NANOSAT

A POLYNOMIAL-COMPLEXITY ALGORITHM FOR SOLVING 3-SAT FORMULAE

P VERSUS NP

BY DAN VOICU

AUGUST 2021

1. ABSTRACT

P versus NP problem (question) is one of the hottest topics in computer science for decades and it debates the current computational limits. This document represents the full description of an algorithm that solves 3-SAT formulae in polynomial complexity, for any number of inputs and clauses. To sustain the proofs, a C# implementation version of the algorithm is provided, rather than the pseudocode. The implementation herein can be used for tests of correctness and performances. In fact, the name (NanoSat) is due to the minimalistic footprint of the executable built out of this implementation (20KB).

As is, algorithm' implementation is not supposed to compete against existing heuristic algorithms, although in many cases, based on the benchmarks described below, it ran faster or even much faster than some. Under the assumption that the reader is acquainted with the specifics of this sort of problem, no further details about its specifics will be described in this paper, unless they are required by the proof itself.

The proof used is based on the calculation of the order of complexity for its main loop, plus (in fact multiplied) the calculation of the number of cycles of the main loop until completion is achieved (either SAT or NON-SAT).

As benchmarks, a very large number of formulae has been used, both SAT and NON-SAT of different sizes, most well-known and already heavily used during several annual SAT competitions, and some ad-hoc-produced by a generic public engine. The C# implementation of the algorithm is meant to bring enough clarity to it and its proofs, plus some measurement for the performance. The current version uses a simple, straight-forward version of solution certification for the SAT formulae, and no certification for the NON-SAT ones, and that is for a reason which will later be explained. The main characteristics of the algorithm are:

- linear/sequential in the sense that it uses no parallel computation or multi-threading
- uses no heuristics
- uses no Boolean resolutions
- no clauses or literals trimming/elimination by other means than Boolean resolution such as "pure literals" search
- does not develop any sort of graph
- not searching for patterns within formulae' clauses and/or literals
- does not look for, nor makes any assumptions based on literals' occurrences/density
- no time, complexity or other factors reset strategies

- not searching for all 8 polarities combinations on any 3 distinct literals (which would make the formula NON-SAT – explained below), just assuming the absence of any, since searching for that is only a matter of optimization
- reduces the state space via a "Pruning by Assumption" technique (PBA)
- uses the concept of partial state, meaning that only a subset of size < n of literals in the formula has a value set/assigned at a certain point in time during algorithm's execution, while all the rest, can have any value, thus undecided, such as the state of a qubit in a quantum computer
- uses a temporary memory DB (MDB) with 128 bit-per-record hash, and a max size of is $O(n^6)$ records, representing all the subsets of assigned literals (visited partial states) encountered during a run, and which partial states are no longer to be reconsidered or accepted as a subset of a SAT solution, during the rest of the execution of the algorithm
- uses the classical concept of "units" for the literals that mandatorily must be assigned the truth value because the other two literals in at least one clause, are already set to False
- uses the classical concept of "conflict" or "collision" for the case where both polarities of some literal mandatorily must be assigned the Truth value (the literal is becoming a unit for both its polarities or, one of its polarities is already set to True and belongs to list of assigned values, while the other polarity becomes a unit)
- each run/execution of the main loop (RunEngine), if no conflict arises, will set one or more undecided literals (assign the Truth value to one of its/their polarities), thus diminishing the number of the unset literals during each successful main loop execution.
- once a conflict occurs during a main loop run, then one of the assigned literals will be unassigned (brought back in unassigned state), while the partial state where such a conflict has occurred is not going to be repeated/accepted during algorithm's run, thus eliminating (pruning) all subsequent partial states via PBA
- agnostic to the starting literal, meaning that the algorithm can start using any of the n inputs, on any polarity
- execution' raw/true complexity will generally be different for each individual literal/polarity the algorithm will start with, sometimes with a difference of magnitude n. The differences in complexity come from the fact that, at its essence, any such formula represents a weighted non-directed graph where the costs from any starting node to the target/end node may be different
- current implementation of its "Engine" is achieved with around 100 lines of executable C# code, while the rest of the lines are only formula-file parsing, data format validations, data-structures manipulations (initializations, loading, etc.), statistics recording, logging and so on
- the order of complexity is $O(n^6)$

As is, algorithm' performance can be greatly improved, but this is not the objective of this document. Due to the restrictions above, its performance is surpassed by most of the current professional 3-SAT engines. However, one of the purposes of these restrictions is the fact that, proving that it runs in polynomial time becomes a much easier task. Some optimizations will be obvious after reading this document. Nevertheless, at the end, there is a non-exhaustive list of the most obvious optimizations which will accelerate algorithm's execution without trading either memory or its upper bound complexity order.

Before starting to read the following chapters, perhaps one would like to start directly with the performance chapters (12 to 14) and run algorithm' implementation. Some questions for which the answers are in the next chapters, might be raised.

2. NOTATIONS

Symbol	Description
Φ	3-SAT formula in CNF format
n	number of literals (Boolean inputs) within Φ; $3 \leq n \in \mathbb{N}^*$
m	number of CNF clauses within Φ; $1 \leq m \leq 7 \binom{n}{3}$
X_k	some literal in Φ, $1 \leq k \leq n$ with undefined polarity (0 or 1)
$\neg X_k$	opposite polarity of X_k
$Cl[X_k]$	non-empty, complete set of clauses where X_k occurs
λ	list of non-conflictual literals having a value assigned (X_k and $\neg X_k$ are mutually excluded)
$\neg\lambda$	opposite of λ, list of literals that have no assigned value yet
$\sigma(param)$	size of param
$\sigma(Cl[X_k])=\sigma(X_k)$	size of number of clauses that X_k is part of
$\sigma(\lambda)$	size of λ at any point during A run; $0 \leq \sigma(\lambda) \leq n$
$\pi(\lambda)$	partial state of selected literals based on the content of λ with $1 \leq \sigma(\lambda) \leq n - 3$
μ	literal or literals that has/have the properties of a unit (its/their value must be set to True since in at least one of the clauses it appears, the other two members are set to False)
$\lambda\mu$	temporary list of μ
β	subset of immutable literals for any SAT solution of Φ, called a "backbone", $0 \leq \sigma(\beta) \leq n$
$\mathcal{A}$	NanoSat - current algorithm
$\Omega(\mathcal{A})$	min $\mathcal{A}$ complexity
$\Theta(\mathcal{A})$	avg $\mathcal{A}$ complexity
$O(\mathcal{A})$	max $\mathcal{A}$ complexity

1. Φ RULES

Name	Description
R1	each clause $C_i \in \Phi$, $(1 \le i \le m)$, contains exactly 3 unique literals
R2	each clause $C_i \in \Phi$, $(1 \le i \le m)$, is unique
R3	each clause $C_i \in \Phi$, $(1 \le i \le m)$, does not contain any opposite literals
R4	all literals within clauses are unique (no duplicate literals)
R5	any combination of 3 literals can have at most $2^3 - 1 = 7$ unique representations within Φ clauses
R6	due to R1 – R5, m cannot be larger than $7 \binom{n}{3} = \frac{7}{6}[n\,(n-1)\,(n-2)]$ otherwise Φ *is* NON-SAT
R7	each literal X_k, $(1 \le k \le n)$, has at least one occurrence in Φ

2. $\mathcal{A}(\Phi)$ FUNCTIONS

Name	Type	Description	Complexity
Add	void	adds a literal at the bottom of a list	$O(1)$
Insert	void	adds a literal at the top of a list	$O(1)$
Remove	void	Removes a literal from a list	$O(1)$
GetNext	void	selects an unused literal to be inserted or added into λ	$O(n)$
FindUnits	void	searches for μ and returns a list of them	$O(n)$
GetUnits	bool	searches (via FindUnits) and inserts newly found μ into λ. Returns false if a conflict occurs	$O(n)$
GetAllUnits	bool	iterates through λ and uses each pair of literals searching for μ. Returns false if a conflict occurs	$O(n^2)$
GetOppUnits	bool	iterates through λ and searches for μ, based on $\neg X_k$ for each X_k in λ. Returns false if a conflict/collision is detected	$O(n)$
SaveState	bool	saves a 128-bit hash of the current λ state if the state has not yet been visited, returns false if the same hash already exists	$O(1)$
IncrementHeader	bool	once λ is empty, this function choses the next unused X_k as a starting point (header). The new X_k can have any polarity. Returns false if no literal is left to be used as a header while Φ is considered NON-SAT	$O(1)$
RunEngine	bool	main loop – it runs until either $\sigma(\lambda) = n$ (SAT case) or $\sigma(\lambda) = 0$ && *there are* no more starting literals to use (NON-SAT case)	$O(n^2)$
Certify	bool	certifies that the SAT solution found is valid	$O(1)$
Opposite (or Opp)	Int	returns the opposite polarity of a literal ($\neg X_k$)	$O(1)$

3. SAT ENGINE DIAGRAMS

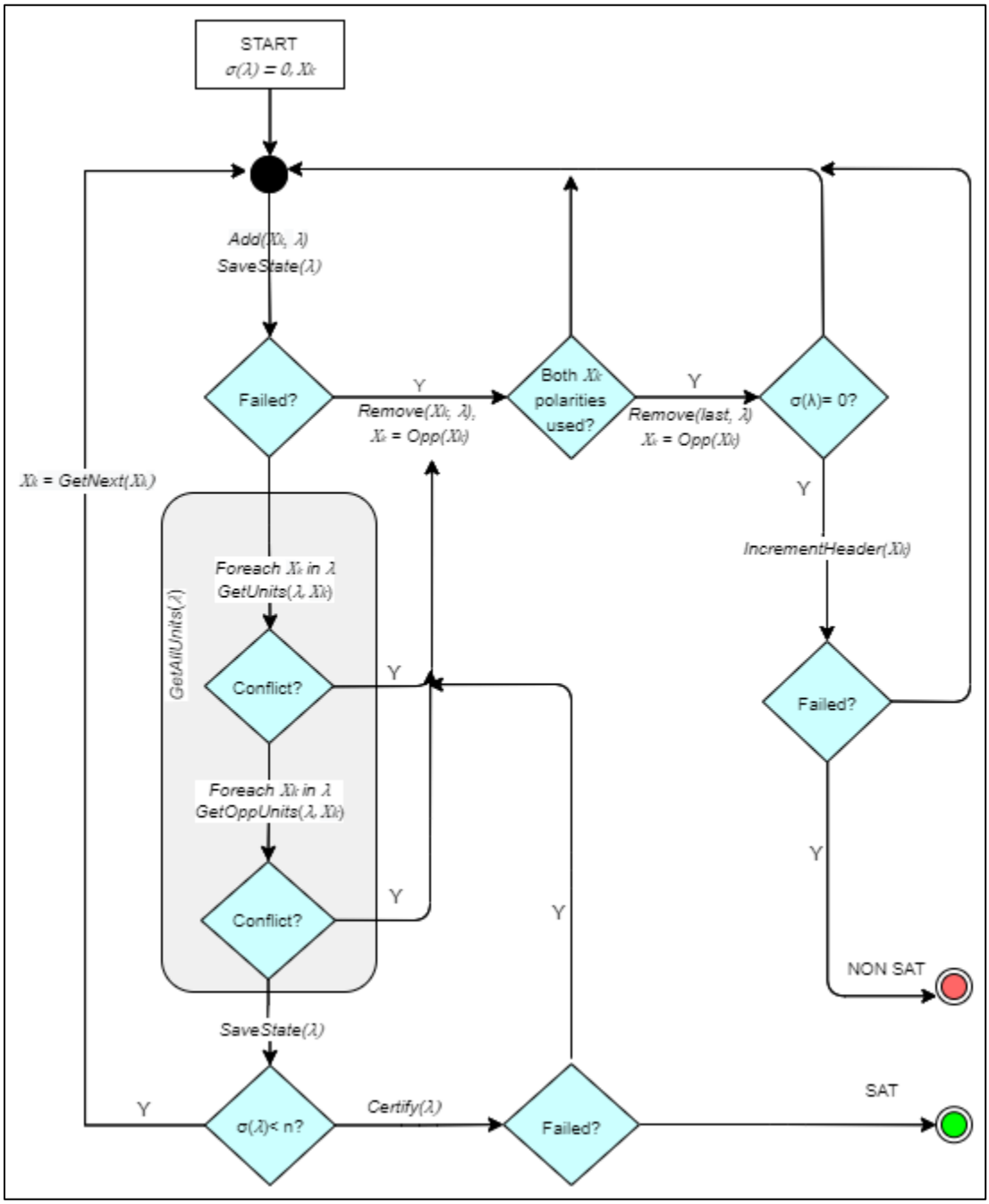

Figure 1 – Diagram of RunEngine - Main Loop

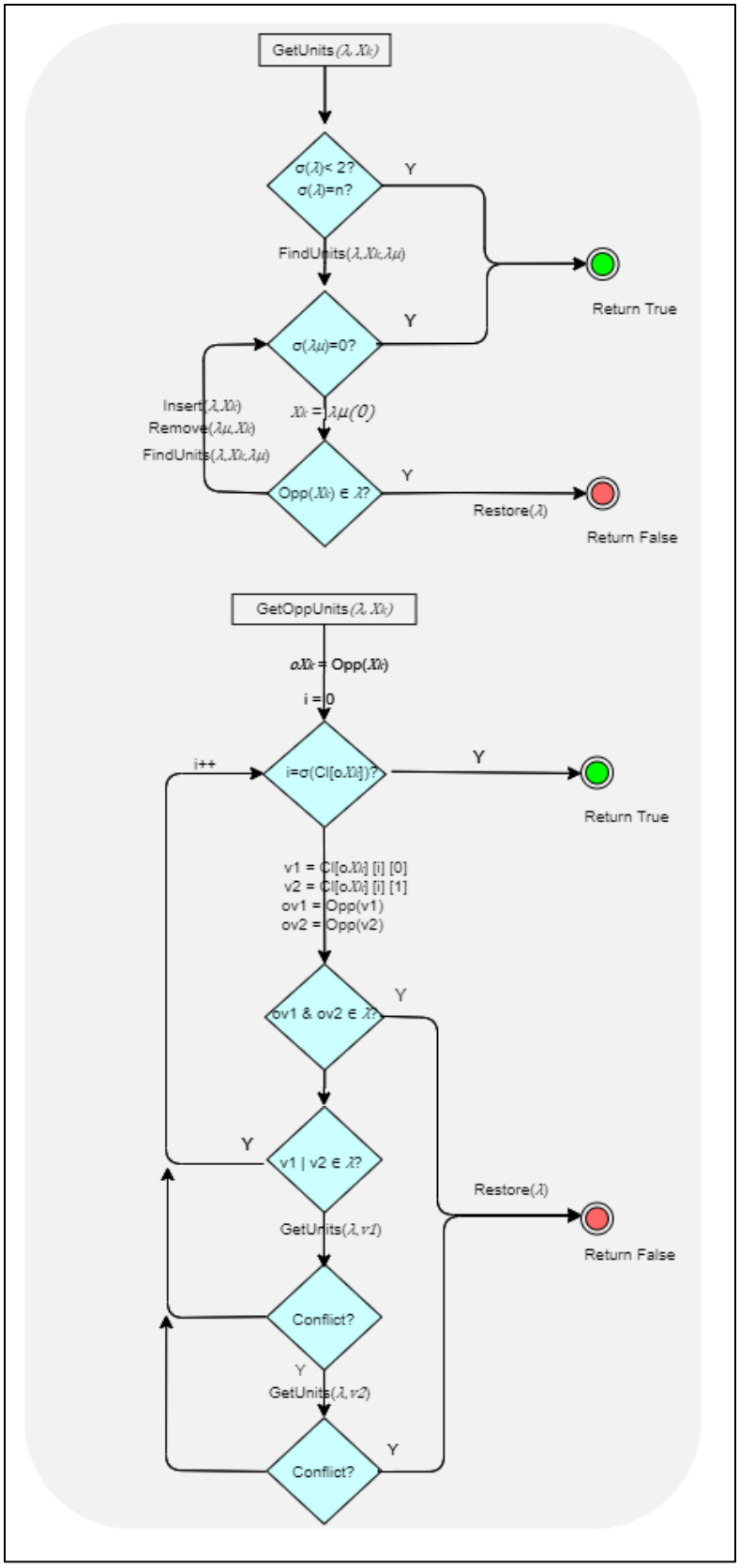

Figure 2 – GetAllUnits Block Diagram

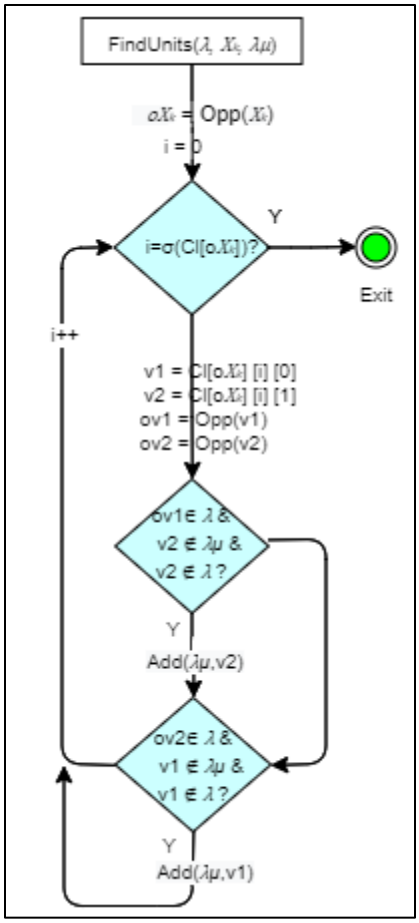

Figure 3 – FindUnits Diagram

4. DESCRPIPTION OF $\mathcal{A}$

The main idea is to populate λ with n unique literals on one of their polarities. Each of λ elements will then be considered as set to True and together will form one of the Φ SAT solutions. In case Φ has no solution then λ will be empty at the end of the run.

LAMBDA'S PARTIAL STATES & PBA

The list that will contain Φ solution (λ) at the end of the run, is gradually filled from empty to n elements. The algorithm starts with an empty list and one of the n input literals on one of its polarities. At each step during algorithm's run, λ will contain zero or more out of the n literals, each of the literals occurring only once, on one of their polarities. A partially filled λ constitutes a "partial state", herein referred to as $\pi(\lambda)$. Any $\pi(\lambda)$ contains only literals whose values are set, while all the rest of the literals not included in λ have no value assigned yet. For example, if $\sigma(\lambda)$ equals p (set literals) and $\sigma(\neg\lambda)$ equals q (unset literals), then $p+q = n$. The idea of using this kind of state space representation comes from the necessity of trimming the state space under certain conditions, with the assumption that, if a certain $\pi(\lambda)$ has reoccurred, it did so because at some point during the run of the algorithm, some literal has produced conflicts on both of its polarities and therefore that literal will always be refuted by that unique $\pi(\lambda)$. The process of eliminating conflicting states is called in this paper Pruning by Assumption or PBA. The reason that $\pi(\lambda)$ will always refute that literal in the future is because these conflicts only arise via searching for μ, while the components of λ will always yield the same μ when combined with the conflicting literal. As a conclusion, that unique $\pi(\lambda)$ will be considered a non-acceptable state of λ and be avoided each time it reoccurs. As a direct effect, many other states will be eliminated together with the $\pi(\lambda)$ in question, namely, part or all states represented by the remaining q unassigned literals, totaling 2^q states. During any main loop run, all $\pi(\lambda)$ will be recorded both in the beginning of the run as well as at the end of the run (if that run has completed successfully without a conflict). However, if at the beginning of a main loop run $\pi(\lambda)$ has been detected as already reached in the past, that run will be considered as a non-successful one, for the reasons mentioned above, therefore skipped. More about PBA philosophy is described in the PBA chapter.

FUNCTIONS' DESCRIPTION

FINDUNITS

This function is searching for μ produced by one of λ elements (X_k) and returns a list ($\lambda\mu$) of them, by using $\neg X_k$ and browsing through all its clauses. The process is depicted in Figure 3 above. One instance of each distinct μ found, will be part of $\lambda\mu$ at the end of each execution. The logic behind this exhaustive search yields: look at each clause that $\neg X_k$ is part of, and, if one of the other two clause' elements is set to False (its opposite is part of λ), then the third element mandatorily becomes a μ for that particular clause to be SAT and will be added to $\lambda\mu$. Adding any new element with such a property into $\lambda\mu$ is done without checking for its opposite polarity counterpart existence in $\lambda\mu$. This action will be later achieved during GetUnits execution which is the function that calls/uses FindUnits.

GETUNITS

The inputs for this function are λ and X_k, where $X_k \in \lambda$, and its purpose is to exhaustively search for all μ produced by X_k and, once new μ are found they will also be used to search for other subsequent μ. Any newly found μ is then inserted on the top of λ. The insertion will happen iff there is no duplicate element to λ, nor its opposite polarity ($\neg\mu$) is part of λ. The function will return False in the case there is a $\neg\mu$ already part of λ or $\lambda\mu$ which means that a certain element X_p and its counterpart $\neg X_p$ should both be assigned with the Truth value, thus a conflict. Function's dataflow is presented in Figure 2, as part of the GetAllUnits block.

GETOPPUNITS

The logic and the functionality are the following: since X_k is part of λ (therefore is being assigned the Truth value), $\neg X_k$ will be set to False. This means that for each clause $\neg X_k$ is part of, at least one of the other two members (say *m1* & *m2*) either already is part of λ or it must accept the Truth value for that clause to be SAT. This way $\mathcal{A}$ validates the fact that all clauses containing $\neg X_k$ can be SAT. If one of m1 or m2 is already part of λ that clause is considered as SAT, but if not, then the function sequentially sets the Truth value to each of them and checks if the assignment does not produce a μ conflict. If one of *m1* or *m2* can accept the Truth value, the eventual μ brought via GetUnits can be left in λ as an optimization (it is used in the current implementation). On the contrary, if both *m1* & *m2* of some clause create a conflict (therefore at least one clause is NON-SAT), the function returns False and X_k is rejected from λ. $\pi(\lambda)$ at this stage is already recorded via SaveState and refuted every time it will reoccur. At this point, neither *m1* nor *m2* can be added to λ since the function only certifies that there is at least one path among all possible paths within all $\neg X_k$ clauses that will lead to SAT for every single clause $\neg X_k$ is part of, but that path is not necessarily the one to be used from that point on. Therefore, assigning a value to either *m1* or *m2* will later be done via the selection logic, or as μ. The dataflow for this is presented in Figure 2 as the second part of the GetAllUnits block.

SAVESTATE

This function that takes λ as its input, orders its content in ascending order, creates a 128-bit hash and then checks the in-memory DB for its existence. If the new hash is unique, the function adds it to MDB returning True, and if it is not unique it returns False without saving a duplicate into DB. The SaveState is called before entering the GetAllUnits block and if it fails, then the block is no longer executed. It is also executed once after GetAllUnits in case it ends successfully, this time with the sole purpose of recording the newly resulting state.

RUNENGINE

The main functionality/loop is represented and executed by this function and its dataflow is shown in Figure 1. As inputs, there are two elements: an empty list called λ in this document, as well as one of the n inputs on one of its polarity as a starting literal, herein referred as X_k. The Diagram in Figure 1 shows the dataflow of this function. RunEngine uses a vector (called "header") as means of tracking the starting literal. Any literal on each of its polarities, can and will be used once (only if necessary) during the run and, as soon as the header is filled and the loop is exercised until λ becomes an empty list, the main loop ends returning a NON-SAT result. A conclusion emerges from this: main loop will "restart" at most $2n$ times from an empty lambda, using a new value as starting literal. Each time a new starting literal is exercised and then changed, the MDB data ($\pi(\lambda)$ hashes) is conserved and used for all other following starting literals' runs. The $\pi(\lambda)$ history, due to its significance, ensures the fact that the algorithm will not loop forever, nor uselessly, returning to states already visited and which will certainly lead to a conflict. Before

deciding to set the last recorded $\pi(\lambda)$ as a non-accepting partial-state, the algorithm also flips the polarity of the X_k in question (the one that, based on the current $\pi(\lambda)$, creates a conflict) and tries another run using $\neg X_k$. If the second run fails, the last inserted literal into λ is going to be removed, and the run will continue by trying to fit, yet again, the same X_k into λ. This sort of "trial-and-removal" will continue until either X_k or $\neg X_k$ has a non-conflicting place into λ, and $\sigma(\lambda)$ continues to grow, or λ is emptied up and a new starting literal (if available) is being chosen. One important detail is the fact that all μ are inserted on the top of λ (stack-like), while all artificially set literals are added at the end of λ (list-like). When removing a literal from λ, that literal will be an artificially set one, and once these are exhausted, then, the μ will start being removed from λ until either X_k (on one of its two polarities) finds its place into λ, or until λ is empty and $\mathcal{A}$ will start with a new literal. There are some other details that need to be clarified as for instance why the second SaveState call result is ignored after a successful run. It is so because its outcome at this point has no impact on the overall functionality since the successful main loop run will, at worst, bring no extra μ into λ while the same $\pi(\lambda)$ has already been recorded at the beginning of the loop, but if it does bring any extra μ, a new $\pi(\lambda)$ will then be recorded. The only question here is "what happens if the new $\pi(\lambda)$ resulting from a successful run has a duplicate in MDB?". If this case occurs when no μ was added (so it is the same $\pi(\lambda)$ as at the beginning of the run) then it is expected, otherwise future runs will unveil the duplicate. In Figure 4 below is depicted the fluctuation in size of λ when the algorithm is hitting conflicts and trims states until is reaching a non-conflicting $\pi(\lambda)$ to insert literal X_{k4}

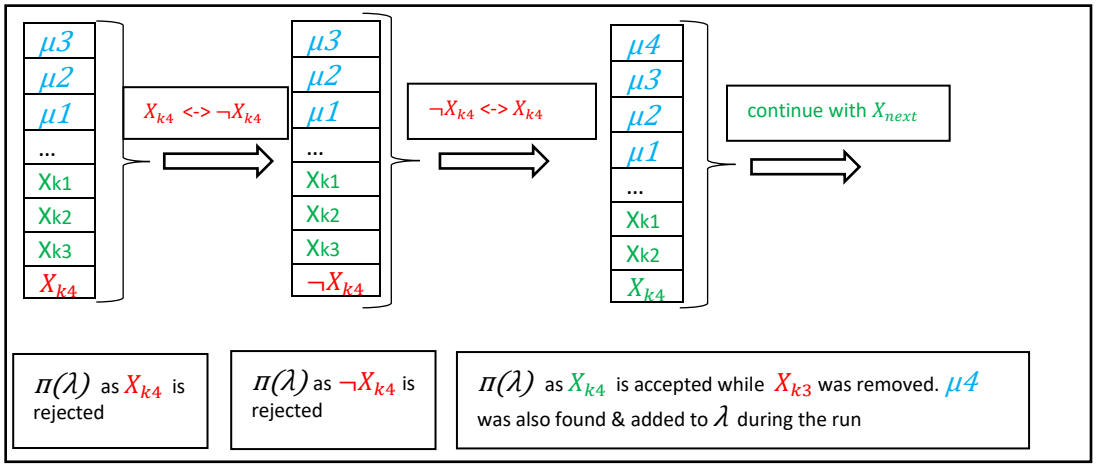

Figure 4 – $\sigma(\lambda)$ Fluctuations while a literal is inserted into λ

The graph in Figure 5 below shows the complete fluctuation cycles of $\pi(\lambda)$ in terms of $\sigma(\lambda)$ during the run of a SAT Φ. The main loop ran for 1521 cycles, among which 19 represent a similar literal insertion as the one depicted above in the last step, while all the rest (the descending slopes) occur just like the first two stages in Figure 4 above. The X scale represents the cycle number while Y represents the number of set-value literals contained in λ. Noticeable is also the fact that once a literal is successfully inserted into λ, $\sigma(\lambda)$ suddenly grows due to the new μ found. Other remarque, perhaps not so obvious but important nevertheless: λ never went down to zero elements. This means that one literal only (X_1 in this run) was used as a starting literal. The formula used as example was solved in exactly 1 sec (CPU time) on a 3 GHz laptop, with a measured complexity of $n^{3.02}$, and belongs to a one solution Φ (see the "Backbone" description below).

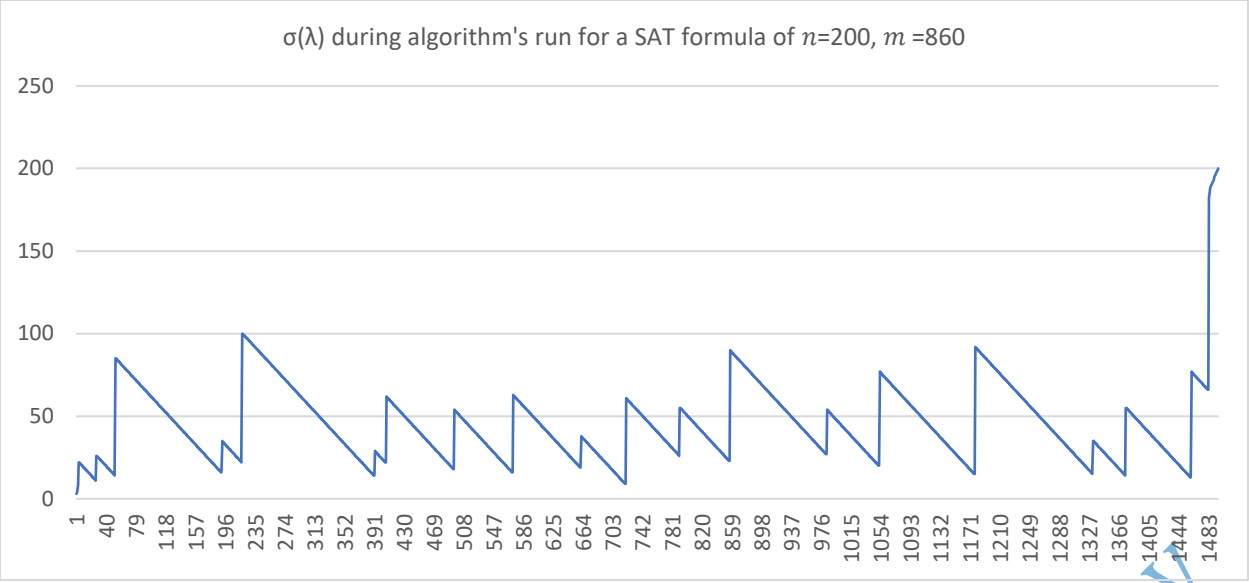

Figure 5 – $\sigma(\lambda)$ During a full run of $\mathcal{A}$ from empty to n elements

5. $\mathcal{A}$ FUNDAMENTALS

THE BACKBONE

Before explaining why and how $\mathcal{A}$ works as it does, a few details about the concept of "backbone", herein referred to as β, will be analyzed. β is a fixed subset of the n literals which, for Φ to be satisfiable, this subset is immutable in terms of assignments of its literals (their polarity), for all SAT paths, therefore any single SAT path of Φ will contain the β subset. Identifying the backbone reduces the search space for getting to a SAT solution of Φ. For instance, if n=100 and $\sigma(\beta)$ =90, then the remaining solutions space is in order of 2^{10} since the rest of the space of 2^{90}, once identified, is fixed and unique. In this example ($\sigma(\beta)$ =90), finding the backbone yields a sole solution in the space of 2^{90} possible combinations. In theory, the closer $\sigma(\beta)$ is to n, the harder is to identify the backbone, and that is because the search space for the unique literals subset and polarities combination is closer to 2^n. The biggest $\sigma(\beta)$ is n, case in which Φ accepts one SAT solution only (a single SAT path out of 2^n possible paths). In exchange, $\sigma(\beta)$ = 0 means that Φ accepts at least one SAT solution for any polarity of any of its n literals, therefore with a much smaller degree of complexity. Identifying β requires two steps: one is to identify its literals, and second is identifying the right polarity for each of them.

For explaining how $\mathcal{A}$ builds up β, the case of the state where σ(λ)=0 will be considered, case where a non-β literal is to be inserted in it. Since any such literal is part of a SAT solution of Φ it will not produce a conflict in combination with any incoming β literal. However, it can produce one with another incoming non-β, but not on both polarities of the latter. Due to the fact that any incoming literal, in case of conflict, will be flipped on both polarities before removing anything from λ, then the latter will get its place into λ while the initial one will not be removed.

Then what if a β-literal is the first to be inserted into λ and it is inserted on the wrong polarity? In this case either it will be removed right from the very beginning if the next chosen literal belongs to β as well and is on its right polarity while the pair produces a conflict. In this case the first one will be brought back later either as a unit or reinserted and flipped-to-fit until it gets into λ on its right polarity, or when it becomes a starting literal again, this time on the right polarity since $\mathcal{A}$ accepts a starting literal on a polarity only once. Otherwise, it will later be removed via the same PBA process and brought back at a later cycle on its right polarity.

In case the first literal is part of β, and it is inserted on its right polarity, then that one will produce no conflict with any other incoming next, β or non- β, and that is because the latter is flipped both sides before removing one literal from lambda, so either way one of its polarity will be accepted.

Next, assuming the case where π(λ) at some point contains only non- β literals (therefore no conflicts since this is the meaning of λ) and some $X\beta \in \beta$ is about to be inserted into λ and produces conflicts, that literal will be flipped on its both polarities while elements of λ will be removed through the process until either λ gets to one last element and Xβ will be accepted into lambda (case where the wrong polarity can occur and was explained above) or Xβ will be accepted earlier than reaching the last element.

In the case where λ only contains elements of β and a non- β is to be inserted then, it will be accepted without a conflict on any of its polarity, without removing a β member because it is one literal for which Φ has a solution on any polarity, as stated before.

What about the case where λ contains a mixture of β and non-β elements? In this case, if some Xβ is about to be inserted then it may produce conflicts due to a number of non-$\beta \in \lambda$ at that point and, this situation is resolved by eliminating literals from λ until Xβ gets a non-conflicting place into λ.

GETALLUNITS BLOCK - THE EXHAUSTIVE SEARCH FOR UNITS

The core functionality of $\mathcal{A}$ resides within the GetAllUnits block. For any newly added X_k to λ, no matter its polarity, $\mathcal{A}$ will exhaustively search for units produced by the addition of this literal. The process is divided into two main steps: GetUnits & GetOppUnits. During the first part of the search (GetUnits execution), every pair produced by X_k together with the rest of lambda components are used to search for μ, by using their opposite polarities. So X_k combined with all $X_j \in \lambda$ / X_k will form the pair $[\neg X_k \neg X_j]$ while the third literal (a μ) will be looked for, in every clause containing that pair. Once such μ is found it will be added to the $\mu\lambda$ and then further be combined in the same manner as X_j and used to further search for other μ. If no conflict arises for any newly found μ, then σ(λ) grows by the amount of μ found. A conflict means that both μ and $\neg\mu$ should be added to λ via this process, or the latter already is part of λ. As such case occurs, the execution of the process is dropped, and the entire block fails and exits, while λ is being restore at the state before the block's execution. If successful, GetUnits is then repeated σ(λ) times, for all members of λ in that cycle of the main loop. Once the first step of GetAllUnits is successfully completed, GetOppUnits execution is next, process which, in the same manner as the previous is going to exhaustively search for μ, using each member in λ. The process is different from the previous because instead of using a pair of opposites from λ, uses only one opposite ($\neg X_k$) and considers that, every clause $\neg X_k$ is part of, in order to be satisfied, must have at least one member which either already exists in λ, or if set to True, will not produce a conflict. In case for some clause none of the two members accepts the Truth value (or already are present into λ with their opposite polarity), then the execution is dropped and the entire GetAllUnits block fails. If GetAllUnits execution succeeds, all newly found μ are now part of λ. By contrary if it fails, no μ is added to λ and, either X_k is replaced by $\neg X_k$ in λ and the process restarts (flip-to-fit), or, if both X_k and $\neg X_k$ have been cycled and failed, then the bottom-most literal (the closest to X_k) will be removed (added to ¬λ) while the process restarts by trying again to insert X_k into λ. The process will continue until either X_k or $\neg X_k$ gets a place into λ, or λ becomes empty via the removal of all its literals. The process is visible in Figure 5 above, where the descending slopes are produced by the removal of literals due to GetAllUnits failures, while the sudden ascending ones are due to X_k insertion into λ plus the newly μ found via this block' execution.

THE "HEADER"

Earlier in this document, a linear structure named "header" was mentioned. It represents the means of keeping track of the starting literals, so they do not repeat, and to make sure that when all starting literals have been used and λ reaches the empty state for one last time, the algorithm returns the NON-SAT result. The first literal is added into this structure at the very beginning of algorithm's execution, while for the rest, the sole condition under which a new literal is added to the header is that λ reaches the empty state again. This event occurs once the top literal of λ, which is also the only one contained in it, together with the one to be inserted (in the earlier example mentioned as X_k) are the only two left and they produce conflicts on both polarities of the latter. This situation can occur right from very first two literals used in the current loop or after several cycles. However, having two such conflicting literals does not mean the formula is NON-SAT. It only means that the pair is generating a chain of μ in which at least one literal proved to be conflicting. Although the header is rather an implementation detail more than $\mathcal{A}$ specific, it will be detailed in order for the rest of the document to be easier to be explained.

THE WAY $\mathcal{A}$ WORKS

Many well-known 3-SAT algorithms use, among other techniques and strategies, some sort of literals' backtracking. $\mathcal{A}$ does not backtrack on literals but rather builds up β via state space pruning using PBA, process that eventually also brings some of the other literals not part of the β into λ, as μ. Also, several literals part of the β may be identified throughout the PBA process, as μ produced after finding the place for some X_k into λ. This is one of the reasons $\mathcal{A}$ is agnostic to the starting literal or its polarity. The "trial-error-remove" herein referred to as "flip-to-fit" PBA method of inserting any X_k or $\neg X_k$ into λ has the purpose and effect of selecting the β literals, on their correct polarity.

As a philosophy, any π(λ) is considered as being part of a SAT solution for Φ and that implies that any literal that is missing from that particular λ (so it is part of $\neg\lambda$) should be able to be added to λ on at least one of its polarities. If not, then that particular π(λ) cannot be part of any SAT solution of Φ. As tests (described in chapter 12), among others, $\mathcal{A}$ solved several known Φ with sizes between 50 and 200 inputs, for which $\sigma(\beta) = n$ (one SAT solution Φ in the space of 2^n). During the runs, $\mathcal{A}$ was restarted using as starting literal each one of the n inputs, on both polarities (so $2n$ runs for each Φ), while the results came up the same since each Φ had one SAT solution only, with a low, measured, polynomial complexity. For all the runs the saved partial states were not conserved while solving the same Φ using different starting literal/polarity, so no history was conserved from one run to the next.

WHY Λ CONTAINS A SOLUTION OF Φ

The justification yields:

1. At the end of the run, when Φ is resolved/exits as SAT, λ contains n distinct elements on one of their polarities – so no conflict
2. Each literal in λ is either a μ, therefore the containing clauses are set to True, or an artificially inserted literal which has been flipped on its polarities until it fitted into λ without producing a conflict, via the RunEngine logic, so the same, all containing clauses these literals are part of, are set to True
3. Since n is the maximum number of inputs, all λ components are part of (cover) all clauses. But even so, this question may arise: "how certain one can be, that these components' polarities do not leave some clauses set to False (no literal set to True contained in it)?". And the answer is:
4. The proof that no clause is being left with the False value (NON-SAT) is guaranteed by the functionality of GetOppUnits, which is applied to all literals inserted into λ, whether units or "artificially" added, and therefore all clauses part of Φ. Besides the fact that it eventually adds more μ, it ensures the fact that no clause is left without at least one member set to True, or else it fails, and if so then λ does not fill up to n elements, therefore $\mathcal{A}$ will not exit

6. $\mathcal{A}$ COMPONENTS' COMPLEXITY

Algorithm's main loop is depicted as RunEngine in Figure 1. This function calls the "GetAllUnits" block which is the most expensive functionality in terms of cycles/complexity, plus the rest of the functions which have a much lower complexity compared to GetAllUnits.

FINDUNITS

As shown in Figure 3, the function runs for a maximum of σ(Cl[$\neg X_k$]) (so $\sigma(\neg X_k)$) steps, therefore its maximum complexity is represented exactly by the maximum occurrence of $\neg X_k$ $(\mathrm{Max}(\sigma(\neg X_k)))$, thus a constant: O(1). However, if we consider the largest input possible per R6, this function can reach O(n^2) when $\neg X_k$ is part of all unique combinations of any other two literals since these are $\binom{n}{2}$. It is highly unlikely that such a Φ will ever be created, and if so, it will have any SAT solution.

In order to explain the upper bound of the clauses as per R6, consider the fact that, having n inputs grouped by 3 would result in a maximum $\binom{n}{3}$ unique combinations, multiplied by their 2^3 polarities variations so 8*$\binom{n}{3}$. But if all 8 unique variations (disjunctions of 3) are present for any clause of 3 unique literals, then those variations will also include the "all set-to-false" clause, thus the containing Φ will be NON-SAT. In conclusion, the highest number of clauses for any Φ obeying R1 to R5 and "hoping" for a SAT solution, for any 3-input combination, is 7*$\binom{n}{3}$, and that is what R6 states.

As a conclusion, in worst case, this function cannot take more than $\mathrm{O}(n^2)$ or else R6 is violated.

GETUNITS

This function uses FindUnits for as long as this function brings new μ throughout its execution and, since the maximum number of μ cannot be larger than n, this function' complexity is bordered by n*FindUnits, therefore $O(n)$.

GETOPPUNITS

As shown in Figure 2, this function is calling GetUnits twice in worst case. This means the function will run 2*GetUnits so $O(n)$.

GETALLUNITS

This functionalities block is calling GetUnits and GetOppUnits for each element in λ. Since $\sigma(\lambda)$ cannot be larger than n the total complexity is n*GetUnits + n*GetOppUnits which yields $O(n^2)$ in worst case.

ALL OTHER FUNCTIONS

The rest of the functions part of main loop (RunEngine), are trivial, with their worst case in $O(n)$ so they are not worthy of any further analysis.

RUNENGINE

Representing the main loop of the algorithm, this function will take the complexity of its most expensive execution block, and that is GetAllUnits, therefore its complexity is $O(n^2)$. If the unlikely real upper bound of FindUnits as being $O(n^2)$ is being considered, then the complexity of the main loop grows accordingly to $O(n^4)$ and, in this case, the main loop will be considered in $\Theta(n^2)$. As explained above though, the real upper bound of FindUnits does not make much sense to be used.

7. PBA & UNITS

PBA is based on the assumption that, if a partial state π(λ) is part of a solution then any unassigned literal that is part of ¬λ should be accepted (no conflict) on at least one of its polarity. On the contrary, if one of the literals from ¬λ will produce conflicts on both of its polarities, that π(λ) cannot be part of any solution path, therefore discarded in future runs. At any cycle of the main loop execution, all encountered π(λ) are saved into MDB. All π(λ) are formed by the literals part of λ which have a polarity set, while its opposite list, ¬λ , contains all other literals with unassigned/unknown/free polarity. The purpose of recording each partial state is to avoid repetitions of states which are known as not being part of a solution while executing $\mathcal{A}$. Therefore, reaching a certain π(λ) and trying to add any random literal Xk from ¬λ to it, an insertion without conflict is expected on one of Xk polarities if that π(λ) is part of a SAT path. In case this is not possible, then that particular π(λ) is considered as a non-acceptant π(λ), thus discarded for future runs. Otherwise, during the insertion process of Xk (together with the eventual units added during the insertion process), the new π(λ) will also be recorded in MDB, while the main loop execution will continue. State space pruning as an effect of PBA, in case of some Xk insertion failure is depicted in the Figure 6 below.

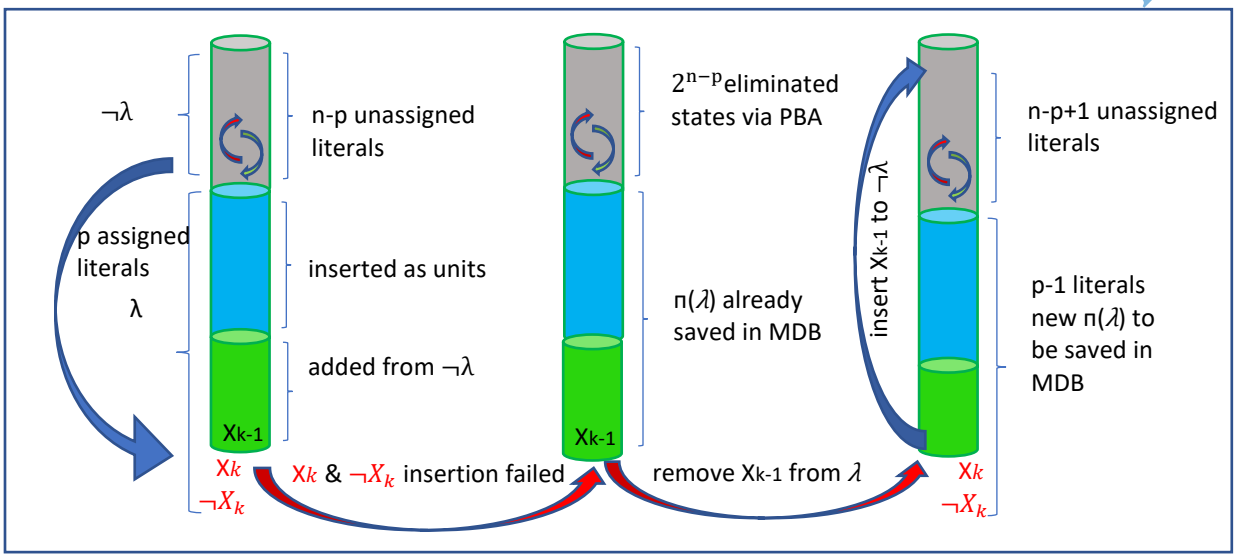

Figure 6 – PBA of 2^{n-p} states after some Xk insertion failure, with a σ(λ) = p elements

Partial states are saved (SaveState) once before the execution of the GetAllUnits block starts and, if successful, they will be one more time saved at the end. While the first SaveState execution has the double role of saving and validating the unicity of that π(λ), the second execution has only the role of saving that π(λ) if it is unique. In the first case if π(λ) is not unique then the entire main loop execution is considered as failed per the logic described above, while the second is neutral from this point of view due to the certainty that the particular new state did not occur in the past. What is not that obvious is the fact that some intermediate portions of π(λ) are not saved per say, and that is because of the units occurring during the Xk insertion process. The example below will make this fact clearer:

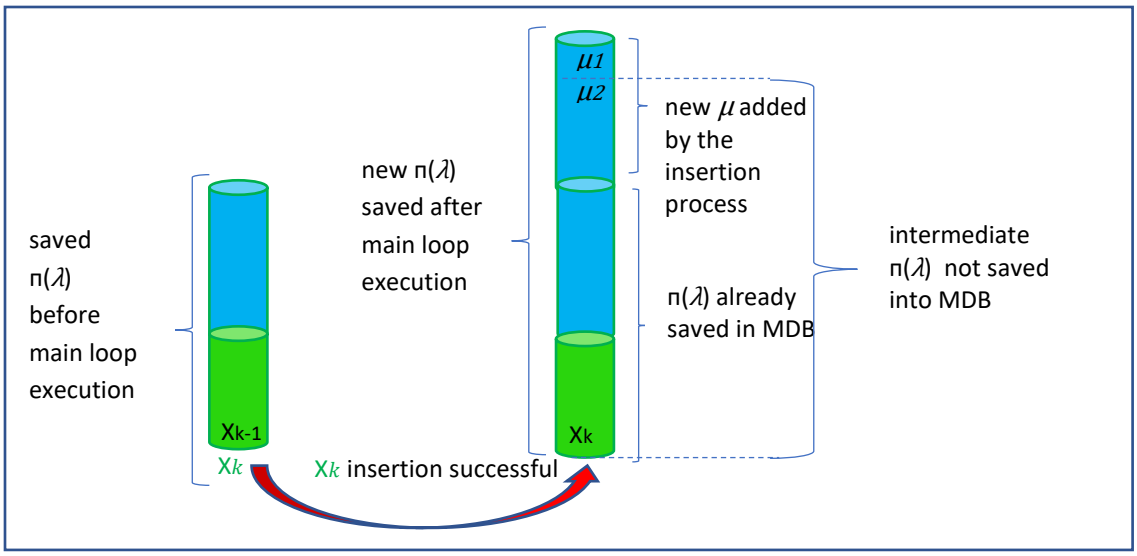

Figure 6.1 intermediate π(λ) not saved into MDB

The effect of this missing intermediate π(λ) is that, while λ is being rewind due to some Xk insertion failures, some main loop runs will be skipped because the particular π(λ) is already in MDB, while others will be executed because the intermediate π(λ) are not yet part of MDB, thus bringing them into MDB during the literals' insertion process.

The concept of μ has been used until now while no explanations about the concept or its dynamics have been given. So, any so-called collision is based on μ search/finding while the entire algorithm is based on it. What if μ do not exist in some Φ, is that even possible? Would $\mathcal{A}$ work in this case? To answer these questions, consider that μ exist due to the fact that for any pair of literals [X, Y] where each of them is set to True, the opposite pair [$\neg X$, $\neg Y$] if it exists in any clause, their third literal from these clauses becomes a μ. However, if absolutely no μ exists, it means that, wherever these pairs occur, at least one of the two literals is set to True as in [$\neg X$, $\boldsymbol{Y}$] or [X, $\neg Y$] thus making every single clause True and so, Φ becomes a tautology. Therefore, having μ will happen in any Φ or else Φ is a trivial one.

8. COMPLEXITY OF A NON-SAT Φ RUN

As described above, PBA is based on the retraction of literals and that may resemble to the backtracking technique while, naturally, this question will arise: "isn't PBA the same as backtracking in terms of complexity?". Because if it is, then there is no point in developing the algorithm any further since its order of complexity is exponential. As a proof that the two techniques are different in most aspects including complexity order, a number of differences are highlighted further.

While backtracking removes the last value which does not fit to the solution, flips the previous, and then continues forward by trying the value removed, PBA saves and refutes all partial states in future cycles since the logic behind yields that these refuted states cannot be part of any SAT solution as mentioned in Chapter 7. It removes the previous value and retries to fit the current into a smaller partial solution. By doing that, one or more branches of possibilities (combinations) will be pruned. For any p-elements state, $2^{n\text{-p}}$ combinations are cut from future runs/combinations. The sum of all pruned combinations reduces the exponential complexity, down to polynomial size, while reasons follow below.

$\mathcal{A}$ uses two complementary list-like structures (λ & ¬λ) which, during Φ execution, the two combined contain all literals. At the beginning of the run, since λ is empty, ¬λ contains all possible 2^n states because no literal has a value assigned yet so $\mathcal{A}$ will pick any of the literals, on any polarity and try to build λ up to n elements. In case conflicts occur so that λ goes back to empty state, $\mathcal{A}$ will try again using a new literal or the other polarity of the same, and so on. Each time λ goes back to empty state, a number of partial states are saved in MDB, and so, once all literals on both polarities will lead to empty state, Φ is considered as NON-SAT, and that is because there is no literal and polarity $\mathcal{A}$ can start with and fill λ up to *n* non-conflictual elements. Reaching the point where σ(λ)=0 for $2n$ times (once for each literal for each polarity) due to conflicts, will prove that Φ is NON-SAT. In this case, MDB will contain all unique π(λ) encountered during all main loop cycles, thus indirectly representing all 2^n states. The question is how many RunEngine cycles would lead to an empty λ though. Otherwise said: how many such cycles would bring any two conflicting literals as the only ones in λ? To answer this, answering this question is firstly required: what are the minimum conflicting literals in a NON-SAT Φ? The minimum number of conflicting literals is $\binom{n}{3}$ for any NON-SAT Φ – which means that there are at least three pairs of literals for any buildup of σ(λ)<=n-3, that will lead to an empty λ. In the case of σ(¬λ) = 1 or σ(¬λ) = 2 no conflicts can arise anymore. To prove this, assume that σ(¬λ) < 3 while looking at the fact that λ at this point contains all non-conflictual, validated n-1 or n-2 literals. Since every clause of Φ contains exactly 3 unique elements per R1 to R4, the 1 or 2 remaining cannot produce a conflict since they cannot represent a clause, while all clauses have already been validated (SAT) via GetAllUnits block. What if Φ is a one-solution formula (that is when σ(β) = n), wouldn't that situation of σ(¬λ) = 1 or 2 bring a conflict in case the one or two remaining can be chosen with the wrong polarity? In this case at the latest during the step (state) where σ (λ) = n-3, the other literals will be brought into λ as units, so σ(¬λ) will never be smaller than 3 elements after a complete loop, or, throughout the flip-to-fit process, they will be set and inserted to the right polarity. As conclusion so far, <u>once σ(¬λ) < 3 during any main loop cycle, Φ is SAT</u>. In reverse, <u>if Φ is NON-SAT, then there are at least 3 literals conflicting for any single and unique X_s used as a starting point</u>. In order for any two of them to reach the state where they are the only ones in the λ stack, one has to be positioned at the top (Xj) having its polarity set since it belongs to λ already, while the other one should be the one to be inserted (X_k) and, during the flip-to-fit process, it removes all the rest of the literals in-between. As an example, the Figure 7 below presents the action of moving literals from λ to ¬λ, (the **blue** downslopes). In this run $\mathcal{A}$ started with $\boldsymbol{X_1}$.

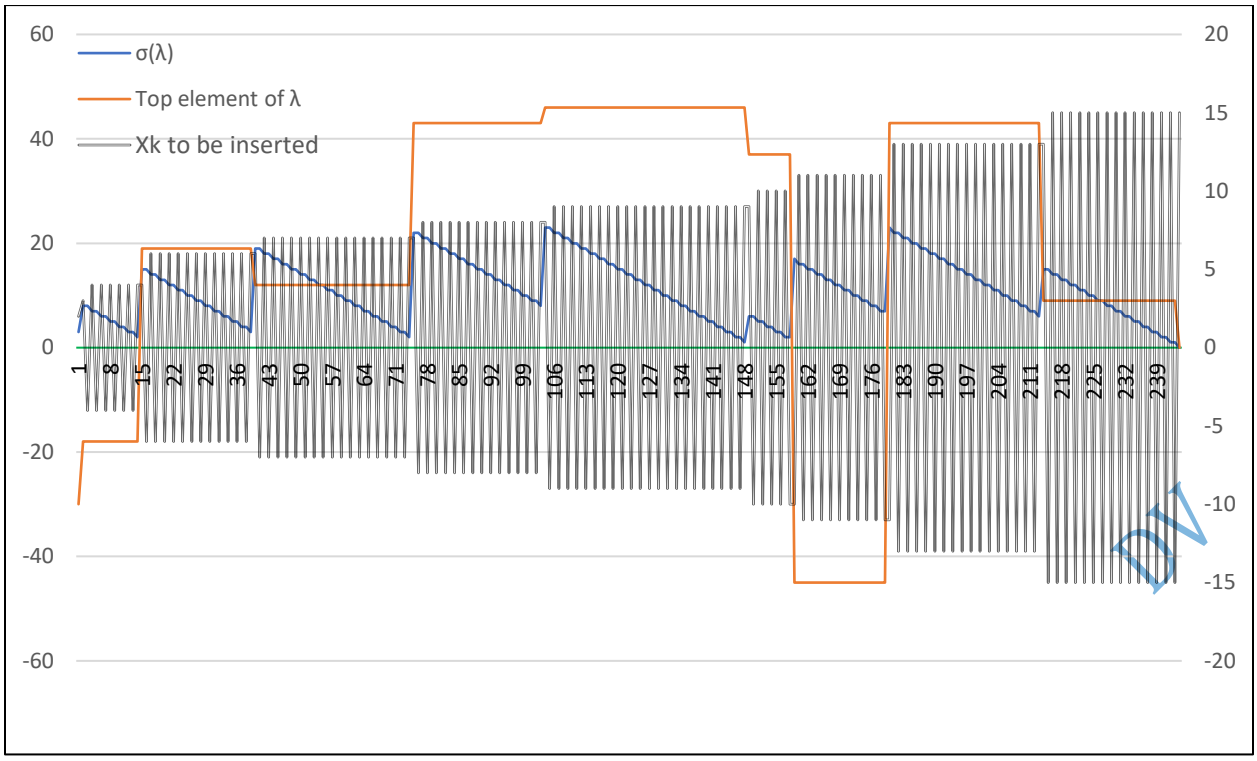

Figure 7 – $\mathcal{A}$ reaching the first σ(λ) = 0

Figure 7 depicts the first "touchdown" of σ(λ)=0 in an n=50 NON-SAT Φ, with every X_k that encountered a collision being flipped on its polarities while elements of λ are removed once every two cycles. This run was the first among the total of 100 and it reached σ(λ)=0 where λ reached the empty state in 244 cycles of the main loop. The Y-left axis refers to σ(λ) (**blue line**) and to the value of λ top-most literal (X_j - **orange line**), while on the Y-right axis is the value (index) of $\boldsymbol{X_k}$ (**black line**). It is visible that, from cycle to cycle of the main loop, $\boldsymbol{X_k}$ changes its polarity and, after covering both polarities, λ decreases by one element. The run was done using the C# implementation version of $\mathcal{A}$. The conflicting literals were $\boldsymbol{X_9}$ and $\boldsymbol{X_{15}}$ while this particular formula was resolved as NON-SAT after 33963 main loop cycles, which corresponds to a raw measured complexity of $n^{4.6669}$. One thing to mention about the example above: $\boldsymbol{X_9}$ was added on top of λ as a μ by the insertion (flip-to-fit) of $\boldsymbol{X_{13}}$, after the latter was "accepted" into λ as its last element. The conflictual $\boldsymbol{X_{15}}$ came next in line to be inserted into λ from ¬λ, thus shrinking λ down to empty state due to conflicts and/or partial states repetitions/denials.

There is a question emerging: if there are three or more conflicting literals, then why not combine any two and reach the NON-SAT conclusion, right from the start and solve it (identify them) in a complexity of O(n^2)? And the answer yields: if in some minimal and unlikely number of NON-SAT Φ this might work, generally it would not, unless the partial states leading to these conflicts are built. Worse, if Φ is SAT, this heuristic may lead to a false-negative result. It is needed to reach the NON-SAT resolution by building the MDB via recording partial states, because without this, the 2^n combinations will not be covered, therefore the proof of NON-SAT would not be certain.

Before going further, a brief recap for the numbers above, and their effects:

- Once σ(λ) > n-3 elements, Φ is SAT

- The other way around, if Φ is NON-SAT, then σ(λ) cannot be larger than n-3 literals, in any step during $\mathcal{A}$ run
- For any σ(λ) > 0 there are at least 3 conflicting literals, thus making O($\binom{n}{3}$) distinct literals combinations that will lead to an empty λ

The way λ and ¬λ are built and used is not an implementation choice, but part of $\mathcal{A}$ requirements and that needs some attention. λ is constructed from its initial empty state, by adding new literals (assign the Truth value to one of their polarities) picked/removed from ¬λ. Any new literal taken from ¬λ will be inserted at the bottom of λ (list-like add), while all new μ are inserted at the top (stack-like push) as shown in Figure 4. The reasons it is done this way yield: if X_k to be inserted encounters a conflict, then the first literal removed would be the one at the very bottom of the list. The effect is that one or more clauses may become NON-SAT. The other way around, if the μ would be at the end of the list, they would be the ones to be removed first and, besides the fact that they will certainly bring a number of clauses in a NON-SAT state (because a μ by definition is paired with other two literals set to False), they would break up the build of λ as well, because at least one of the inserted X_k that produced them should be changed, and so, at least one of the two literals which produced that μ must also be removed or reverted. And that becomes subject to literals backtrack, which $\mathcal{A}$ actually is conceived to avoid.

One other technical aspect of $\mathcal{A}$ that is worth mentioning, is the fact that any literal removed from λ, must be placed at the bottom of ¬λ so its "turn" to be inserted back to λ comes last, from that point on, unless it is selected earlier as a μ. Doing that, "starvation" of literals will not occur. One other reason is to preserve the circularity of $\mathcal{A}$ plus the consistency in λ and MDB buildups.

For any NON-SAT Φ, at the end of the run, all 2^n must somehow be represented in MDB. In fact, not all 2^n states need to be physically saved into MDB. It is sufficient to have all n literals representation, on both polarities, for the states where σ (λ) =1 in MDB, to prove that Φ is NON-SAT. This means that each time λ reaches the empty state while the starting literal of that run was already added to MDB, that literal will no longer be used as a starting literal. Therefore, any new ɲ(λ) with a σ(λ) = 1 is added to MDB, represents and eliminates up to 2^{n-1} states whenever λ reaches the empty state from that point ahead. And this needs to happen $2n$ times (once for every polarity of each literal), which actually will cover for up to $n2^n$ states. But $n2^n > 2^n$ so why not simply 2^n? In fact, the sum of all refuted states is indeed 2^n and not larger, because the rest are just reoccurred states from previous cycles. For instance, if λ at some point contains literals [1,2,3] while a collision occurs so this partial state is being saved into MDB, then whenever the other combinations will occur ([1,3,2], [2,1,3],[2,3,1],[3,1,2],[3,2,1]) they will actually represent the same partial state as the [1,2,3] therefore engine's execution for the cycles containing them, will be skipped.

SaveState is the process that brings new states into MDB, and the number of cycles to get to a full representation is the following: Assume that λ contains n-1 literals while the very last one to be inserted, produces conflicts on both its polarities. Although this case together with σ (λ) = n-2 is not possible to occur as explained before, they still are to be used for the sake of complexity computation. So back to one unassigned literal left as ¬λ member. In this situation, after flipping on its polarities (two trials, meaning two main loop cycles) and both winding up in conflicts, two partial states (one for each polarity) will be saved into MDB as non-acceptable, where σ(λ) = n-1. This can happen $2n$ ways since there only are n distinct literals with two polarities each, strictly considering that all of them reach that stage once. Throughout this rotation of literals, every single one will be part of λ at some point otherwise λ will not contain n-1 elements. This will be called "Stage n-1" or Sn-1 for fast reference purposes. The same, having left two unassigned literals in ¬λ (so σ (λ) = n-2) will take n literals ($2n$ main loop iterations) to cover for all n literals once, wrt σ (λ) and that is Sn-2. Going forward and progress downwards from Sn-3 to S1 the logic holds the same, in spite the fact that one might argue that at every single Sk, there are 2^{n-1} possible combinations

to reach S_k again for the same literal, which is true, and that only makes it easier to reach that certain stage, once, for every literal. Although important, the point here is not how many times/ways a certain S_k can be achieved for some literal, but rather the number of cycles it takes to achieve a full range of literals on that S_k, plus the number of states this process will eliminate. The way PBA is applied to λ due to insertion of a new literal, while eliminating elements of λ, several (descending) stages are covered for that literal. This can be observed in Figure 6 (blue line and black line). The longest descending such path for any X_k is n-3, since λ will never contain more than n-3 elements and be NON-SAT at the same time. Every time λ becomes empty, a new literal will be assigned as a starting literal, therefore the case of σ (λ) = 1 (S_1 that is) will be achieved $2n$ times for any NON-SAT Φ. The significance of all $2n$ occurrences of σ(λ) = 1 (every literal being used on both polarities as a starting literal), yields a NON-SAT Φ since λ cannot start with any of them, on any polarity, and not reaching the empty state via either collision or reaching an already recorded π(λ).

As a conclusion, for every start from σ(λ)=0, adding one literal, there are at least 3 literals, which combined, lead back to σ(λ)=0 state, therefore there are at least 3 pairs (X_j, X_k) of such literals for any starting literal, for any given NON-SAT Φ, in order to reach that state. In spite the fact that four combinations are possible for each pair, only three of them are needed to bring λ to an empty state, since the one already part of λ is fixed and not going to be flipped. The top-most literal in question can be either a literal selected and inserted from ¬λ or a μ brought there by one of the literals inserted from ¬λ. The process that brought the top-most literal has no importance. Its role has.

Consider X_k as the one both-sides conflicting literal selected to be inserted, and, while X_j is fixed in terms of its polarity, X_k polarity will flip-to-fit each main loop cycle. There are a number of things to highlight:

1. If X_j is part of λ then X_k is not yet because they conflict and cannot both belong to λ at the same time
2. If X_j is part of λ and it is not on top, before λ gets empty, it will be removed by the insertion process of X_k, as X_k is artificially selected for insertion, unless it is "bubbled up". Although the "bubble up" process to bring X_j on top of λ is only an optimization (therefore not mandatory), tests showed that it could reduce the complexity by a factor of up to $0.5n$ and this is one of the reasons it is mentioned
3. All n literals (excluding the pure literals), on all polarities, are the "X_j" (one of their polarities conflicts with both polarities of some X_k from ¬λ) for any combination of the 2^n possible, otherwise Φ has a SAT solution.
4. As a direct implication of number 1, at any point during main loop execution, λ contains only non-conflicting literals, while all the rest including the conflicting ones wrt any current $\pi(\lambda)$, belong to ¬λ

The points above are highlighted in order to bring up the following conclusions: by design, $\mathcal{A}$ assumes that X_j (on one polarity at least) is conflicting with X_k (on both polarities) then X_j has to be on top of λ, while Xk is the one to be inserted first in line, and that is a mandatory condition in order to obtain an empty λ based on these two literals. Therefore, at any point when the literal on top is X_j for some literal X_k, λ will reach the empty state via the insertion process of X_k. During the process, a number of partial states will be retained in MDB, thus shrinking the search space for future main loop cycles since any $\pi(\lambda)$ of size p saved in MDB will eliminate 2^{n-p} states.

A summary of the most significant conclusions so far, for any NON-SAT Φ, while excluding the pure literals since they cannot conflict with any other literal (neutral) so they cannot be part of any restriction rule, nor the clauses they are part of:

1. There are at least three conflicting literals on both polarities, as part of ¬λ, per any λ buildup, otherwise Φ has a SAT solution.

2. The conflicting literals cannot coexist in λ since they refute one another
3. In case σ(λ)=n-3 while one of three unassigned literals does not collide with at least one literal from λ on both polarities, or with at least two literals from λ on different polarity each, then that one would be accepted as element number n-2 into λ, thus making Φ, SAT
4. Whenever any literal Xk is brought/selected from ¬λ, during the insertion process, some combination of the following will occur:
    a) each time if Xk creates conflicts on both polarities, the bottom literal of λ will be moved to ¬λ
    b) each time a partial state reoccurs on both polarities of Xk, the bottom literal of λ will be moved to ¬λ
    c) the number of μ added to λ is >= 0 for any Xk added to λ
    d) Xk eventually fits into λ with one of its polarities, or
    e) λ reaches the empty state
5. The only rules of selecting the next literal Xk from ¬λ in order to be inserted into λ yield:
    a) no polarity of Xk can already be part of λ
    b) its index is the next available after the previous Xk to ensure rotation and non-starvation of literals' use
6. Polarities and literals' indexes have no meaning to $\mathcal{A}$, other than the ones mentioned at 5. It is agnostic to both
7. The order of literals in λ has no meaning since the contained literals, in any order, represent the same partial state. As a side effect, the "bubble up" process will not change/affect $\pi(\lambda)$.
8. Building λ from empty state, in order to contain/cover any combination of 3 literals on all the polarities, will take at most 7 $\binom{n}{3}$ main loop iterations where GetAllUnits block is being executed, while for any already visited/recorded $\pi(\lambda)$ that block execution is discarded. This number of iterations is true only in theory as absolute value because in practice, whenever one single literal is successfully added to any $\pi(\lambda)$ with a σ (λ) = k, it will add extra $\binom{k+1}{3}$ - $\binom{k}{3}$ = (k - 1) k /2 new combinations, or more when μ are also added on top of λ, while the sum for unique additions yields $\sum_{k=1}^{n-2} k(k-1)/2$ = n(n - 1)(n - 2)/6.
9. Among all possible 7 $\binom{n}{3}$ combinations, at least one will lead back to an empty λ via flip-to-fit process, while producing them is in $O(n^3)$
10. Using 9, reaching $2n$ times an empty λ is in $O(n^4)$ in worst case
11. **From 10, exhaustively covering for all 2^n combinations possible, is in $O(\text{main loop}) * O(n^4) = O(n^6)$**
12. **As a side effect of 11, finding a solution for a SAT Φ, is in $O(n^6)$**

For further detail and exemplification of the affirmations above, the case presented in Figure 6 will be used. In this case, the fixed, top-most literal (Xj) was X_9, and which was a μ brought by the insertion (flip-to-fit) of X_{13} while X_{15} (Xk), was next in line to be added from ¬λ, and it was conflicting with X_9. As such, during the flip-to-fit process, the bottom-most literal has been removed until X_9 was removed too, thus bringing λ to an empty state. As an observation, it is visible that the positive polarity of X_9 would bring conflicts whenever combined with X_{15} or $\neg X_{15}$ and, as such, X_9 on positive polarity does not need to be used anymore as the first element (starting literal) on an empty state λ. However, this does not say anything about $\neg X_9$ while the fact that X_9 will no longer be used a as starting literal in the future is only an optimization. And as mentioned earlier, there are plenty of sound optimizations to bring to $\mathcal{A}$, but these are not the subject of this paper. Now referring to $\neg X_9$, another question needs to be raised: what if, in a similar case as the one studied above, one simply flip X_9 to $\neg X_9$ and, in combination with both polarities of X_{15}, conflicts still occur on both polarities of the latter. Would not this be proof enough that

Φ is NON-SAT? No. Not necessarily. In order to claim that there is no solution to Φ, given the current scenario, $\mathcal{A}$ needs to prove it by building the correspondent MDB and eliminate partial states from bottom-up. And that is because the case may occur when, $\neg\boldsymbol{X_9}$ is accepted aside of some polarity of $\boldsymbol{X_{15}}$ without conflicts, largely depends on factors such as the rest of the content of λ. At this point, having only two literals involved which produce conflicts (or reaching an already visited π(λ)) at that stage, does not mean that among the other 2^{n-2} remaining untapped combinations cannot be one where no conflicts occur between the two literals in question. So, no shortcut/optimization here. The study case presented in this document would help to a better understanding on how $\mathcal{A}$ solves such problem, while making it a bit easier for me to explain the details. Below is included the data on all 100 starting literals runs – it is a n=50 inputs, NON-SAT Φ. Notice the fact that, although some pairs [λ-top, Xk] occur several times such as [$\boldsymbol{X_{42}}$, $\boldsymbol{X_{27}}$], the rest are different conflicting pairs, reaching the conditional positions of [λ -top & literal to insert]. In fact, this study case contains 7 such pairs. A triple occurrence of such pair has been highlighted for analysis. The pair [$\neg\boldsymbol{X_{19}}$, $\boldsymbol{X_{20}}$] occurred in O(1), O(n) and O(n^2) in terms of main loop iterations, until λ reached the empty state. This means that, for the correspondent starting literals, this pair was either the only one – but it has justified that this cannot be the case – or the fastest to occur. Just making sure there is no misunderstanding here: for each run (new starting literal), MDB is being conserved, so MDB obtained from previous runs are used by the current one – not to be confused with rerunning $\mathcal{A}$ using different starting literals from scratch on the same Φ, case where MDB starts empty for each starting literal. For a number of three starting literals ($\neg\boldsymbol{X_5}$, $\neg\boldsymbol{X_{11}}$, $\neg\boldsymbol{X_{19}}$), the pairs [$\neg\boldsymbol{X_{19}}$, $\neg\boldsymbol{X_{20}}$] and [$\neg\boldsymbol{X_{19}}$, $\boldsymbol{X_{20}}$] are leading to an empty λ and, at the same time, the pairs [$\boldsymbol{X_{20}}$, $\neg\boldsymbol{X_{19}}$] and [$\boldsymbol{X_{20}}$, $\boldsymbol{X_{19}}$] are leading to empty λ when ($\neg\boldsymbol{X_{17}}$) is used as a starting literal. Now to bring the point above, back to this example: while two literals can produce an empty λ in certain conditions (starting literals) it does not mean anything else but that. Otherwise, the pair [$\boldsymbol{X_{19}}$, $\boldsymbol{X_{20}}$] would have produced more other among all other starting literals. The main idea here is the fact that for $\mathcal{A}$ to prove a NON-SAT Φ, all literals on all polarities will be used as starting literals, otherwise $\mathcal{A}$ has no deterministic proof that Φ is NON-SAT.

Starting Literal	λ Top literal (Xj)	Conflicting literals on both polarities (Xk)	Main Loop Cycles
1	9	15	244
2	-20	46	1587
3	45	19	160
4	32	41	373
5	-8	46	542
6	42	27	1056
7	-3	13	92
8	13	28	1012
9	42	27	876
10	-8	27	241
11	-20	35	410
12	32	24	1050
13	-8	24	169
14	-8	19	27
15	-37	17	9
16	32	26	827
17	7	27	2333
18	42	23	45
19	18	37	269
20	-5	24	46
21	13	28	821

22	-1	44	259
23	44	37	164
24	-1	27	754
25	-46	3	330
26	-29	42	217
27	18	7	1130
28	9	12	515
29	-3	35	43
30	-46	33	12
31	50	34	17
32	48	20	543
33	-14	10	368
34	42	37	14
35	-44	9	371
36	-23	44	124
37	32	44	87
38	19	40	9
39	-3	41	7
40	-21	20	390
41	34	37	518
42	42	43	2
43	48	20	362
44	-8	24	448
45	45	46	2
46	-42	48	7
47	31	34	472
48	48	49	2
49	-20	41	537
50	50	1	2
-1	-9	7	55
-2	-2	3	2
-3	-8	34	489
-4	-35	12	97
-5	-19	20	259
-6	-12	9	16
-7	47	10	36
-8	-8	9	2
-9	-9	10	2
-10	-7	43	465
-11	-19	20	2376
-12	-12	13	2
-13	9	44	471
-14	-1	39	367
-15	-2	18	734
-16	-8	19	48
-17	20	19	7
-18	44	39	324
-19	-19	20	2
-20	-20	21	2

-21	-23	27	45
-22	-2	38	202
-23	-23	24	2
-24	-8	36	941
-25	-20	17	613
-26	-8	43	211
-27	-16	39	146
-28	-39	37	709
-29	39	46	221
-30	-48	10	413
-31	42	39	65
-32	42	16	480
-33	24	40	79
-34	8	9	292
-35	12	42	104
-36	-36	37	2
-37	-3	24	588
-38	-38	39	2
-39	42	10	288
-40	-3	9	272
-41	34	6	229
-42	-14	46	30
-43	-8	49	65
-44	39	11	281
-45	-42	48	12
-46	-46	47	2
-47	34	26	1110
-48	42	9	101
-49	19	6	113
-50	-1	47	694

The run in the example above did 33963 cycles of the main loop which corresponds to a measured complexity of $n^{2.66}$ which multiplied by n^2 per each cycle, yields a measured complexity of $n^{4.66}$ which is much faster than O(n^6). In reality, even $n^{4.66}$ is not quite accurate because many of the cycles did not run since the $\pi(\lambda)$ of those cycles were duplicates already saved in MDB as explained above, and these cycles only take O(n) in worst case. But of course, this is only one example out of infinitely many and cannot be used for any proof, but only as an example for how $\mathcal{A}$ works.

9. ZERO KNOWLEDGE PROOF (ZKP)

For tests and benchmarking, all formulae from https://www.cs.ubc.ca/~hoos/SATLIB/benchm.html have been used, plus a number of complete 3-SAT formulae generated from https://toughsat.appspot.com/ In the chapter "Benchmarks, Test & Results" below, a suite of 42000 different Φ of size n=100 and various $\sigma(m)$, with $\sigma(\beta)$ of 10, 30, 50, 70 & 90 is mentioned. Each Φ was solved 200 times, in order to use all literals on all polarities as starting literals, without conserving the data from the MDB from one run (starting literal) to another. That made 8.400.000 runs with a maximum measured complexity of $n^{4.3}$ and an average of $n^{3.03}$, therefore all polynomial runs. Also, runs for 48 $\sigma(\beta)$ =n with n=50, 100 & 200 and various $\sigma(m)$, for each input as starting literal, for each polarity, have been executed. The same, 3000 NON-SAT Φ of different n between 50 and 100 ran in similar way, and all were done in polynomial complexity of under n^6. Altogether summed around 10.000.000 different runs in poly time/complexity, SAT and NON-SAT.

Ignoring the proofs from the chapters above and just looking at the results mentioned in this chapter, the chances of "guessing" a solution in polynomial complexity, would be $1/(2^{10.000.000})$, per ZKP fundamentals. Even ignoring the bulk of 8.400.000 runs and simply use just the 12 one-solution formulae of size n=200, the chance of guessing that unique solution in the space of 2^{200}, and that for each individual input on each polarity (so 400 runs for each such Φ), plus guessing all of them in polynomial time, is very close to null.

However, this infinitesimally small chance of guessing a correct response only produces a statistical guarantee that $\mathcal{A}$ is polynomial, and absolutely no guarantee that there is no Φ out of infinitely many, that would contradict $\mathcal{A}$. Therefore, using ZKP, the fact that $\mathcal{A}$ is polynomial becomes a conjecture, so the proof before ZKP is still needed in order to formally settle the dilemma of P vs NP.

10. BENCHMARKS, TESTS AND RESULTS

The benchmarks used are quite various and rich, for both SAT and NON-SAT Φ. As mentioned, the majority were downloaded from: https://www.cs.ubc.ca/~hoos/SATLIB/benchm.html and many other were produced by the public 3-SAT engine: https://toughsat.appspot.com

From the published ones, all the known 3-SAT from 20 to 400 inputs were solved for all individual input on both polarities, including the ones that only had one solution only in the space of 2^n possible paths. All known NON-SAT Φ from 50 to 100 inputs ran in the same manner. The total number of runs, summed, was greater than 10.000.000 and all were executed on 2.8 GHz laptop, so readers could do the same if they chose to, since this kind of CPU performance most of the current regular desktops or laptops are fit to achieve. As results, among the SAT Φ the highest measured complexity was $n^{4.3}$ and $\Theta(n^{3.12})$ while among the NON-SAT ones was $n^{5.3}$ and $\Theta(n^{5.07})$. The largest Φ used was built as n=4000 inputs and m=9999 clauses in $O(n^{2.79})$. Among all the runs, the current $\mathcal{A}$ implementation had no failure in the sense of crashing or returning a false result, or worse, never finish. Although, if the latter would happen, no one would know that this is the case…

11. OPTIMIZATIONS

While $\mathcal{A}$ fundamentals will be preserved, a large number of optimizations can be applied. Here is a very short and non-exhaustive list that would make this algorithm perform better of even far better while mentioning that far more than this short list can be implemented so its performance can be dramatically increased:

ALGORITHM

- Boolean resolution – applying this technique, the number of clauses may shrink by a lot. In case of a NON-SAT Φ, it can lead very fast to the right outcome, by far faster than the way $\mathcal{A}$ performs it
- Parallelization – Since starting with any literal, on any polarity, will reach the outcome with a different complexity/time, this will give the fastest result if all 2n possible starting literals will be used in parallel. When running $\mathcal{A}$ implementation using the "exh" switch, the report at the end will mention the fastest and the slowest input plus the total complexity for all inputs. This should give an idea about the possible differences
- Pure literals – this will eliminate upfront all clauses containing such literals
- Bubble up – This optimization is specific to $\mathcal{A}$ and will move one literal on the top of λ anytime that literal creates a collision with both X_k and $\neg Xk$. The experiments show that this can bring up to 0.5n acceleration in a NON-SAT Φ, while making no difference in a SAT one
- Using all the μ coming from GetOppUnits function – it is used in the current implementation
- Mark any X_k as "used" as a starting literal once λ is containing just it – this is implemented in the current code – will accelerate up to 0.3n
- Standard techniques such as BCP (Boolean Constraints Propagation)

SOFTWARE

- Possibly using other language than C# would make the execution faster from the CPU point of view
- Depending on the purpose/scenarios, implementing a task-specific version of $\mathcal{A}$ while conserving its fundamentals, may lead to an application-specific performance improvement

HARDWARE

- An ASIC – since $\mathcal{A}$ implementation only has around 100 lines of executable code plus the fact that the memory size is in the same ballpark with its complexity, an ASIC should be fairly easy to build, and it could do the job way faster than a regular, non-dedicated computer.
- Since $\mathcal{A}$ uses no math computations at all (besides comparisons), memory shifting is the main time-consuming activity during a run. Having a faster or an Overclocked memory is going to accelerate its execution

12. C# IMPLEMENTATION

The implementation copies almost exactly the algorithm's specifics described in this paper. There are very few implementation specifics to be mentioned such as the following:

- The negative polarities indexes ($\neg Xk$) are represented by positive numbers on the form of k + n. This specific is used for indexing purposes. This way the -74 from $\neg X74$ becomes 174 if n=100.
- There are several switches to run Φ:
  - cnf_file_name.cnf – runs one cnf file containing a Φ in DIMACS format
  - all – runs formulae from all files ending with extension **.cnf** from the current folder
  - exh – exhaustive run, meaning that will rerun for all inputs and all polarities (so 2n runs)
  - stp – in case of an exh run, if the user expects that the Φ is sat and with one of the inputs the result is NON-SAT (wrong result), then the run will stop on the NON-SAT output
  - sol – after each run will display on the screen the solution found. Only the positive values/indexes are displayed
  - rec – logging in a different file 3 statistic columns in order: size of λ, top of λ, last element of λ – it is used to see how λ fluctuates from start to finish. In a SAT Φ, only the first column makes sense (such as in Figure 5), while for a NON-SAT will give you the insights of λ dropping to empty as in Figure 7. The name of the file is built as in: "run_" + `currentFormula_filename` + DateTime.Now.Millisecond + ".csv" (ex: **run_aim-100-1_6-yes1-1.cnf875.csv**)

Some examples:

- NanoSat.exe abc.cnf –will run NanoSat.exe on abc.cnf file
- NanoSat.exe all – run NanoSat.exe on all formulae files existing in current directory
- NanoSat.exe abc.cnf exh sol stp – run NanoSat.exe on abc.cnf file using all literals on both polarities as inputs while printing the solution for each of the inputs at the end of each run. The switch stp is being used to stop the run if the result is NON-SAT
- NanoSat.exe abc.cnf rec – run NanoSat.exe and record/log extra statistics
- NanoSat.exe all exh sol stp

After each run of an individual Φ, by default some statistics including a SAT solution if the case, are logged in csv format, with the file name formed as: currentFormula_filename + DateTime.Now.Day + "_" + DateTime.Now.Month + "_" + DateTime.Now.Year + "_" + DateTime.Now.Hour + "_" + DateTime.Now.Minute + "_" + DateTime.Now.Second + ".csv" + "_results.csv". Example: **aim-100-1_6-yes1-1.cnf31_5_2021_19_45_42_results.csv**

In case you're running "bulk" – switch "all" the name of the log file will look like this:

results_all_run_of_30_5_2021_9_32_31.csv

Also, the entire solution for each Φ will be saved in a new directory. Each individual Φ will be saved as .txt where the result is sorted so that the literals set to true are on the first column, and also sorted in absolute value in ascending order, so the reader can just look at the first column and visually verify that there is no conflicting literal in it. Example: **aim-50-1_6-yes1-1.cnf.txt**

Very important: Each Φ file needs to obey R1 to R5 rules in order for the current implementation to perform successfully.

To compile the code, from the cmd line you need .NET 3.5 or higher. The command to build the executable is:

c:\windows\Microsoft.NET\Framework\v3.5\bin\csc.exe /out: NanoSat.exe NanoSat.cs

13. CONCLUSIONS

In spite of the relatively high order of complexity, the algorithm presented in this document together with its implementation, leads to a polynomial upper bound complexity for all problems in the NP space and, since any NP problem can be reduced to 3-SAT (which is an NP-C problem) in polynomial time/complexity as per Cook-Levin theorem, therefore P = NP.

REFERENCES

1. *Cook, Stephen (1971). "The complexity of theorem proving procedures". Proceedings of the*
2. ^ Jump up to:[a] [b] *Karp, Richard M. (1972). "Reducibility Among Combinatorial Problems" (PDF). In Raymond E. Miller; James W. Thatcher (eds.). Complexity of Computer Computations. New York: Plenum. pp. 85–103. ISBN 0-306-30707-3.*
3. ^ *T. P. Baker; J. Gill; R. Solovay (1975). "Relativizations of the P = NP question". SIAM Journal on Computing. **4** (4): 431–442. doi:10.1137/0204037.*
4. ^ *Dekhtiar, M. (1969). "On the impossibility of eliminating exhaustive search in computing a function relative to its graph". Proceedings of the USSR Academy of Sciences (in Russian). **14**: 1146–1148.*
5. ^ *Levin, Leonid (1973). "Универсальные задачи перебора" [Universal search problems].*
6. ^ Jump up to:[a] [b] *Garey, Michael R.; David S. Johnson (1979). Computers and Intractability: A Guide to the Theory of NP-Completeness. W. H. Freeman. ISBN 0-7167-1045-5.*
7. *Ladner, R. E. (1975). "On the structure of polynomial time reducibility"*
8. *William Gasarch (June 2002). "The P=?NP poll"*
9. Wigderson, Avi. "P, NP and mathematics – a computational complexity perspective"
10. Davis, Martin; Putnam, Hilary (1960). "A Computing Procedure for Quantification Theory"