index values of $F1_1$ can be chosen from is equal to the set the literal index values of $F0_0$ can be chosen from, extended by $p(1)$. Equally, the corresponding epsilon value set was extended by a 1.

  If no initially false $F1_0$ exists, an initially false $F0_1$ and an an initially false $F1_1$ can take over its task. The sets the literal index- and epsilon values of $F0_1$ and $F1_1$ can be chosen from have again been extended as required.

  So we see one initially false $F0_0$ can be compensated by the two initially false clauses $F0_1$ and $F1_1$. Similarly, one initially false $F1_0$ can be compensated by another two initially false clauses $F0_1$ and $F1_1$. This fulfills the requirements of the extension.

- The analog is true for $F0_1$, which can be compensated by $F0_2$ and $F1_2$. Furthermore, $F1_1$ can be compensated by another $F0_2$ and $F1_2$.

  Please notice that the formula excerpts which contain indices representing $\lambda = 2$ have been taken from the formula for $\lambda \geq 2$ presented in 4.2.9. The "λ" symbols have here been replaced by the number "2" for better understandability.

- In the original formulas (4.2.6, 4.2.7, 4.2.8, 4.2.9), the locations where the just described extension is done are marked with the comments `// EXTENSION` $\lambda = ..., S_{F0}$ (if the extension creates the set S_{F0}) resp. `// EXTENSION` $\lambda = ..., S_{F1}$ (if the extension creates the set S_{F1}).

Case $\lambda \to \lambda + 1$

It was just shown which initially false clauses must exist to disable the basis (J_0, K_0) for $0 \leq \lambda \leq 2$. Now the general case for $\lambda \geq 2$ is regarded. This general case can be examined just as in the prior passage, except that all excerpts were taken from 4.2.9 only. In the nested, recursive parts all "λ" symbols from 4.2.9 have been replaced by "$\lambda + 1$". Like this, it becomes visible how the extension is done for any transition $\lambda \to \lambda + 1$.

The formula locations where the extension becomes visible are underlined.

$\pi(J_\lambda, K_\lambda) := 0$ `if:`
$(\exists F0_\lambda \mid (\tau(F0_\lambda) = 0 \wedge (((f0_{\lambda_a}, \epsilon_{f0_{\lambda\,a}}), (f0_{\lambda_b}, \epsilon_{f0_{\lambda\,b}})) \in$
$\{(j_{0\,1}, \epsilon_{j_{0\,1}}), (j_{0\,2}, \epsilon_{j_{0\,2}}), (j_{0\,3}, \epsilon_{j_{0\,3}}), (k_{0\,1}, \epsilon_{k_{0\,1}}), (k_{0\,2}, \epsilon_{k_{0\,2}}), (k_{0\,3}, \epsilon_{k_{0\,3}}), ..., (p(\lambda - 1), \epsilon_{p(\lambda-1)})\} \wedge$
$(f0_{\lambda_c}, \epsilon_{f0_{\lambda\,c}}) = \underline{(p(\lambda), 0)})))$
`If no such initially false` $F0_\lambda$ `exists,` $\pi(I_\lambda, J_\lambda) = 0$ `is accepted.`
$\pi(I_\lambda, J_\lambda) := 0$ `if:`
$(\exists F0_{\lambda+1} \mid (\tau(F0_{\lambda+1}) = 0 \wedge (((f0_{\lambda+1_a}, \epsilon_{f0_{\lambda+1\,a}}), (f0_{\lambda+1_b}, \epsilon_{f0_{\lambda+1\,b}})) \in$
$\{(j_{0\,1}, \epsilon_{j_{0\,1}}), (j_{0\,2}, \epsilon_{j_{0\,2}}), (j_{0\,3}, \epsilon_{j_{0\,3}}), (k_{0\,1}, \epsilon_{k_{0\,1}}), (k_{0\,2}, \epsilon_{k_{0\,2}}), (k_{0\,3}, \epsilon_{k_{0\,3}}), ..., (p(\lambda + 1 - 1), \epsilon_{p(\lambda+1-1)})\} \wedge$
$(f0_{\lambda+1_c}, \epsilon_{f0_{\lambda+1\,c}}) = \underline{(p(\lambda + 1), 0)})))$
`If no such initially false` $F0_{\lambda+1}$ `exists,` $\pi(I_{\lambda+1}, J_{\lambda+1}) = 0$ `is accepted.`
$\wedge$
$(\exists F1_{\lambda+1} \mid (\tau(F1_{\lambda+1}) = 0 \wedge (((f1_{\lambda+1_d}, \epsilon_{f1_{\lambda+1\,d}}), (f1_{\lambda+1_e}, \epsilon_{f1_{\lambda+1\,e}})) \in$
$\{(j_{0\,1}, \epsilon_{j_{0\,1}}), (j_{0\,2}, \epsilon_{j_{0\,2}}), (j_{0\,3}, \epsilon_{j_{0\,3}}), (k_{0\,1}, \epsilon_{k_{0\,1}}), (k_{0\,2}, \epsilon_{k_{0\,2}}), (k_{0\,3}, \epsilon_{k_{0\,3}}), ..., (p(\lambda + 1 - 1), \epsilon_{p(\lambda+1-1)})\} \wedge$
$(f1_{\lambda+1\,f}, \epsilon_{f1_{\lambda+1\,f}}) = \underline{(p(\lambda + 1), 1)})))$
`If no such initially false` $F1_{\lambda+1}$ `exists,` $\pi(H_{\lambda+1}, J_{\lambda+1}) = 0$ `is accepted.`
$\wedge$
$(\exists F1_\lambda \mid (\tau(F1_\lambda) = 0 \wedge (((f1_{\lambda_d}, \epsilon_{f1_{\lambda\,d}}), (f1_{\lambda_e}, \epsilon_{f1_{\lambda\,e}})) \in$
$\{(j_{0\,1}, \epsilon_{j_{0\,1}}), (j_{0\,2}, \epsilon_{j_{0\,2}}), (j_{0\,3}, \epsilon_{j_{0\,3}}), (k_{0\,1}, \epsilon_{k_{0\,1}}), (k_{0\,2}, \epsilon_{k_{0\,2}}), (k_{0\,3}, \epsilon_{k_{0\,3}}), ..., (p(\lambda - 1), \epsilon_{p(\lambda-1)})\} \wedge$
$(f1_{\lambda\,f}, \epsilon_{f1_{\lambda\,f}}) = \underline{(p(\lambda), 1)})))$
`If no such initially false` $F1_\lambda$ `exists,` $\pi(H_\lambda, J_\lambda) = 0$ `is accepted.`

$\pi(H_\lambda, J_\lambda) := 0$ `if:`
$(\exists F0_{\lambda+1} \mid (\tau(F0_{\lambda+1}) = 0 \wedge (((f0_{\lambda+1_a}, \epsilon_{f0_{\lambda+1_a}}), (f0_{\lambda+1_b}, \epsilon_{f0_{\lambda+1_b}})) \in$
$\{(j_{0_1}, \epsilon_{j_{0_1}}), (j_{0_2}, \epsilon_{j_{0_2}}), (j_{0_3}, \epsilon_{j_{0_3}}), (k_{0_1}, \epsilon_{k_{0_1}}), (k_{0_2}, \epsilon_{k_{0_2}}), (k_{0_3}, \epsilon_{k_{0_3}}), ..., (p(\lambda+1-1), \epsilon_{p(\lambda+1-1)})\} \wedge$
$(f0_{\lambda+1_c}, \epsilon_{f0_{\lambda+1_c}}) = \underline{(p(\lambda+1), 0)})))$
`If no such initially false` $F0_{\lambda+1}$ `exists,` $\pi(I_{\lambda+1}, J_{\lambda+1}) = 0$ `is accepted.`
$\wedge$
$(\exists F1_{\lambda+1} \mid (\tau(F1_{\lambda+1}) = 0 \wedge (((f1_{\lambda+1_d}, \epsilon_{f1_{\lambda+1_d}}), (f1_{\lambda+1_e}, \epsilon_{f1_{\lambda+1_e}})) \in$
$\{(j_{0_1}, \epsilon_{j_{0_1}}), (j_{0_2}, \epsilon_{j_{0_2}}), (j_{0_3}, \epsilon_{j_{0_3}}), (k_{0_1}, \epsilon_{k_{0_1}}), (k_{0_2}, \epsilon_{k_{0_2}}), (k_{0_3}, \epsilon_{k_{0_3}}), ..., (p(\lambda+1-1), \epsilon_{p(\lambda+1-1)})\} \wedge$
$(f1_{\lambda+1_f}, \epsilon_{f1_{\lambda+1_f}}) = \underline{(p(\lambda+1), 1)})))$
`If no such initially false` $F1_{\lambda+1}$ `exists,` $\pi(H_{\lambda+1}, J_{\lambda+1}) = 0$ `is accepted.`

It is easy to see that any $F0_\lambda$ can be replaced by two initially false clauses $F0_{\lambda+1}$ and $F1_{\lambda+1}$. These two clauses each have their set the literal index values can be chosen from extended by one further index $p(\lambda+1)$. Also the two clauses' epsilon value set has been extended by 0 (for $F0_{\lambda+1}$) resp. 1 (for $F1_{\lambda+1}$). The same applies to $F1_\lambda$. This is exactly what we want, because this kind of extension fulfills the requirements of the extension as defined in 4.2.1.2.

Conclusion

After having examined all λ cases, it is now clear the polynomial solver implements the extension as desired, i.e. as requested in 4.2.1.2. Formally, the presented cases for $\lambda = 0$, $\lambda = 1$, $\lambda = 2$ and $\lambda \to \lambda + 1$ can be seen as parts of an induction proof. $\lambda = 0$, $\lambda = 1$, $\lambda = 2$ are the base case of the induction proof. $\lambda \to \lambda + 1$ represents the inductive step of the induction proof. This inductive step is valid for any $\lambda \geq 2$. Thus, for all base cases and any inductive step it was shown the polynomial solver implements the extension as desired.

4.2.6 Final Conclusion and Summary

As basic premise of this proof it is given that there is at least one initially false clause in each CT line. The helper function `recursive_F_search()` has been introduced and it was explained why this function reliably detects any initially false clause in the CT lines. It has been observed that important key components of `recursive_F_search()` are the abilities to do "the recursion" and "the extension". To check if the polynomial solver does this work analog to `recursive_F_search()`, the polynomial solver's tuple disabling mechanisms have been described by "formulas". It was shown the formulas do describe the recursion and the extension. From this it can finally be concluded that the polynomial solver does work like `recursive_F_search()` and it does thus detect any initially false clause in each CT line. Therewith the correctness of the polynomial solver has been proven. □

4.2.7 Further Notes

In the $\lambda \geq 2$ case the proof suggests it is not required to involve (I_λ, K_λ) or (H_λ, K_λ). This has been tested by me using computer-aided verification. The result is that the K_λ tuples were really not required in all tests done. But this does not apply for the $\lambda = 1$ case, what can be derived from the proof. There it is shown that one I_1 or H_1 literal will be equal to one literal in *either* J_0 *or* K_0 (J_0 is also called $K1_1$ and K_0 is also called $K2_1$ in 4.2.4). It is not known in advance if J_0 or K_0 will be required. This depends on the SAT CNF to solve. I verified also this statement (that (I_λ, K_λ) and (H_λ, K_λ) is mandatory) using a test program, with the observation the statement seems to be correct. An extensive test run series with missing (I_λ, K_λ) and (H_λ, K_λ) checks occasionally lead to wrong results of the polynomial solver.

The proof was presented assuming there is one initially false clause in each CT line. If the 3-SAT CNF to solve should contain clauses in a way there would be two or more initially false clauses in each CT line, then this does not change the way to prove the correctness. The proof shows any basis tuple (J_0, K_0) gets disabled if there's one initially false clause in each CT line. If there are more initially false clauses,

all the better. Because the basis tuple (J_0, K_0) gets disabled *as soon as* "enough" initially false clauses have been found, more initially false clauses will likely even lead to a quicker disabling of the basis tuple (J_0, K_0).

Please notice that RULE 1 can *not* be replaced by the INITIALIZATION or RULE 2 [9]. This can easily be understood by regarding a suitable example: We assume $n = 6$ and $J = (0x_2 \vee 0x_3 \vee 0x_4)$ and $K = (0x_2 \vee 0x_4 \vee 1x_5)$. J and K are assumed to be *no* initially false clauses. If there is some initially false clause $I = (0x_2 \vee 0x_3 \vee 1x_5) \mid \tau(I) = 0$, then *only* RULE 1 sets $\pi(J, K) := 0$. The reason is that I is initially false and contained within (J, K). The INITIALIZATION won't disable (J, K) because J or K had to be initially false. Please see 3. Also RULE 2 won't disable (J, K) because RULE 2 would need *two* initially false clauses I and H, each with one literal index not in J or K. Hence RULE 1 is irreplaceable.

It might (often) be the case that not all clauses of the SAT CNF are required to disable a basis tuple (J_0, K_0). Mostly a sub set of all SAT CNF clauses is sufficient. But the set of finally effective SAT CNF clauses might differ from basis tuple to basis tuple.

4.2.8 Examples

In this "Why Unsatisfiable Detection is Reliable" proof the formulas 4.2.6, 4.2.7, 4.2.8, 4.2.9 were presented. They describe which initially false clauses or disabled tuples the polynomial solver accepts at some practical recursion depth λ to disable some tuple(s) (J_λ, K_λ).

Four scenarios to disable (J_λ, K_λ) were distinguished:

- "Case I" accepts one or two disabled tuples $((I_\lambda, J_\lambda) \vee (I_\lambda, K_\lambda))$ with I_λ being contained within (J_λ, K_λ).
- "Case I+H" accepts two, three or four disabled tuples $((I_\lambda, J_\lambda) \vee (I_\lambda, K_\lambda)) \wedge ((H_\lambda, J_\lambda) \vee (H_\lambda, K_\lambda))$ with I_λ and H_λ being contained within (J_λ, K_λ).
- "Case F" accepts one initially false clause F_λ which disables one tuple (I_λ, J_λ) of "case I". Then "case I" is applied to disable (J_λ, K_λ).
- "Case F0/F1" accepts two initially false clauses $F0_\lambda$ and $F1_\lambda$ which disable two tuples (I_λ, J_λ) and (H_λ, J_λ) of "case I+H". Then "case I+H" is applied to disable (J_λ, K_λ).

To make the reader understand better how these cases work in practice, two example usages will be shown.

In each of the both examples a basis tuple (J_0, K_0) is given. Additionally some initially false clauses are defined. It will be shown that first those initially false clauses disable tuples in some practical recursion depth λ. Here the actual 'top-down' disabling order is displayed, as it would be applied in practice. Please recall the paragraph at end of 4.2.4. This means the highest λ is shown first. After the initially false clauses disabled tuples, it will be shown for each λ how the already disabled tuples disable further tuple(s) (J_λ, K_λ). The applied case ("case I", "case I+H" etc.) is always stated. Additionally it is always stated which rule of the polynomial solver (INITIALIZATION rule, RULE 1 or RULE 2) implements the tuple disabling. It will be reasoned why the presented rule can be applied. In the first example, excerpts from the original pseudo-code definition (see 3) are cited. These excerpts are not repeated in the second example to save page space, but they could be placed there in the exact same manner as in the first example. At the very end of each example, the basis tuple will have gotten disabled: $\pi(J_0, K_0) := 0$, as desired.

The literal index- and epsilon values of all involved clauses are given in the classical mathematical notation or/and in tuple notation. This means any 3-SAT clause $C \in PC$ can be noted like this:

$$C = (\epsilon_{c1} x_{c1} \vee \epsilon_{c2} x_{c2} \vee \epsilon_{c3} x_{c3}) = ((c_1, \epsilon_{c1}), (c_2, \epsilon_{c2}), (c_3, \epsilon_{c3}))$$

[9] This is addressed here as I got reader feedback about this topic.

The tuple notation is used as parallel to the formulas 4.2.6, 4.2.7, 4.2.8, 4.2.9, where this notation is used as well. Like this, the reader can compare the generalized tuples in the formulas with examples how the content of these tuples can look in practice.

As already mentioned, it is shown consistently in the following two examples how disabled tuples in some practical recursion depth λ disable tuple(s) (J_λ, K_λ) subsequently used in some practical recursion depth $\lambda - 1$. In reality, the polynomial solver might skip practical recursion layers. This would be the case e.g. in the first example, $\lambda = 1$. Instead of "passing on" the disabled tuple to $\lambda = 0$ (as described in the example), the polynomial solver might instantly use (J_2, K_2) as (I_0, J_0) (again see example). The polynomial solver can do this because it does not have to use $p(Lambda)$ (see 4.2.2.2) in a descendant order as presented in the past "Why Unsatisfiable Detection is Reliable" proof. However, nevertheless the polynomial solver would in practice also work if it would be extended by code to use $p(Lambda)$ in the strictly descending order as shown. The proof contains this and some additional restrictions to keep it as simple as possible and thus still understandable for me and the reader. I strongly assume this does not impact the correctness of the proof.

Please notice that in the pseudo-code excerpts, e.g. "`foreach` $J_2 \in PC$" shall just mean the `foreach` loop does currently point to the clause J_2.

Some clauses have several clause variables assigned, like "$\texttt{I}_0\texttt{=((1,0),(2,0),(3,0))=}J_1$". This is the case because the same clauses, i.e. the same three literal index- and epsilon value tuples, are named differently in each practical recursion depth. For instance, J_1 in practical recursion depth $\lambda = 1$ is the very same clause as J_0 in practical recursion depth $\lambda = 0$. This renaming is just done to demonstrate where a clause is used within RULE 1 and RULE 2. For instance, the tuple being disabled by RULE 1 or RULE 2 is to be called (J, K). This is the naming convention used in the definitions of RULE 1 and RULE 2, see 3. In the first example at $\lambda = 1$, the tuple which gets disabled is (J_1, K_1). In the next examined practical recursion layer $\lambda = 0$, this (J_1, K_1) is involved in RULE 1. (J_1, K_1) is used within RULE 1 at a pseudo-code location where the definition of RULE 1 would call the tuple (I, J). For this reason (J_1, K_1) from practical recursion depth $\lambda = 1$ is named (I_0, J_0) at practical recursion depth $\lambda = 0$. Please notice the polynomial solver differentiates clauses only by their content (i.e. literal index- and epsilon values), so there can *not* be two instances of the same clause which are treated *differently* (in any way).

4.2.8.1 First Example

The following literal index range n, the following basis tuple (J_0, K_0) and the following initially false clauses are given:

$$n = 6, p = \{4, 5, 6\}$$
$$J_0 = (0x_1 \vee 0x_2 \vee 0x_3) = ((1,0),(2,0),(3,0))$$
$$K_0 = (0x_1 \vee 0x_2 \vee 0x_3) = ((1,0),(2,0),(3,0))$$
$$F1 = (0x_1 \vee 0x_3 \vee 0x_6) = ((1,0),(3,0),(6,0)) \mid \tau(F1) = 0$$
$$F2 = (0x_2 \vee 0x_3 \vee 1x_6) = ((2,0),(3,0),(6,1)) \mid \tau(F2) = 0$$

$p()$ contains $\{4, 5, 6\}$ because these literal indices appear neither in the basis J_0 nor in the basis K_0.

The solver does the following steps to disable the basis tuple:

INITIALIZATION:
`case F0: the INITIALIZATION has disabled (among others) a tuple` $(F1, J1)$ `with` [10]
`F1=((1,0),(3,0),(6,0)),`
`J1=((1,0),(2,0),(3,0))`
`case F1: the INITIALIZATION has disabled (among others) a tuple` $(F2, J2)$ `with`
`F2=((2,0),(3,0),(6,1)),`
`J2=((1,0),(2,0),(3,0))`

Because it applies $\tau(F1) = 0$ and $\tau(F2) = 0$, any tuple holding $F1$ or $F2$ will be initialized to a disabled tuple.

The original pseudo-code definition of the responsible INITIALIZATION rule with clause variables of this consideration inserted reads as follows:

```
foreach F1 ∈ PC
  foreach J1 ∈ PC
    if ((τ(F1) = 1) ∧ (τ(J1) = 1) ∧ (F1 ≡ J1))
      π(F1, J1) := 1
    if ((τ(F1) = 0) ∨ (τ(J1) = 0) ∨ (F1 ≢ J1)) // this applies
      π(F1, J1) := 0 // this applies

foreach F2 ∈ PC
  foreach J2 ∈ PC
    if ((τ(F2) = 1) ∧ (τ(J2) = 1) ∧ (F2 ≡ J2))
      π(F2, J2) := 1
    if ((τ(F2) = 0) ∨ (τ(J2) = 0) ∨ (F2 ≢ J2)) // this applies
      π(F2, J2) := 0 // this applies
```

$\lambda = 2$:
`case I+H:`
I_2`=((1,0),(3,0),(6,0))=`$F1$`,`
J_2`=((1,0),(2,0),(3,0))=`$J1$
`and`
H_2`=((2,0),(3,0),(6,1))=`$F2$`,`
J_2`=((1,0),(2,0),(3,0))=`$J2$
`disable` [11]
J_2`=((1,0),(2,0),(3,0))=`$J1$`=`$J2$`,`
K_2`=((1,0),(2,0),(3,0))`

At the beginning it applies $\pi(J_2, K_2) = 1$, because the tuple (J_2, K_2) has not been initialized to disabled.

I_2 and H_2 are contained within (J_2, K_2) because all literal index- and epsilon tuples of I_2 and H_2 appear in J_2 or K_2 or both. The only exception is the tuple $(6, 0)$ in I_2 and $(6, 1)$ in H_2. This is the conflict literal corresponding to $p = 6$ which is 'dropped' in (J_2, K_2).

It applies $\pi(F1, J1) = 0$ (from INITIALIZATION), which is identical to $\pi(I_2, J_2) = 0$. It applies $\pi(F2, J2) = 0$ (from INITIALIZATION), which is identical to $\pi(H_2, J_2) = 0$.

[10] In both examples 4.2.8.1 and 4.2.8.2 only those disabled tuples are shown which are of relevance in respect of disabling the basis tuple (J_0, K_0).

[11] This kind of notation is here used as short form for (($\pi(I_2, J_2) = 0$ and $\pi(H_2, J_2) = 0$) leads to $\pi(J_2, K_2) := 0$). The same applies to any equal notation in the space of this examples section. The clause names at left are the clause names used in the current practical recursion depth λ. The clause names at right are the clause names used in the previous practical recursion depth $\lambda + 1$ resp. in the initialization.

This fulfills the three `if ()` conditions of RULE 2, so that RULE 2 will set $\pi(J_2, K_2) := 0$.

The original pseudo-code definition of RULE 2 with clause variables of this consideration inserted reads as follows:

```
foreach J2 ∈ PC
  foreach K2 ∈ PC
    if (π(J2, K2) = 1)
      foreach I2 ∈ PC
        foreach H2 ∈ PC
          if (I2 and H2 are contained within (J2, K2))
            if ((π(I2, J2) = 0 ∨ π(I2, K2) = 0) ∧ (π(H2, J2) = 0 ∨ π(H2, K2) = 0))
              π(J2, K2) := 0
              Changed := true
```

$\lambda = 1$:
`case I:`
$\mathtt{I}_1$`=((1,0),(2,0),(3,0))=`J_2`,`
$\mathtt{J}_1$`=((1,0),(2,0),(3,0))=`K_2
`disables` [12]
$\mathtt{J}_1$`=((1,0),(2,0),(3,0)),`
$\mathtt{K}_1$`=((1,0),(2,0),(3,0))`

At the beginning it applies $\pi(J_1, K_1) = 1$, because the tuple (J_1, K_1) has not been initialized to disabled.

I_1 is contained within (J_1, K_1) because all literal index- and epsilon tuples of I_1 appear in J_1 or K_1 or both.

It applies $\pi(J_2, K_2) = 0$ (from $\lambda = 2$), which is identical to $\pi(I_1, J_1) = 0$.

This fulfills the three `if ()` conditions of RULE 1, so that RULE 1 will set $\pi(J_1, K_1) := 0$.

Figuratively spoken, the disabled tuple is passed on (or "forwarded") unchanged to the next lower $\lambda = 0$. The main work in this case is just to decrease λ. This is required as the formulas (4.2.3) describe a recursion where λ is increased (resp. decreased in 'top-down' disabling order) gradually.

The original pseudo-code definition of RULE 1 with clause variables of this consideration inserted reads as follows:

```
foreach J1 ∈ PC
  foreach K1 ∈ PC
    if (π(J1, K1) = 1)
      foreach I1 ∈ PC
        if (I1 is contained within (J1, K1))
          if (π(I1, J1) = 0 ∨ π(I1, K1) = 0)
            π(J1, K1) := 0
            Changed := true
```

$\lambda = 0$:
`case I:`
$\mathtt{I}_0$`=((1,0),(2,0),(3,0))=`J_1`,`
$\mathtt{J}_0$`=((1,0),(2,0),(3,0))=`K_1
`disables`
$\mathtt{J}_0$`=((1,0),(2,0),(3,0)),`
$\mathtt{K}_0$`=((1,0),(2,0),(3,0))`

[12] This kind of notation is here used as short form for ($\pi(I_1, J_1) = 0$ leads to $\pi(J_1, K_1) := 0$). Else the same rules apply as described in the previous footnote for case I+H.

At the beginning it applies $\pi(J_0, K_0) = 1$, because the tuple (J_0, K_0) has not been initialized to disabled.

I_0 is contained within (J_0, K_0) because all literal index- and epsilon tuples of I_0 appear in J_0 or K_0 or both.

It applies $\pi(J_1, K_1) = 0$ (from $\lambda = 1$), which is identical to $\pi(I_0, J_0) = 0$.

This fulfills the three `if ()` conditions of RULE 1, so that RULE 1 will set $\pi(J_0, K_0) := 0$.

The original pseudo-code definition of RULE 1 with clause variables of this consideration inserted reads as follows:

```
foreach J_0 ∈ PC
  foreach K_0 ∈ PC
    if (π(J_0, K_0) = 1)
      foreach I_0 ∈ PC
        if (I_0 is contained within (J_0, K_0))
          if (π(I_0, J_0) = 0 ∨ π(I_0, K_0) = 0)
            π(J_0, K_0) := 0
            Changed := true
```

4.2.8.2 Second Example

The following literal index range n, the following basis tuple (J_0, K_0) and the following initially false clauses are given:

$$n = 6, p = \{4, 5, 6\}$$
$$J_0 = (0x_1 \vee 0x_2 \vee 0x_3) = ((1,0),(2,0),(3,0))$$
$$K_0 = (0x_1 \vee 0x_2 \vee 0x_3) = ((1,0),(2,0),(3,0))$$
$$F1 = (0x_1 \vee 0x_3 \vee 0x_6) = ((1,0),(3,0),(6,0)) \mid \tau(F1) = 0$$
$$F2 = (0x_2 \vee 0x_5 \vee 1x_6) = ((2,0),(5,0),(6,1)) \mid \tau(F2) = 0$$
$$F3 = (0x_1 \vee 0x_2 \vee 1x_5) = ((1,0),(2,0),(5,1)) \mid \tau(F3) = 0$$

$p()$ contains $\{4, 5, 6\}$ because these literal indices appear neither in the basis J_0 nor in the basis K_0.

The solver does the following steps to disable the basis tuple:

INITIALIZATION:
`case F0: the INITIALIZATION has disabled (among others) a tuple` $(F1, J1)$ `with`
`F1=((1,0),(3,0),(6,0)),`
`J1=((1,0),(2,0),(5,0))`
`case F1: the INITIALIZATION has disabled (among others) a tuple` $(F2, J2)$ `with`
`F2=((2,0),(5,0),(6,1)),`
`J2=((1,0),(2,0),(5,0))`
`case F : the INITIALIZATION has disabled (among others) a tuple` $(F3, J3)$ `with`
`F3=((1,0),(2,0),(5,1)),`
`J3=((1,0),(2,0),(3,0))`

Because it applies $\tau(F1) = 0$ and $\tau(F2) = 0$ and $\tau(F3) = 0$, any tuple holding $F1$ or $F2$ or $F3$ will be initialized to a disabled tuple.

$\lambda = 2$:

case I+H:
I_2=((1,0),(3,0),(6,0))=$F1$,
J_2=((1,0),(2,0),(5,0))=$J1$
and
H_2=((2,0),(5,0),(6,1))=$F2$,
J_2=((1,0),(2,0),(5,0))=$J2$
disable
J_2=((1,0),(2,0),(5,0))=$J1$=$J2$,
K_2=((1,0),(2,0),(3,0))

At the beginning it applies $\pi(J_2, K_2) = 1$, because the tuple (J_2, K_2) has not been initialized to disabled.

I_2 and H_2 are contained within (J_2, K_2) because all literal index- and epsilon tuples of I_2 and H_2 appear in J_2 or K_2 or both. The only exception is the tuple $(6, 0)$ in I_2 and $(6, 1)$ in H_2. This is the conflict literal with $p = 6$ which is 'dropped' in (J_2, K_2).

It applies $\pi(F1, J1) = 0$ (from INITIALIZATION), which is identical to $\pi(I_2, J_2) = 0$. It applies $\pi(F2, J2) = 0$ (from INITIALIZATION), which is identical to $\pi(H_2, J_2) = 0$.

This fulfills the three `if ()` conditions of RULE 2, so that RULE 2 will set $\pi(J_2, K_2) := 0$.

$\lambda = 1$:

case I+H:
I_1=((1,0),(2,0),(5,0))=J_2,
J_1=((1,0),(2,0),(3,0))=K_2
and
H_1=((1,0),(2,0),(5,1))=$F3$,
J_1=((1,0),(2,0),(3,0))=$J3$
disable
J_1=((1,0),(2,0),(3,0))=K_2=$J3$,
K_1=((1,0),(2,0),(3,0))

At the beginning it applies $\pi(J_1, K_1) = 1$, because the tuple (J_1, K_1) has not been initialized to disabled.

I_1 and H_1 are contained within (J_1, K_1) because all literal index- and epsilon tuples of I_1 and H_1 appear in J_1 or K_1 or both. The only exception is the tuple $(5, 0)$ in I_1 and $(5, 1)$ in H_1. This is the conflict literal with $p = 5$ which is 'dropped' in (J_1, K_1).

It applies $\pi(J_2, K_2) = 0$ (from $\lambda = 2$), which is identical to $\pi(I_1, J_1) = 0$. It applies $\pi(F3, J3) = 0$ (from INITIALIZATION), which is identical to $\pi(H_1, J_1) = 0$.

This fulfills the three `if ()` conditions of RULE 2, so that RULE 2 will set $\pi(J_1, K_1) := 0$.

$\lambda = 0$:

case I:
I_0=((1,0),(2,0),(3,0))=J_1,
J_0=((1,0),(2,0),(3,0))=K_1
disables
J_0=((1,0),(2,0),(3,0)),
K_0=((1,0),(2,0),(3,0))

At the beginning it applies $\pi(J_0, K_0) = 1$, because the tuple (J_0, K_0) has not been initialized to disabled.

I_0 is contained within (J_0, K_0) because all literal index- and epsilon tuples of I_0 appear in J_0 or K_0 or both.

It applies $\pi(J_1, K_1) = 0$ (from $\lambda = 1$), which is identical to $\pi(I_0, J_0) = 0$.

This fulfills the three `if ()` conditions of RULE 1, so that RULE 1 will set $\pi(J_0, K_0) := 0$.

4.3 Polynomial Algorithm does not have the Restrictions of the Logical Resolution

As I tried to publish a prior version of this paper [13] in a journal, the peer reviewer recommended to reject my work. His argumentation was, without any formal proof, that my algorithm would just be a special case of a logical resolution, and it is long-since known that a logical resolution has an exponential worst-case complexity. Additionally, the peer reviewer inferred without further justification that my algorithm would detect large-enough pigeon hole problem CNFs as solvable, although they are known to be unsatisfiable.

In the following, I will hold against the following two facts: First, I will reveal a crucial difference between a logical resolution and my polynomial algorithm, which leads to diverging complexities of both algorithms. Secondly, I will report about a speed-optimized implementation of my polynomial algorithm, which detected the pigeon hole problem PH_6 as unsatisfiable. This correct result would not have been possible if the claim would be true that the polynomial algorithm was just a logical resolution.

The proceeding of a logical resolution is easy to understand:

"Two clauses C_1 and C_2 are said to be resolvable if there exists a literal u such that $u \in C_1$ and $\bar{u} \in C_2$. In this case a third clause C_3 can be defined, called the resolvent of C_1 and C_2, by $C_3 = (C_1 \setminus \{u\}) \cup (C_2 \setminus \{\bar{u}\})$"[6].

In my own words, the resolving step works as follows: If there are two clauses C_1 and C_2, which do have taken together two literals with the same literal indices but different epsilon values, then all literals of C_1 and C_2 are written into the newly created clause C_3, except the two literals with the conflicting epsilon values. This means that if C_1 has n_{C_1} many literals and C_2 has n_{C_2} many literals and e be the count of literals that appear in both C_1 and C_2, then C_3 has $n_{C_3} = (n_{C_1} - 1) + (n_{C_2} - 1) - e$ many literals. Example: $C_1 = (0x_1 \vee 0x_2 \vee 0x_4)$, $C_2 = (0x_3 \vee 1x_4 \vee 1x_5) \rightarrow C_3 = (0x_1 \vee 0x_2 \vee 0x_3 \vee 1x_5)$.

With the logical resolution, one can determine if a given SAT CNF is solvable or not. Therefore resolvents of clauses from the SAT CNF or/and previously generated resolvents are created until no more new resolvents can be created. If one resolvent being an empty clause appears then the SAT CNF is unsatisfiable. Proofs for the correctness of this proceeding can be found in usual literature, e.g. [6].

The following important fact can be observed: It is possible that C_3 consists of more literals than C_1 or C_2. For instance, if C_1 has 3 literals and C_2 has also 3 literals which do all not appear in C_1, then C_3 will have 4 literals. The resolvent of C_3 with 4 literals and another clause C_4 with e.g. also 4 distinct literals will result in a clause C_5 with 6 literals. This clarifies the problem with the resolution: We assume there is such a pool of clauses at disposal so that any resolvent must be created out of clauses with each at least 3 literals. Then that resolvent might have more literals than each of the two clauses it was created out of. So in this case, the length and therewith also the count of clauses grows with each resolution step. It is not excluded that exponentially many resolvents must be created until the resolution comes to an end. Indeed, this is the case when determining the solvability of such-called pigeon hole problem CNFs using the logical resolution. [14]

The crisis is that the logical resolution can create clauses with more than 3 literals, and this can cause an exponential complexity. The polynomial exact-3-SAT solving algorithm, as defined in this paper in 3, does with every application of RULE 1 or RULE 2 never output more literals in (J, K) than it accepted in (I, J), (H, J), (I, K) or (H, K). The formulas in section 4.2.3 make this clear formally: Any clause tuple

[13] There was no difference of the algorithm definition between the peer-reviewed paper and the document you are currently reading.

[14] To avoid too many off-topic details, here neither the definition of the pigeon hole problem nor the proof of the logical resolution's exponential solving complexity are given. Please consult customary literature, e.g. [6], if needed.

does exclusively consist of 3-SAT clauses. Therefore it is impossible that there are more than $|PC|^2$ (see 2.3) many clause tuples to be stored and processed by the polynomial solver. The polynomial algorithm is not designed to generate 4-, 5-, etc. SAT clauses at all, like the logical resolution does. So the polynomial solver does not copy the exponential time- and space complexity from the logical resolution.

The only doubt left is if the polynomial algorithm does detect large-enough pigeon hole problem CNFs really as unsatisfiable. This means, while the polynomial algorithm has a polynomial complexity, maybe it outputs a wrong result? The formal proof of correctness presented in this document in section 4 suggests that this cannot happen. To clear up doubt, I implemented a speed-optimized version of the polynomial solver [15] and made it solve the pigeon hole problem PH_m for $m = 6$. m refers to the problem size in this case. In [7] it is stated that every resolution refutation [(i.e. application of the resolution)] for a pigeon hole problem PH_m involves a (resolvent) clause that has at least $2(m^2)/9$ many literals. This means, any resolution proof for the pigeon hole problem for $m = 6$ requires a clause with $72/9 = 8$ literals. As my polynomial solver cannot store more than 6 literals in each of its internal clause tuples (J, K), it is thinkable that the polynomial solver could fail, i.e. output satisfiable. This was analogously claimed by the peer reviewer. However, the polynomial solver took around 5 weeks solving time on a Ryzen 5 1600 and has finally outputted "UNSAT". This means the polynomial solver did correctly solve also the pigeon hole problem in polynomial time and is so "not just a special case of a logical resolution". Although RULE 2 of the polynomial solver does also look for conflicting clauses I and H as input and lets the two conflicting literals vanish in the output (J, K), it is not a logical resolution in its established form. Therefore the requirement for exponential complexity does not compellingly need to apply.

5 Complexity Analysis

5.1 Polynomial Solver has Complexity O(nˆ18)

The size of the set PC is of great importance because the polynomial solver's main work consists substantially of looping through the set of possible clauses. There are $|PS| = O(n^3)$ many possible clauses, because we can place the three indices of all possible clauses using three nested loops, each having an iteration range not larger than 1 to n. Furthermore there are $2^3 = 8$ possibilities for each clause to choose the three ϵ values out of $\{0, 1\}$. But because this is a constant complexity, it will not be observed in the O notation. Regarding all possible combinations of x many possible clauses one time has a complexity of $O((n^3)^x)$. This is the case because we had to implement x many nested loops, each having an iteration range of 1 to $|PS|$.

Next, we determine the complexity of all 3 solving steps. We regard the 3 steps independently because they are executed sequentially.

INITIALIZATION Regard $J, K \in PC \Rightarrow O((n^3) \times (n^3)) = O(n^6)$.

RULE 1 Regard $I, J, K \in PC \Rightarrow O((n^3) \times (n^3) \times (n^3)) = O(n^9)$.

RULE 2 Regard $I, H, J, K \in PC \Rightarrow O((n^3) \times (n^3) \times (n^3) \times (n^3)) = O(n^{12})$.

We apply RULE 1 and RULE 2 at maximum up to the point all $O((n^3) \times (n^3))$ many clause tuples have been disabled. This means we apply RULE 1 and RULE 2 maximal $O(n^6)$ times, whereby RULE 2 is the most comprehensive operation. So we get a total complexity of $O(n^{12} \times n^6) = O(n^{18})$. In this consideration it was assumed that checking for containment is done in constant time. This can be achieved by pre-computing if I, H is contained within (J, K). The pre-computing would require $O((n^3)^4) = O(n^{12})$ for examining all required clause combinations. Similarly, the solver can also pre-compute for each possible clause if it appears in the SAT CNF. The pre-computing would require $O((n^3) \times (n^3)) = O(n^6)$ to loop through all possible clauses to check for appearance, multiplied with the SAT CNF's highest possible

[15] The such-called "PigeonHoleSolver" is in the zip file served for download, together with an HTML file presenting screenshots of the test run. The download URL is given in 6.

clause count. The pre-computing does not increase the final overall complexity because it is independent from the work with highest complexity.

There are several additional measures thinkable to speed up the polynomial solver. For instance, in RULE 1 and RULE 2 the I and H `foreach`-loops could be left instantly as soon as $\pi(J,K) := 0$ and $Changed := true$ has been set. It does namely not make sense to stay in those loops because it would happen nothing new except trying to set $\pi(J,K) := 0$ and $Changed := true$ again, what is surplus. The interested reader might think out and implement even more pre-computations and speed-ups. The polynomial solver's algorithm has been presented in 3 in a very non-optimized form only to keep it scarce and therefore easy to understand and prove.

5.2 Why the Polynomial Solver has a Polynomial Complexity

It is important to notice that each time RULE 2 is examined recursively in this proof, the count of required I (resp. $I1$, $I2$ and so on) and H (resp. $H1$, $H2$ and so on) doubles. Although there's this theoretical doubling and thus a supposed exponential growth of complexity, this is in practice not the case for the presented polynomial solver. The reason is that even in the most comprehensive RULE 2, not more than quadruples of possible clauses are regarded. As the count of possible clauses grows polynomially with the problem size n, it is impossible to get an exponential complexity. If the practical recursion of RULE 2 would be implemented using recursive procedure calls, quadruples, respectively contained sub tuples, would be regarded by RULE 2 multiple times in the recursive sub calls. If a tuple has already been disabled, it is surplus work to regard it multiple times in recursive sub calls. If a tuple can not be disabled because other tuples it depends on have not yet been disabled, it is again surplus work to regard this tuple multiple times in recursive sub calls.

Example: We assume:

$$J = (0x_1 \vee 0x_2 \vee 0x_3)$$
$$K = (0x_1 \vee 0x_2 \vee 0x_4)$$
$$I = (0x_1 \vee 0x_2 \vee 0x_5)$$
$$H = (0x_1 \vee 0x_2 \vee 1x_5)$$
$$I1 = (0x_1 \vee 0x_5 \vee 0x_6)$$
$$H1 = (0x_1 \vee 0x_5 \vee 1x_6)$$
$$I3 = (0x_1 \vee 1x_5 \vee 0x_6)$$
$$H3 = (0x_1 \vee 1x_5 \vee 1x_6)$$

(J,K) is here the basis tuple which is to be disabled. $I1$, $H1$, $I3$, $H3$ are initially false clauses. This means the INITIALIZATION rule has set $\pi(I1,I) = 0$, $\pi(H1,I) = 0$, $\pi(I3,H) = 0$, $\pi(H3,H) = 0$. This means in return RULE 2 does at next set $\pi(I,J) := 0$ and another usage of RULE 2 sets $\pi(H,J) := 0$. This happens because $I1$ and $H1$ are contained within (I,J). All literal indices and corresponding epsilon values ($0x_1$, $0x_5$) appear in I or J or both. The only exception is $p = 6$, because x_6 does not appear in I or J, and the epsilon value related to x_6 is 0 for $I1$ and 1 for $H1$. Please recall 2.7 and also 4.2.1.3, where the practical recursion is introduced. Similarly, $I3$ and $H3$ are regarded by RULE 2 to set $\pi(H,J) = 0$. Finally a third usage of RULE 2 sets $\pi(J,K) = 0$, because I and H are contained within (J,K) and it applies $\pi(I,J) = 0$ and $\pi(H,J) = 0$.

Now comes the crucial point: We assume in the same run of the polynomial solver it is next to be decided if

$$J' = (0x_1 \vee 0x_2 \vee 0x_4)$$
$$K' = (0x_1 \vee 0x_2 \vee 0x_3)$$

is to be disabled. These J' and K' are just J and K, but swapped.

The polynomial solver, as defined in 3, will still have internally saved $\pi(I1,I) = 0$, $\pi(H1,I) = 0$, $\pi(I3,H) = 0$, $\pi(H3,H) = 0$ and $\pi(I,J) = 0$ and $\pi(H,J) = 0$. So the polynomial solver can disable

also (J', K') by merely *one* usage of RULE 2. This is possible because I and H are contained within (J', K') and $\pi(I, J) = 0$ and $\pi(H, J) = 0$ are still saved since the disabling of (J, K). In contrast, an explicitly programmed recursive implementation of the polynomial solver would need to set $\pi(I1, I) = 0$, $\pi(H1, I) = 0$, $\pi(I3, H) = 0$, $\pi(H3, H) = 0$ and $\pi(I, J) = 0$ and $\pi(H, J) = 0$ again before (J', K') gets disabled.

Because of this waiver of re-doing recursive tuple disabling operations for each basis tuple (J, K), the polynomial complexity of the polynomial solver is achieved. We are likely lucky that the inner state of the polynomial solver can be saved in not more than clause tuples.

6 Further Reading

The present document explains the polynomial exact-3-SAT solving algorithm using mathematical notation. There's an older document version online which uses more linguistic paraphrases. Furthermore there are C++ sample implementations of the algorithm available which run on Windows or Linux. Since the publishing of paper DM-2.1, the sample implementations determine a concrete solution ('model') if the SAT CNF is solvable (the correctness of this feature has not been proven yet, so please see it as bonus). Besides the solver implementations, several applications for testing selected suppositions used in this document's proofs have been deployed. All these items can be downloaded from the author's homepage www.louis-coder.com.

7 Acknowledgments

I thank Mr. Mihai Prunescu, Simion Stoilow Institute of Mathematics of the Romanian Academy, for helpful tips and a reference to the polynomial algorithm in one of his articles (see [4], resp. [5]).

References

[1] Michael R. Garey and David S. Johnson: Computers and intractability: A guide to the theory of NP-completeness, W. H. Freeman & Co., 1979.

[2] Christos H. Papadimitriou: Computational complexity, Addison-Wesley, 1994.

[3] Bronstein, Semendjajew, Musiol, Mühlig: Taschenbuch der Mathematik, Verlag Harri Deutsch, Thun und Frankfurt am Main 2000, ISBN 3-8171-2015-X.

[4] Mihai Prunescu: About a surprizing computer program of Matthias Müller, https://imar.academia.edu/MihaiPrunescu (link checked 2018-November-11).

[5] Mihai Prunescu: About a Surprising Computer Program of Matthias Müller, Convexity and Discrete Geometry Including Graph Theory: Mulhouse, France, September 2014, Springer International Publishing, ISBN 978-3-319-28186-5_9, http://dx.doi.org/10.1007/978-3-319-28186-5_9 (link checked 2018-November-11).

[6] Schöning, Torán: The Satisfiability Problem, Lehmanns Media Berlin 2013, ISBN 978-3-86541-527-1.

[7] Anup Rao: More Pigeons, and a Proof Complexity Lower Bound, https://homes.cs.washington.edu/~anuprao/pubs/CSE599sExtremal/lecture5.pdf (accessed 2018-November-09).