Contents

1 Overview **3**

2 SAT Algorithms **4**
2.1 DPLL and non-determinism 4
2.2 GSAT and random-restart local search 5
2.3 WalkSAT and random walks 7
2.4 SASAT and adaptive probability 8
2.5 Genetic Algorithms and refinements 9

3 SAT-Related Problems **12**
3.1 CSPs and Cycle-Cutset induced search 12
3.2 Reduction and Bayesian Networks 13
3.3 Max-SAT and approximated algorithms 15
3.4 Model Counting and random sampling 17

4 Probability Logics **20**
4.1 Towards a new formalism 20
4.2 Probability Functions 21
4.3 PSAT 22
4.4 Probability Operators 24
4.5 Subjective Logic 25

5 Implementation **28**
5.1 Classes and Functions 28
5.2 Preprocessing 34
5.2.1 Prefix Notation 34
5.2.2 CNF 37
5.2.3 Random instances 40
5.3 Unit Propagation and Pure Literals Rule 41
5.4 SAT Algorithms 44
5.4.1 GSAT 44

5.4.2 WalkSAT 44
5.4.3 SASAT 46
5.4.4 Genetic SLS 48
5.5 Max-SAT Algorithms 51
5.5.1 Johnson's Algorithm 51
5.5.2 Bidirectional Bounds Refinement 53
5.5.3 Look-Ahead SLS 54
5.6 Model Counting Algorithms 55
5.6.1 ExactCount 55
5.6.2 ApproxCount 56
5.6.3 SampleCount 58
5.7 Comparison 60

Bibliography **61**

Chapter 1

Overview

Is it possible to combine probability theory with logics? This might appear as a strange question. Logic is a field that more than others seems far from a probabilistic approach. Anyway, getting a little closer, we can observe how many points of contact exist between them, and how many must be already found. In fact, there is an interesting debate on how to push forward this simple idea. The goal of this project is to bring to light this topic, presenting well-known elements from different branches under this unconventional perspective, and also innovative "fringe" arguments. In each section, we explore a theme following the presented common thread: the focus will be on presenting the ideas behind the themes and all that is needed for a full understanding of them. Conversely, we discard the implementation details: we are not interested in the state-of-the-art little improvements because they are not useful for the comprehension and they cause us to look away from the big picture.

The following chapters are divided into macro-categories. The first investigates how probability is used as a powerful method to design algorithms that solve the main logic task, namely SAT problem. Then, we move to a compound presentation of some important related problems, which allow us to explore new interesting relations between logical reasoning and everything associated with it. Finally, we explore the open discussion about some approaches to unify logics and probability, and what it means for these fields. To deepen furtherly some relevant topics, an appendix is added including the code of a program that puts into practice real problems and algorithms.

Chapter 2

SAT Algorithms

2.1 DPLL and non-determinism

The most famous problem in dealing with logic is the satisfiability problem (SAT), which is finding a model for a given formula. In other words, we want to know if there exists an interpretation (assignment[1]) that makes the formula true. Due to the centrality of the role covered by SAT and its related algorithms, it seems reasonable to start here on the path to the introduction of a probabilistic approach to logic. Actually, there are well-known algorithms that use randomness and stochastic methods. Before that, it can be useful to understand how we can get there, analyzing a ”negative” example of a classical algorithm that makes no use of stochasticity.

For this purpose, we can briefly discuss the DPLL algorithm. It operates on formulas expressed in conjunctive normal form (CNF), and it consists in combining a backtracking-based assignment process with heuristics to avoid some recursive calls and speed up the search. In particular, DPLL recursively expands a partial interpretation by adding a new assignment to some variable; if it finds a conflict, it comes back and tries another one. A propositional formula F is in CNF if it is expressed as conjunctions of disjunctions, that is $\phi = \bigwedge_{i=1}^{m} C_i$, where $C_i = \bigvee_{j=1}^{n_i} x_{ij}$. The heuristics are:

early check with partial interpretation it is possible to establish if the formula is satisfied or unsatisfiable even before having a full assignment for the variables, exploiting the semantics of propositional operators

unit propagation if a variable is the only one present in a (eventually already-simplified) clause, we can directly assign the truth value to that variable in order to get the clause satisfied

[1]In propositional logic, interpretation and assignment can be used as synonyms

pure literal rule if a variable only occurs positively or negated in all clauses, we can set it respectively to true and false because it will rule out those very clauses

This is the pseudo-code:

Algorithm 1 DPLL(F, I): boolean

```
Input: a CNF formula $F$ and a (partial) interpretation $I$
Output: true if $F$ is satisfiable, false otherwise
  if every clause is true when $F$ is evaluated over $I$ then
      return true
  end if
  if some clause is false when $F$ is evaluated over $I$ then
      return false
  end if
  ($F$, $I$) = Unit-Propagation($F$, $I$)
  if $I$ is inconsistent then
      return false
  end if
  ($F$, $I$) = Pure-Literal($F$, $I$)
  if $F$ is empty then
      return true
  end if
  choose an unassigned variable $x$
  return DPLL($F$, $I \cup \{x = true\}$) or DPLL($F$, $I \cup \{x = false\}$)
```

How to choose what unassigned variables must be picked first? There is more than one way to face this question. Anyway we are more interested in focusing on another part of the story: the fact that this open choice makes DPLL a nondeterministic algorithm. The scope is to design a technique that is stochastic, and the nondeterminism may seems a step in that direction. Is it so? Actually, a probabilistic algorithm is nondeterministic, but the contrary does not hold in general. Furthermore, DPLL is often presented as a completely deterministic algorithm. This means that we have to think of an algorithm with a stronger requirement on its "randomic character".

2.2 GSAT and random-restart local search

A first stochastic algorithm comes up when we explore procedure that leverages local search. More specifically, GSAT is an algorithm that starts with an interpretation I and tries to modify it at each iteration in order to satisfy more clauses (until a model is found). This changing is done by flipping the assignment of a single variable in I; all the interpretations I' that are reachable by I in this way are called successors of I.

What makes GSAT stochastic is the choice of the interpretation it starts with, that is it chooses a random point in the space of interpretations and makes a greedy local search from that location. This can lead to situation in which we get stuck in a local maximum: all successors are interpretation worse than the current one, and we cannot satisfy all the clauses. This eventuality is not avoidable but we can use a random-restart routine, trying multiple times with different starting point. The space topology contains also plateaus, another tricky aspect we have to deal with. It is possible to mitigate this problem implementing a clever ”sidewalk-like” steps schema. An easy improvement is obtained with the use of tabu search, described later.

Algorithm 2 GSAT(F): I^* (or failure)

```
Input: a CNF formula $F$
Output: a model $I^*$ or failure
  while max-restart number is not reached do
      choose a random interpretation $I$
      while stopping criterion is not met do
          for each variable $x_j \in F$ do
              obtain $I_j$ by I changing the assignment of $x_j$
          end for
          $I \leftarrow I_j$ that maximize number of satisfied clause in $F$
          if $F$ is satisfied by $I$ then
              return $I$
          end if
      end while
  end while
  return failure
```

Different from DPLL, GSAT does not guarantee to find a model even if it exists, i.e. it is incomplete. This is because the local search, that can be more efficient but, at the same time, it doesn’t explore all the search space. In practice, GSAT cannot prove satisfiability, but it is somehow more effective in finding the model.

As we have seen, this algorithm makes use of a random step as a starting point for its exploration, and this means that we can consider it stochastic. Anyway, it is possible to go further, describing a procedure that exploits probability to perform the actual search.

2.3 WalkSAT and random walks

WalkSAT is the algorithm that we answer our question. It shares some aspects with GSAT: it's a local seach, starts from a random interpretation and is an incomplete algorithm. However, it uses a different paradigm: random walks. In standard local search, we move blindly towards the best successor state starting from the current one, maximizing locally one objective function. This is often referred to as hill-climbing search, and it is precisely what we have done in GSAT, trying to maximize the number of satisfied clauses. However, this is the reason why we cannot leave a local maximum. Random walks allow us to take some "bad" actions with a certain probability, that is move in a state (interpretation) that is less rewarding (satisfies fewer clauses). This may seem counterintuitive, but let us explore more freely the search space, sometimes pointing to a worse direction and leaving local maxima.

Algorithm 3 WalkSAT(F, p): I^* (or failure)

```
Input: a CNF formula $F$ and a probability value $p$
Output: a model $I^*$ or failure
  while max-restart number is not reached do
      choose a random interpretation $I$
      while stopping criterion is not met do
          choose a random clause $C$ from $F$ unsatisfied under $I$
          with probability $p$, choose a random variable in $C$ and flip its assignment
          otherwise, flip the variable in $C$ that maximizes the number of satisfied clauses
          if $F$ is satisfied by $I$ then
              return $I$
          end if
      end while
  end while
  return failure
```

The backbone of WalkSAT consists of this procedure. We first choose at random an unsatisfied clause. Then, with probability $1 - p$ we flip the assignment of the variable that allow us to maximize the satisfied clauses, as we do in GSAT, and with probability p we flip the assignment of a random-picked variable, even if it is harmful, in order to take new paths.

WalkSAT has covered an important role, being the most used, referred to and re-interpreted among all the others. It is the baseline not only for SAT problem, but also for all the related ones (they will be discussed in the next chapter). There exist a lot of variantions and improvements of this algorithm, each one useful for solving related problems or speeding up a little bit the computation over well-known (hard) problems. Anyway, the core structure (and the use of randomness approach) remains the same. To be precise, we could come up with slightly different algorithms depending on how we combine greediness and random walk. Anyway, this is not of much interest today because WalkSAT have proved to outperforms other methods (such as GSAT with random walk).

2.4 SASAT and adaptive probability

Local search algorithms are not limited only to the two presented. Simulated Annealing is share some aspects with both greedy search and random walks. It allows to pick bad successor states (interpretation), as in WalkSAT, but this choice is not taken with the same probability every time. In fact, it seems reasonable to avoid moves that are too harmful, and also to change our decisions based on how much iteration is performed. This last point can be justified simply by thinking that if we have performed many iterations, then it's more likely that we are close to the optimum.

In order to translate these statements into a runnable algorithm, we can define a probability value using the following logistic function

$$p = \frac{1}{1 + e^{-\delta/t}}$$

where δ is the increase in the number of clauses made true, and t is called temperature and updates every iteration k

$$t \leftarrow T \cdot e^{-k \cdot decay}$$

Summing up, instead of picking always the best interpretation, we choose one and we accept it with a certain probability p even if it doesn't satisfy more clauses, taking into account that this probability must decrease with the harmfulness of the choice and with the temperature (time that goes by).

The presented algorithm is just one of the possible processes we can come up with and could be furtherly refined. For example, it is possible to flip the assignment with higher probability if the related variable is present in one of the unsatisfied clauses, or we can simply pick one successor interpretation probabilistically, avoiding iterating over the variables. Depending on what implementation we use, the algorithm tradeoffs exploration and exploitation with respect to the search space (same as the Genetic Algorithms discussed

Algorithm 4 SASAT(F): I^* (or failure)

```
Input: a CNF formula $F$
Output: a model $I^*$ or failure
  while max-restart number is not reached do
      choose a random interpretation $I$
      $k \leftarrow 0$
      while stopping criterion is not met do
          update temperature
          for each variable $x_j$ do
              compute delta
              obtain $I$ flipping the assignment of $x_j$ with probability $p$
          end for
          update $k$
          if $F$ is satisfied by $I$ then
              return $I$
          end if
      end while
  end while
  return failure
```

in next section). The method to update temperature can vary and it is often referred to as the cooling schedule.

2.5 Genetic Algorithms and refinements

Most promising ideas to solve SAT problems come from the belief that a mixed strategy of the previous techniques could be more effective. In the search for the global optimum, we have to be efficient and we have to avoid the pitfalls of local search already mentioned. More precisely, we want to use both exploration and exploitation, that is to move rapidly over the search space and refine the result locally when we are close to the solution.

This is where genetic and evolutionary algorithms come into play, a family of algorithms that are inspired by natural selection and reproduction and combine several steps to reach the goal. The main characteristics of a genetic algorithm (GA) are:

population it begins with a set of k random states (in our case, assignments), not only one as in the previous cases. This aspect appears also in a simpler method, stochastic beam search. What makes GAs unique is the encoding of the states in a string of bits. How to do this depends on the problem: the more natural and studied way for the SAT problem is to create strings of m bits, where m is the number of variables, and consider the value of the bit as the truth assignment to the related variable. There exist other representations (floating point, clausal, path representation...), but they are way less used.

fitness function it refers to the objective function to be optimized. Depending on if we see the problem as a hill-climbing search or a gradient descent one, it must be maximized or minimized (as we have done for all other algorithms, we choose the hill-climbing version, but this is not so relevant for the overall understanding of the method). In the SAT problem, often, it simply counts the number of satisfied (or unsatisfied) clauses with a given assignment.

selection it is the choice by which states are mixed together, that is the probability of being selected for reproduction. This probability is often proportional to the corresponding fitness function of the states. The steps of reproduction are used to run primarly (even if not exclusively) the exploration phase, producing new states (children) starting from the given ones (parents) and eventually reaching new areas of the search space. It allows to take "bad" decisions such as what happens in WalkSAT and Simulated Annealing. Sometimes, as happens in nature, we want to ensure some diversity in the chosen parents (string distance) to avoid a too low recombination power.

crossover it is the actual mixing step, in which strings representing states are taken by the selection and recombined to obtain new ones. This is done by choosing a turning point within the length of the strings and crossing over the substrings obtained. In the end, we obtain again k states.

mutation it is the last step and it is useful for a further move toward new interpretations: with a certain probability a random bit in some states is flipped. This is exactly the same as what we do in WalkSAT, flipping the assignment of a variable, so that mutation is sometimes implemented using WalkSAT itself or other similar algorithms.

local search finally, the reproduction cycle ends. Before recurring the process, starting from the computation of the fitness function for the new states, we can decide to perform a local search to give strength to the exploitation phase. In this case, we simply apply some local search algorithms to each state in parallel.

Obviously, each phase described can be implemented in several ways and is prone to multiple refinements. There are many parameters to be fine-tuned performing statistical analysis, and some steps could be discarded to promote an approach over a different one.

As an example, at the end of every step, we can decide the number of states we have to consider, that is how many states we want to choose as parents for the following round. After each crossover and mutation step, should we keep only the generated states, or should we take the best k among all parents and children? Or, as happens in stochastic local beam, should we introduce a probability value to keep some bad states and elude the local maximum? We can also decide the probabilities of crossover and mutation. In literature, an adaptive strategy is often used, in which these probabilities depend on the number of iterations performed and especially on the values of fitness obtained.

The fitness function itself can be adaptive: some implementation does not consider only the sum of satisfied clauses, but a weighted sum, where the weights are updated at each iteration and stand for how much is difficult to satisfy the related clause (this is called stepwise adaptation of weights). In this way, the algorithm is pushed towards finding a solution to those difficulties. Differently, a ”refined” fitness function uses a simple sum, but it adds a parametrized discriminative term that acts as a tie-breaker for the interpretations that satisfy the same number of clauses.

Finally, an alternative way to perform changes on individual states (for example, during local search or mutation) is the tabu search. This consists simply in using a queue of past states that are forbidden, so that the algorithm cannot return on those very states for a limited amount of time. It can be helpful against local maximum and plateaus, and it can be used regardless of the genetic algorithm (same for weighted clauses).

Chapter 3

SAT-Related Problems

3.1 CSPs and Cycle-Cutset induced search

One way of finding solutions for a given problem is always trying to disclose relations between that problem and other known problems that are similar to the first one. We could use algorithms that are born for different purposes, adapting them to our case or mixing different strategies.

Our first attempt is to see SAT problem in a new light. In fact, it can be easily interpreted as a constraint satisfaction problem (CSP). Let's first define a CSP. It is composed of a set of n variables, a set of n domains that represent possible values for each corresponding variable, and a set of m constraints over the cartesian product that defines allowed relations between variables. Solving a CSP means finding an assignment for the variables that is consistent with the constraints.

There is more than one way to translate a SAT problem into a CSP. The more straightforward way is to use each boolean variable as a variable of CSP, and each clause as a constraint (all the combinations of values are allowed except for the ones that do not satisfy a clause). It is also used another method: each clause can be considered a variable of CSP, the related domain contains the literals within the clause, and the constraints are given by two clauses that contain opposed literals (all the combinations of values are allowed except for the ones that assign a positive literal to one clause and the same literal negated to another one). This second formulation is a little bit more convoluted, but it allows us to define a binary CSP from our SAT problem.

A binary CSP has the property of containing only binary relations, and this has several consequences, first of all the possibility of creating a simple constraint graph to help us visualize the problem and run graph algorithms.

A constraint graph has a vertex for each CSP variable and an edge for each relation between two variables. Obviously, we can have as well graphs with k-ary relations, even if they are slightly more difficult to manage.

Talking about algorithms, there are several techniques to solve CSP, some of which resemble SAT solvers. There exists a backtracking algorithm, powered by various heuristics, and also local search does the job. Here, we are interested in a method that builds on stochastic algorithms to come up with a new strategy. The idea is to leverage the ”problem structure”, visible in the described constraint graph.

It can be proved that a CSP can be solved in linear time if the related graph has no loop, using the Tree Algorithm that creates a topological sort over the nodes and ensures directed arc consistency (that is, there always exists an allowed assignment for a variable with an incoming edge). This is a strong bound on the complexity of reasoning. Obviously, not all CSPs have this characteristic, and here is where we can make use of stochastic local search.

We can first identify a set of variables such that the graph obtained by removing those variables is a tree (in general, a forest). Thus we have partitioned the vertices of the graph into two subsets: the set of ”tree” variables and the so-called cycle cutset variables. At this point, we can run a double-phase algorithm. First, we pick a random assignment for the variables. Then we alternatively execute the Tree Algorithm (with the values of cycle cutset variables fixed), and a stochastic local search algorithm (with the values of the tree variables fixed). It is possible to substitute this last one with another procedure, but SLS has proved to be effective in this case. For what concerns the Tree Algorithm, it is not required to satisfy all the requirements at once, and it is guided by a reward (or cost) function that assigns a value with respect to the number of satisfied (or unsatisfied) clauses. This method not only maintains the benefit of the two algorithms, but also enforces the joint effort between them: it is not a simple interleaved execution, but a real exchange of information on how to direct the search. It means that Tree Algorithm informs local search and vice versa.

3.2 Reduction and Bayesian Networks

As we have seen, it is always possible to ”cast” our SAT problem to another one, and solve the latter with a stochastic algorithm (or a deterministic one). This is the same idea exploited in polynomial-time reduction ($\leq_p$), the process of trasformation used to classify the complexity of problems. SAT, as a well-known NP-complete problem, is often been used to perform the reduction.

The following list sums up some examples:

- SAT $\leq_p$ 3-SAT
- 3-SAT $\leq_p$ INDEPENDENT-SET $\leq_p$ VERTEX-COVER $\leq_p$ SET-COVER
- 3-SAT $\leq_p$ DIRECTED-HAMILTON-CYCLE $\leq_p$ HAMILTON-CYCLE
- 3-SAT $\leq_p$ 3-COLORABILITY
- 3-SAT $\leq_p$ SUBSET-SUM $\leq_p$ KNAPSACK

The most interesting reduction for our discussion is the first one, which states that every SAT problem can be translated into a 3-SAT problem, where each clause in CNF has exactly three literals. The transformation can be easily conducted by padding clauses of lengths one and two, and breaking down clauses longer than three into smaller ones with some auxiliary variables. This will be useful later.

Let's now focus again on our topic. Saying that a point of contact between probability and propositional logic comes by the reduction of SAT to another different problem might seem a stretch, or at least a rather big leap. Actually, it turns out that polynomial-time reduction can help us to point out a new aspect of this point of contact. In fact, probability is not only used to design algorithms. There are problems intrinsically probabilistic in their definition, and we can use reduction to show that a SAT problem can be interpreted as one of these "probabilistic" problems, revealing a hidden connection in their nature that makes the two part of the same class.

In particular, we can reduct 3-SAT to the problem of Probabilistic Inference using Belief Networks. A Belief Network (or Bayesian Network) is a graph with probabilities associated with it. In detail, we have a vertex for each random variable, and directed edges that represent conditional dependencies between variables. Source nodes are assigned by prior probabilities, and if we have an edge from a set of predecessors X_i to Y, there is a conditional probability $P(Y|X_1, X_2, ...)$. In this way, we can avoid the full explicit representation of the joint probabilities distribution, considering only the restricted ones related to dependencies and saving space.

Doing inference on Bayesian Networks means calculating (conditional) probabilities among (conjunctions of) variables. How 3-SAT can be translated into a probabilistic problem? First of all, we use a source node for each boolean variable x_j. The prior probabilities of these nodes will be $P(x_j = True) = \frac{1}{2}$. Then, we introduce a node for each clause C_i and three incoming edges related to the variables x_j contained in that clause. The conditional probability attached to the edges $P(C_i = True \,|\, x_{i_1}, x_{i_2}, x_{i_3})$ is one when at least one variable is True. Finally we can create a node F that represent the validity of the entire formula. There must be an edge from each "clause" node to that last one and the probability $P(F = True \,|\, C_1, C_2, ...)$ is one if all clauses are True. Using this construction, the probability $P(F = True)$ is greater than zero if and only if the initial 3-SAT problem is satisfiable. As an alternative to the last node, exploiting a tradeoff between graph representation and input data representation, we can add some extra node D_i such that there is an edge from C_i and D_i and one from D_{i-1} and D_i. In this equivalent case, the last D_m is interpreted as F and the probabilities $P(D_i = True \,|\, C_i, D_{i-1})$ is one when both C_i and D_{i-1} are True.

This is a first use of Bayesian Network, but the bridges betweeen the two realities are not done yet. What is more important for the sake of our goal is not just all the algorithmic point of views (which, however, are many). We will see in the next sections how to link logical reasoning and probabilistic reasoning.

3.3 Max-SAT and approximated algorithms

Until now, we have described problems that are somehow related to SAT. Same considerations, or even tougher ones, can be done by analyzing other kinds of problems, those that are born directly from SAT and evolved from the same ground. They are Max-SAT and Model Counting. A propositional formula can have one model, multiple models, or none at all. Then, when we cannot satisfy the entire formula, we may ask how many clauses within that formula can be satisfied at once. This is the Max-SAT problem.

Max-SAT offers us the opportunity to explore approximation algorithms, because they are commonly used to deal with the problem. A very simple idea is to assign to each variable independently a random truth value, that is True or False with a probability 1/2. Doing so, we have a probability of $1/2^{n_i}$ of the clause C_i to be unsatisfied, because each of its n_i literal must have the "wrong" value. Let Z_i be a Bernouilli-distributed random variable that is one if C_i is satisfied. We have that the expectation $E[Z_i]$ is equal to the probability of C_i being satisfied, that is $E[Z_i] = 1 - 1/2^{n_i} \geq 1/2$. Using linearity of expectation,

$$E[Z] = \sum_{i=1}^{m} E[Z_i] \geq \frac{m}{2} \geq \frac{\text{OPT}}{2}$$

because the maximum number of satisfiable clauses cannot exceed the overall number of clauses. In this way we have a 1/2-approximation of our problem. At a first glance it may seem a poor result, but the interesting aspect comes up when we consider clauses of fixed size k. For the Max-3-SAT problem, when $n_i = k = 3$, we have $E[Z_i] = 7/8$ and $E[Z] = 7m/8$. Thus, we can obtain a 7/8-approximation. This is quite impressive if we consider the simplicity of the process and the related results obtained in literature: it is proved that this is the best approximation for the problem unless P=NP, and the Johnson's algorithm of executing multiple random re-assignments succeeds in exceeding the expected value with polynomial expected running time. The overall process can be de-randomized and the tight bound is found to be an approximation factor of 2/3.

Another way to solve the problem with a good approximation ratio is the iterative update of lower and upper bounds on how many clauses are satisfied. Algorithms that refine upper bounds (UBs) are called linear search Sat-Unsat, while algorithms for lower bounds (LBs) are linear search Unsat-Sat. The idea is to combine the two strategies. We can repeatedly call a SAT solver to search for a solution and update each time the bounds on satisfiable clauses. There is also an alternative, interpreting the bounds differently. Let LB be the clauses already satisfied and UB the clauses not yet satisfied, we can set a variable at each iteration, compute the new bounds and verify which of the two truth assignments has a positive gain in the average between LB and UB. If there is only one with a positive gain, we set the variable to that truth value and simplify the problem; if both assignments have a positive gain, we can break the tie with a weighted random choice.

There are plenty of other mechanisms that are out of the scope of our discussion, because they consist in interpreting Max-SAT as an Operations Research problem and they don't put probabilistic and random processes as their core steps. For the sake of completeness, we can mention integer linear programming (ILP), with the relative LP-relaxation technique, and semidefinite programming (SDP).

Everything we have said can be repeated with minor changes for Weighted Max-SAT, where a weight is associated with each clause and we want to maximize the weight of satisfied clauses (or minimize the weight of unsatisfied clauses). Max-SAT corresponds to Weighted Max-SAT where all the weights are equal. We can generalize the problem even more: Partial Max-SAT consists of a set of hard constraints, that have to be satisfied, and a set of soft constraints, that we'd like to satisfy for the most possible part. Max-SAT is Partial Max-SAT with only soft constraints and no hard constraints. Finally Weighted Partial Max-SAT has both hard constraints and weights for each clause.

Max-SAT does not differ too much from SAT problem and this is why we can simply adapt the algorithms already discussed for this new case.

In particular, all stochastic local search methods are useful also with Max-SAT, and they are used not only in practice, but also as a baseline to compare other algorithms more specific for the situation. The most common techniques are Simulated Annealing and MaxWalkSAT, net of particular applied heuristics. On the contrary, plain unit propagation can fail greatly. If we uncritically satisfy all unit clauses, we may end up with a huge number of other unsatisfiable clauses. Due to the fact that we cannot satisfy them all, sometimes we prefer "sacrifice" unit clauses in order to have the other satisfied. Sometimes, unit propagation is used as an addiction component of larger procedures.

Most of the state-of-the-art algorithms go in two directions. The first one is the core-guided search, which is a method by which we exploit the use of unsatisfiable cores. They are sets of clauses that cannot be satisfied all together, and identifying the minimal ones can help us on counting maximum satisfiability. The second direction is the development of SLS algorithms equipped with a strategy to continuously adapt weights for the difficult clauses and a k-neighboring search (or, more generally, a variable-depth neighbor search). All the methods proposed for SAT are based on the principle that we generate each interpretation starting from a "preceding" one and changing a single assignment (one flip). We have already talked extensively about the curse of local optima. The most natural way to overcome this difficulty is the implementation of look-ahead methods to escape from a local maximum and, at the same time, perform a better exploration. We first flip the assignment of a variable, even if it is harmful, then we perform another change with respect to another variable, trying to jump to a more distant search space area in one iteration. The idea can be made explicit in many ways and can be repeated for more than two flips, also adapting the choice to the context. For what concerns the weights, it's almost always used a strategy based on different schemas for hard and soft clauses, as happens in SATLike algorithms. This is way more effective, given the fact that we prefer to satisfy hard clauses first. The refinement of these schemas is a field of great improvements in these types of algorithms.

3.4 Model Counting and random sampling

The propositional model counting problem tries to answer the question: how many different models does the formula have? The research around this topic has followed the two main directions (already presented) of complete search and local search. The first one is based upon DPLL and the correlated adaptations. Anyway, to cope with model counting means an exhaustive enumeration of all existing assignments, and this is quite impractical when the formulas contain a lot of variables. This is why the actual model counting algorithms have acceptable performances only on smaller problems with respect to those solved by SAT.

Due to these reasons, we can try to use the stochastic approach: it is more efficient for SAT instances, and likely it will be so also in this case. Actually it is, but there are differences. First, traditional stochastic local search goes deep in the search space to identify quickly an area where we find a single satisfying assignment. But now we are interested in all satisfying assignments. To get around the obstacle, it is developed an algorithm called ApproxCount, which takes advantage of sampling to compute an approximation of the model count for one formula.

Let's assume that a formula F has M different models, and we can sample uniformly at random this set of satisfying assignments. Thus, we can compute the ratio of the sampled models having a certain value, say x, set to True. This number approaches the real fraction $\gamma = M_x/M$ of models of F with x equals to True over the total number of models. How is this useful? We can use the inverse $M = M_x/\gamma$ to estimate the number of models, and now we have reduced our problem to a smaller one, because the value of a variable is fixed. We can iterate the same idea until we obtain a full assignment, or at least a formula small enough to perform a real enumeration of assignments over the remaining variables. It is possible to use some heuristics to choose the variable to set, but the most effective method is a random pick. Obviously, we consider the case of setting the value to True, but it is equal to setting it to False: we use the value most present in the sample for the chosen variable. To perform the sampling, a first proposed method is SampleSAT, based on combining steps of WalkSAT with steps of the Metropolis algorithm (similar to Simulated Annealing on using probability controlled by a "temperature" parameter).

ApproxCount has some good properties, such as efficiency and scalability with respect to the size of the problem, but it gives us acceptable results only under a strong assumption: we need to sample the models uniformly or near-uniformly. This is why it has been developed an idea that works well even in presence of a bad sampling process, named SampleCount. SampleCount guarantees a fast computation of good lower bounds to the real model count. The quality of the bounds may deteriorate, but they are correct with high confidence regardless of the sampling, and it can be formally proved using Markov inequality. More in detail, we use the sampler no more for selecting the variable and the multiplier, but only for choosing the order of the variables. We choose first the more balanced variable, that is the one with almost equally positive and negative settings in the solution. Using a random truth value for each variable and a multiplier fixed to 2, we obtain surprising good results even with biased samplers. In expectation, we have exactly the true model count, i.e. $E[M_S] = M$. Since that, by Markov inequality

$$P\left(M_S \geq a \cdot E[M]\right) \leq \frac{1}{a} \;\Rightarrow\; P\left(\frac{M_S}{a} \leq M\right) > 1 - \frac{1}{a}$$

The described method is assisted by some refinements such as

- the use of SampleSAT to find the balanced variables
- the biased coin strategy: to use a biased assignment for the variable, no more randomly set, taking into account that this obliges us to consistently modify the multiplier by the same bias value
- the safety check: to verify that a variable can be set in both ways, eventually using WalkSAT or other techniques, or one of the two would lead the formula to be always unsatisfiable
- the ”equivalence” strategy: when there is no sufficiently balanced variable, we can select a couple of variables with balanced combined polarity and replace one with the other

Finally, starting from SampleCount, it has been developed an alternative way to speed up the process based on probabilistic reasoning, BPCount. We have already seen in previous sections how SAT and Belief Networks are connected. In this case, we can use belief propagation in order to compute the marginal probabilities of the variables and identify faster what is the balanced ones. In fact, the fraction of a variable with a certain value in satisfying assignments is nothing else than the marginal probability of that variable with the same value. The iterative computation underlied by BPCount does not always converge, but this wrong behavior can be mitigated by a slight variant of the base algorithm (”message-damping”). There exist also methods to compute the upper bounds of the model count for a given formula, but they are mainly based on multiple applications of DPLL algorithm variants.

Chapter 4

Probability Logics

4.1 Towards a new formalism

From what we have seen, Logic and Probability do have some points of contact. Up to now, we showed how it is possible to fruitfully apply stochastic processes inside algorithms for solving logic instances. Despite it is a great fact for so many fields within computer science and artificial intelligence, it might be thought that it doesn't satisfactorily answer the questions that opened up our discussion. Logic still remains untouched, and we have restricted our range of action only to the reasoning on SAT. It's therefore reasonable to go ahead and try to find a way to construct a new logic formalism, capable of unifying the two fields. This formalism must have a fundamental characteristic: it must be able to structurally express uncertainty. It is a major change with respect to the possibility and expressing power of traditional logics, and it may create new opportunities for the previously mentioned fields (knowledge representation, planning, reasoning, data management, formal methods, information retrieval...).

Let's start with some possible counter-argumentations. Someone points out that "trivially" probability itself can be seen as an extension of propositional logic, with values that range from 0 to 1 rather than taking only 0 and 1 as truth values. This can be a point of view on the subject, but it is not all the story. As said, we are searching for a way to address directly uncertainty and confidence within logical reasoning, and this can be modeled in several ways. This goal goes way beyond propositional logic. Others might say that fuzzy logic is already a hyped solution to our problem. Fuzzy logic belongs to the family of many-valued logics and deals with vagueness. It has values in the interval [0,1], and these values are interpreted as degrees of truth. The debate about distinctions between probability and fuzzy logic has led many researchers to show their substantial difference, not only in how we interpret their values, but also in how they operate and in which contexts. Again, this is out of our scope, and it does not have any influence on our topic.

It would be more precise to talk about a family of probability logics. In fact, as also happens for traditional logics, there is no unique way to see the question, and it depends on what result we want to obtain. In the following sections, we'll see respectively how to define a first probability logic and its reasoning background, how to think of new operators, and how to construct a logic that operates on "opinions". This is not a complete discussion, but a perfect dive into this growing topic to analyze different perspectives. An interesting common element seems to be the strong connection between these logics and complex inference systems such as Bayesian Networks and Trust Networks. This is thanks to the very idea beyond probability logics, which is being able to manage probabilistic reasoning.

4.2 Probability Functions

Our first attempt is to introduce a probability function to assert uncertainty about some propositions, maintaining however the whole traditional structure and classical operators. Let $P : F \mapsto [0, 1]$ be a probability function that maps logical formulas to a probability value. We have properties such that if a formula is valid, its probability value is one; if two formulas are logically equivalent, then they have the same probability value. It's easy to construct the corresponding uncertainty function $U(\phi) = 1 - P(\phi)$.

Given the probability function, we can define the concept of probabilistically validity: given a set of premises Γ and a consequence ϕ, $\Gamma \models_p \phi$ if and only if, for all probability functions P, if $P(\gamma) = 1$ for all $\gamma \in \Gamma$, then $P(\phi) = 1$. These probability values can be seen as a generalization of the truth values or as a way to measure uncertainty about the propositions. From this new concept we can state that, when we are certain about each premise of a deduction, we are certain also about the consequence. What about other values of probability for the premises?

A more general result is given by the upper bound $U(\phi) \leq \sum_\gamma U(\gamma)$. It says that the uncertainty about the consequence is at most the sum of the uncertainty about the premises. This upper bound depends on both the probability values and the number of premises. Obviously, if some premise is not necessary to derive the consequence, the upper bound will be less refined, while it will be exact when all the premises are relevant. Starting from this last remark, we can make the upper bound even better just by weighting the sum by a "degree of essentialness", which is a value proportional to how relevant each premise is.

For example, let's consider four premises p, q, r, s and the valid conclusion $p \wedge (q \vee r)$. Let $P(p) = 14/15$, $P(q) = 11/15$, $P(r) = 11/15$, $P(s) = 12/15$. It is easy to compute the upper bound

$$U(p \wedge (q \vee r)) \leq \frac{1}{15} + \frac{4}{15} + \frac{4}{15} + \frac{3}{15} = \frac{12}{15}$$

In order to get a better result, we can note that s is not a relevant premise for the conclusion. Discarding it, we obtain $U(p \wedge (q \vee r)) \leq 9/15$. But we have not done it yet. Analyzing better the conclusion we can see that only p is "strictly relevant", while we need just one among q and r. This is when the degree of essentialness comes into play. We can write that $E(p) = 1$ and $E(q) = E(r) = 1/2$. Formally, $E(\gamma) = 1/|S_\gamma|$, where $|S_\gamma|$ is the cardinality of the smallest essential premise set that contains γ. In our example, we can then use the computed values as weights and obtain

$$U(p \wedge (q \vee r)) \leq 1 \cdot \frac{1}{15} + \frac{1}{2} \cdot \frac{4}{15} + \frac{1}{2} \cdot \frac{4}{15} + 0 \cdot \frac{3}{15} = \frac{5}{15}$$

From what we have said it follows that we can come up with a more general notion of probabilistic validity for an argument, in which $\Gamma \models_p \phi$ if and only if, for all probability functions P, $U(\phi) \leq \sum_\gamma E(\gamma)U(\gamma)$.

The most notable limitations of this result are the absence of a lower bound, less interesting than the upper bound but still useful in some contexts, and especially the requirement to know the exact probability values of the premises. In many cases, we can have access only to partial informations, or the premises of an argument can be obtained as a product of another reasoning process. Further research on this topic shows how it is possible to overcome these limitations. This leads to a complete notion of probabilistic validity that involves the development of functions to compute the minimal probability interval Y (lower and upper bounds) for the conclusion of an argument given the intervals X_i of each premise. This is sometimes called standard probabilistic semantics, and denoted by $\gamma_1^{X_1}, ..., \gamma_n^{X_n} \models \phi^Y$. Efforts to create a formal framework for probabilistic logic and real applications to inference problems go in this direction.

4.3 PSAT

Introducing the idea of probability functions, we can define a new satisfaction problem, namely PSAT. It consists of probability statements on a set of propositional formulas, and the objective is to check the consistency (or inconsistency) of the statements altogether. Formally, we have n variables, k formulas ϕ_i using those variables, and a set of k statements $P(\phi_i) \lesseqqgtr p_i$. Let π be a probabilistic distribution over the 2^n possible interpretations for the n variables, and $I(\phi)$ equal to 1 if the formula ϕ is satisfied under the interpretation I (0 otherwise). Given π, the probability of a formula is given by

$$P_\pi(\phi_i) = \sum_{I:I(\phi_i)} \pi(I)$$

Thus, PSAT searches for a distribution π such that the statements given as input are all satisfied. It is possible to rewrite the problem using a matrix formulation $A\pi \lesseqqgtr p$, where $\pi \geq 0$ is the vector representing the distribution, p is the vector of probabilities constraints (plus 1 as the first element), and $A \in M^{(k+1)\times 2^n}$ is a matrix in which the first row has all ones (useful to ensure that the sum of components of π is one) and other elements are $a_{ij} = I_j(\phi_i)$. Most of the algorithms for solving PSAT are based on linear programming.

Let's consider the following example. A is looking for a gift for his friend B. A knows that B loves to read, but he doesn't know if his friend currently has a large backlog of unread books. In this case, buying him a book may not be the right choice. On the contrary, A thinks it's unlikely that B doesn't have a backlog and at the same time doesn't appreciate the gift. Let x_1 be the boolean variable that represents if B is happy to have a new book, and x_2 if he has a large backlog. Given his informations A writes down the following probability statements: $P(x_1 \wedge x_2) \leq 0.1$ and $P(\neg x_1 \vee x_2) \geq 0.4$. Is there a probabilistic distribution π such that the statements are satisfied and $P(x_1) \geq 0.5$? What about $P(x_1) \geq 0.75$? In the first case, a possible solution is

π	x_1	x_2	$x_1 \wedge x_2$	$\neg x_1 \vee x_2$
0.10	T	T	T	T
0.60	T	F	F	F
0.15	F	T	F	T
0.15	F	F	F	T
1.00	0.70	0.25	0.10	0.40

As happens for CNF, it is also possible to define a PSAT Normal Form. It is a binary partition of the set of probability statements. The first subset must contain only the statements of the type $P(\phi) = 1$, and it represents a SAT subproblem (often written only using formulas, getting rid of probabilities). The second subset contains statements of the type $P(y) = p_i$, with $p_i < 1$.

When we have an equality statement, it is straightforward and y is simply the formula ϕ. In other situations, we rewrite the statement using the extra variable y and imposing a consistent SAT constraint over ϕ and that new variable. In the end, PSAT Normal Form consists of a probability assignment part (called evidence) that must be consistent with the SAT instance (called theory). From this point of view, π becomes a probability distribution over $k+1$ theory-consistent assignments.

How can we transform the previous example in PSAT Normal Form? First of all we can rewrite the statements in a coherent form: $P(x_1 \wedge x_2) \leq 0.1$, $P(\neg(\neg x_1 \vee x_2)) \leq 0.6$ and $P(\neg x_1) \leq 0.5$. Then, we can add three support variables y_1, y_2 and y_3, and use them to construct the two sets described before. The evidence (probabilistic) set will be $\{P(y_1) = 0.1, P(y_2) = 0.6, P(y_3) = 0.5\}$; the theory (SAT) set will be $\{x_1 \wedge x_2 \rightarrow y_1, \neg(\neg x_1 \vee x_2) \rightarrow y_2, \neg x_1 \rightarrow y_3\} \equiv \{\neg x_1 \vee \neg x_2 \vee y_1, \neg x_1 \vee x_2 \vee y_2, x_1 \vee y_3\}$.

4.4 Probability Operators

A different way to ideate probability logics is creating new operators, alongside the classical ones, that directly deal with uncertainty about propositions. These operators can respond to a qualitative approach, in which we want to compare uncertainty but we don't know and we don't need exact probability values, and a quantitative approach, where real values are managed.

In the first case, for example, they have been proposed a unary operator $\Box\phi$ that stands for "probably ϕ" (it is not a modal operator even if it shares the same symbol), and a binary operator $\phi \geq \psi$ that stands for "ϕ is probable at least as ψ". This last proposal is quite interesting because it is easily extendible to sequences $(\phi_1, ..., \phi_n \geq \psi_1, ..., \psi_m)$, and it is more expressive than what appears at a first glance. In fact, it allows us to formulate quantitive properties without explicitly addressing them. We can state for example that ϕ is certain ($\phi \geq \top$, i.e. $P(\phi) \geq 1$) or ϕ has a probability greater than 50 percent ($\phi \geq \neg\phi$, i.e. $P(\phi) \geq 1 - P(\phi)$).

For what concerns the quantitative approach, the first formula we'd like to express is $P(\phi) \geq q$, in order to state the truth-ness of "the probability of a formula is at least a number q". This is already quite expressive, but we can add also other two fundamental features: the sum and the product between probabilities, i.e. $P(\phi) + P(\psi)$ and $P(\phi)P(\psi)$. Even without the extension provided by linear combinations, we could already express many non-trivial statements, such as $P(\phi) \leq q$ (i.e., $P(\neg\phi) \geq 1 - q$), $P(\phi) = q$ (i.e., $P(\phi) \geq q \wedge P(\phi) \leq q$) and $P(\phi) \geq P(\gamma)$ (i.e., $P(\phi) - P(\gamma) \geq 0$).

Finally, the more general form of sums and products are linear combination and polynomial weight formulas $a_1P(\phi_1) + ... + a_nP(\phi_n) \geq q$. They help us to state complex relation and concepts such as the statistical independence of two variables $P(\phi)P(\gamma) - P(\phi \wedge \gamma) = 0$.

4.5 Subjective Logic

Among the family of probability logics there is one interesting formalism called Subjective Logic that tries to take into account the level of confidence in the probability arguments. Then, we are not worried only about the uncertainty of propositions, but also (possibly different) beliefs that agents (whether they are human or not) have about a situation. Subjective Logic takes its move from the Dempster-Shafer belief model. Possible situations (states) about a fact are represented by a set named "frame of discernment" Θ, and an observer can assign a belief mass $m_\Theta(x)$ to each of the subset $x \in 2^\Theta$. A binary frame of discernment is a set with just x and $\neg x$ as states. It is possible to collapse whatever frame of discernment to a binary by performing a partition: we consider one subset (not necessarily atomic) of the starting set as x and the complementary subset as $\neg x$. This produced result is called "focused" frame of discernment.

Here is an example of what we can model with a frame of discernment. Let's consider two countries A and B. Country A wants to find out whether country B intends to use military aggression, and the analysts consider three possible hypotheses: no military aggression, minor military operations, and full invasion. We can assign to the hypotheses respectively the states x_1, x_2 and x_3. These states together define our current frame of discernment, and they are possibly associated with a belief mass. If we want to "focus" on a particular binary condition, let's say "there is an actual military offense, regardless of the intensity of it", we can create a new frame of discernment containing just two states. In our case, we can state $y = x_2 \cup x_3$ and $\neg y = x_1$.

From the introduction of belief masses, we can define four new values:

Belief function $b(x) = \sum_{y \subseteq x} m_\Theta(y)$

Disbelief function $d(x) = \sum_{y \cap x =} m_\Theta(y)$

Uncertainty function $u(x) = \sum_{y \cap x =,\, y \not\subseteq x} m_\Theta(y)$

Base rate $a(x/y) = |x \cap y|/|y|$

How all these definitions are linked to Subjective Logic? They trace the "opinion space", to which our new formalism will operate. A binary opinion about x is the tuple $\omega_x = (b(x), d(x), u(x), a(x))$. We can generalize the concept to multinomial opinions. It is particularly advantageous to have a binary (or focused) frame of discernment, because there exists a simple mapping between opinions' components and the belief mass of each state in that case. For example, if we want to express a certain proposition we'll have $\omega_x = (1, 0, 0, 0.5)$, while if we want to express a dislike for x with a certain amount of uncertainty we'll have something like $\omega_x = (0.2, 0.6, 0.2, 0.5)$.

As said, Subjective Logic is worried not only by representing belief about a fact, but also by keeping track of the different points of view about the same fact. This comes from the idea that never exists any objective truth. Thus, we denote as ω_x^A an opinion about the binary proposition x from the agent A. Recalling the previous example, in this way we can manage and discriminate opinions about a possible military invasion (y) from the analyst P (i.e., ω_y^P) and from analyst Q (i.e., ω_y^Q).

There are a lot of interesting theoretical results about the opinions that need to at least be mentioned. In theory, we could discard a redundant component from the tuple because $b(x) + d(x) + u(x) = 1$, but it is used a four-dimensional opinion for simplicity. One of the relevant characteristics of opinions is the one-to-one mapping that is possible to delineate between them and a probability distribution family named beta distribution. The components of an opinion can be expressed according to the parameters of the beta distribution, and vice versa. The generalization to multinomial opinions makes use of the Dirichlet distribution. There also exists a triangular-shaped graphical representation of the opinion space in which we associate vertices with the first components b, d, u and the base with the prior probability.

Now that we have described opinions and their theoretical background, how to operate with them? As one might expect, the first operators are a generalization of operators taken from binary logic and probability theory. Therefore, we can define for example conjunctions and disjunctions. Let's consider the first case. We have ω_x^A and ω_y^A, two opinions about states (x and y) of two binary frames of discernments (each of which contains one state and its negation), and we want to express the opinion about $x \wedge y$. Then we define the conjunction operator $\omega_{x \wedge y}^A = \omega_x^A \cdot \omega_y^A$ (called multiplication in the context of Subjective Logic). The resulting opinion is a tuple $(b_x b_y, d_x + d_y - d_x d_y, ...)$. The components are computed following the semantics of the frame of discernment computations: in this case, briefly, we construct a product frame of discernment taking the cartesian product of the two initial ones, we re-compute the belief mass of each subset and then we obtain the four components using the respective definitions (exploiting the focused framed of discernment composed by $x \cap y$ and its negation). Consider, for example, a sensor s that monitors the correct functioning of two industrial machines x and y. We need both machines to work as expected for our process z. In this case, having collected the sensor's ω_x^s and ω_y^s, we can derive the conjunction $\omega_{x\wedge y}^s$.

The whole reasoning process can be repeated for all the operators (disjunction, negation, union, difference, ...). As we wanted, all operators have similar behavior to their traditional counterparts when applied to opinions reflecting "logical" operands (for example, when we have $b = 1$ and no uncertainty). On the contrary, some properties of the operands do not hold anymore. One interesting consequence of applying operators is the possibility to perform computations that would be difficult in other contexts: operating on beta distributions can be complex (even intractable) and with no "closure" property, while passing through Subjective Logic we always have opinions approximating the exact distribution. Operators that handle a more convoluted piece of computation are deduction and abduction, useful to perform the basic conditional reasoning tasks and a generalized version of the Bayes theorem (elements so pervasive in probability theory).

Subjective Logic introduces also new operators, which are not borrowed from well-known formalisms. Their function is to manage the peculiarity of considering opinions coming from multiple sources. Let B an agent with an opinion about x and let A another agent with an opinion about how much B is reliable. A can make an opinion about x taking into consideration these beliefs. This is called discounting, and it is written $\omega_x^{A;B} = \omega_B^A \otimes \omega_x^B$. Other operators that emerge naturally are the fusions operators. They are useful to combine the opinions that two (or more) agents have about the same fact, diminishing uncertainty or summing up two different points of view. Depending on the definition, a fusion $\omega_x^{A\diamond B} = \omega_x^A \oplus \omega_x^B$ can be cumulative or averaging.

All these operators find their natural application context on the already mentioned Bayesian networks, trust networks and flow-based reputation systems. Here is an example of a subjective trust network that includes the new operators introduced. Consider three witnesses W_1, W_2 and W_3 in a courtroom that are giving testimony with respect to an event x. Their opinions are collected and denoted by $\omega_x^{W_i}$. The judge J has his own opinions about how much each witness is reliable, and those opinions are $\omega_{W_i}^J$. Using the discounting operator, we can compute the opinions that the judge has about x taking into consideration each testimony and the related witness' reliability, that is $\omega_x^{JW_i} = \omega_{W_i}^J \otimes \omega_x^{W_i}$. Assuming that the opinions are independent, they can finally be combined using the consensus operator to produce the judge's final opinion about x, that is

$$\omega_x^{J(W_1,W_2,W_3)} = (\omega_{W_1}^J \otimes \omega_x^{W_1}) \oplus (\omega_{W_2}^J \otimes \omega_x^{W_2}) \oplus (\omega_{W_3}^J \otimes \omega_x^{W_3})$$

Chapter 5

Implementation

To put into practice what we have discussed, it is undoubtedly interesting to implement the most important algorithms and strategies, showing how they work in depth and how we can untangle the possible tricky aspects of dealing with real cases. For this purpose, Google Colab and Python have been chosen as tools for their flexibility in the development, but it is straightforward the use of other programming languages with minor changes that do not affect in any way the structure and the logic behind the program. This appendix is aimed to explain the design that underlies the code[1], which anyway is fully commented and runnable independently with respect to this report.

5.1 Classes and Functions

The first section is composed of all classes and basic functions that will be useful for the rest of the program. In detail, we use four classes: CNF, Clause, Literal and Interpretation. CNF has an attribute that is a list of clauses, Clause has a list of literals, and Literal has two attributes, respectively the name of the variable and its polarity (if it is negated or not). These structures can reproduce any propositional formula. Obviously, we need also a way to keep track of the assignment, and this is covered by the class Interpretation. It has a dictionary to map each variable to the corresponding assignment, plus a list named fixed which is used to store the unchangeble variables that have already been "correctly" assigned. Even if not strictly necessary in Python, all classes are provided by getter and setter methods. In addition, Interpretation has a function flipAssignment, and Literal has functions to check if two literals are equal or opposed to each other, and one function to evaluate the truth value under a given interpretation passed as input.

[1] Colab Notebook

```
# Objects of class CNF represent formulas in Conjunctive Normal
    Form.
# The attribute clauses is a list of object of class Clause.
class CNF:
  def __init__(self, clauses):
    self.clauses = clauses

  def setClauses(self, clauses):
    self.clauses = clauses

  def addClause(self, clause):
    self.clauses.append(clause)

  def getClauses(self):
    return self.clauses

  def printCNF(self):
    i = 0
    for C in self.getClauses():
      i += 1
      j = 0
      print("Clause ", i)
      for x in C.getLiterals():
        j += 1
        print("Literal ", j)
        print(x.name, x.polarity)

# Objects of class Clause represent disjunctions.
# The attribute literals is a list of object of class Literal.
class Clause:
  def __init__(self, literals):
    self.literals = literals

  def setLiterals(self, literals):
    self.literals = literals

  def addLiteral(self, lit):
    self.literals.append(lit)

  def getLiterals(self):
    return self.literals

  def equalTo(self, C):
    if len(self.literals) != len(C.literals):
      return False

    for x1 in self.literals:
      is_present = False
      for x2 in C.literals:
        if x1.equalTo(x2):
          is_present = True
      if not is_present:
        return False

    return True

# Objects of class Literal represent a variable and its polarity.
# If the polarity is True, the literal is non-negated, otherwise
    it is negated.
class Literal:
  def __init__(self, name, polarity):
    self.name = name
    self.polarity = polarity

  def getName(self):
    return self.name

  def getPolarity(self):
    return self.polarity

  def getValue(self, I): # getValue returns the truth value of the
      literal given an interpretation
    return ((not I.getAssignment().get(self.name) or self.polarity) and (
        not self.polarity or I.getAssignment().get(self.name)))

  def equalTo(self, lit):
    return self.name == lit.name and self.polarity == lit.polarity

  def oppositeOf(self, lit):
    return self.name == lit.name and self.polarity == (not lit.polarity)

# The attribute assignment is a dict: keys are variables and
    values are truth assignment to related variables.
# The attribute fixed is a list of variables for which we already
     have the best assignment and it must be not modified.
class Assignment:
  def __init__(self, assignment, fixed):
    self.assignment = assignment
    self.fixed = fixed

  def getFixedVars(self):
    return self.fixed

  def addFixedVar(self, var):
    self.fixed.append(var)

  def setFixedVars(self, fixed_vars):
    self.fixed = fixed_vars

  def flipAssignment(self, var):
    self.assignment.update({var: (not self.assignment.get(var))})

  def modifyAssignment(self, var, val):
    self.assignment.update({var: val})

  def setAssignment(self, assignment):
    self.assignment = assignment

  def getAssignment(self):
    return self.assignment

  def serialize(self):
    string = ''.join(map(str, map(int, self.assignment.values())))
    return string

  def deserialize(self, string):
    i = 0
    for var in self.assignment.keys():
      self.assignment.update({var: bool(int(string[i]))})
      i += 1
```

The functions defined in this first section are the fundamental modules to be composed in more complex algorithms. The function randomInterpretation gives a random assignment to the variables (except the fixed ones), and it is useful to execute the random restarts at the beginning of GSAT and WalkSAT. Then, we have the functions to check if a clause or an entire formula is satisfied, and to count how many clauses are satisfied inside the formula under a certain interpretation.

```
# It generates a random assignment for all the variables but
    those in fixed
def randomInterpretation(I):
  for var in I.getAssignment().keys():
    if var not in I.getFixedVars():
      I.getAssignment().update({var: random.choice([True, False])})

# It checks if a clause is satisfied under an interpretation (at
    least one literal must be True)
def isClauseSatisfied(C, I):
  for x in C.getLiterals():
    if x.getValue(I):
      return True
  return False

# It checks if a clause is satisfied under an interpretation
```

```
    restricted to those variables that are fixed
def isClauseSatisfiedByFixedVars(C, I):
  for x in C.getLiterals():
    if x.getValue(I) and x.getName() in I.getFixedVars():
      return True
  return False

# It checks if a clause is unsatisfiable under an interpretation
    restricted to those variables that are fixed
def isClauseUnsatisfiableByFixedVars(C, I):
  for x in C.getLiterals():
    if x.getName() not in I.getFixedVars():
      return False
    elif x.getValue(I):
      return False
  return True

# It counts the number of satisfied clauses under an
    interpretation
def numberSatisfiedClauses(cnf, I):
  res = 0
  for C in cnf.getClauses():
    if isClauseSatisfied(C, I):
      res += 1
  return res

# It counts the number of satisfied clauses under an
    interpretation restricted to those variables that are fixed
def numberSatisfiedClausesByFixedVars(cnf, I):
  res = 0
  for C in cnf.getClauses():
    if isClauseSatisfiedByFixedVars(C, I):
      res += 1
  return res

# It counts the number of unsatisfiable clauses under an
    interpretation restricted to those variables that are fixed
def numberUnsatisfiableClausesByFixedVars(cnf, I):
  res = 0
  for C in cnf.getClauses():
    if isClauseUnsatisfiableByFixedVars(C, I):
      res += 1
  return res

# It checks if a formula is satisfied under an interpretation (
    all clauses must be True)
def isFormulaSatisfied(cnf, I):
  return numberSatisfiedClauses(cnf, I) == len(cnf.getClauses())
```

Finally, there are two more functions that will constitute the backbone of the two local search algorithms implemented, namely pickUnsatisfiedClause (to choose randomly one among the unsatisfied clauses in WalkSAT) and bestFlip (to identify the best successor interpretation that satisfies the most clauses possible). The latter uses a list of variables as a parameter to specify a restriction on the entire set of variables on which we perform the flip: this is useful in WalkSAT, where the variables to be considered must belong to the chosen unsatisfied clause.

```
# It chooses a random clause among the unsatisfied ones in a
    formula
def pickUnsatisfiedClause(cnf, I):
  unsatisfiedClauses = []

  for C in cnf.getClauses():
    if not isClauseSatisfied(C, I):
      unsatisfiedClauses.append(C)

  return random.choice(unsatisfiedClauses)

# It checks how many clauses are satisfied after flipping a
    variable and performs the best flip (ties are broken randomly
    ).
# The list of variables to use is a param, when unspecified we
    consider all the variables (in both cases, we filter out
    fixed ones).
def bestFlip(cnf, I, vars=[]):
  max_sat = numberSatisfiedClauses(cnf, I)
  best_var = []

  if vars == []:
    vars = I.getAssignment().keys()

  vars = list(filter(lambda var: var not in I.getFixedVars(), vars))

  for var in vars:
    I.flipAssignment(var)
    num_sat = numberSatisfiedClauses(cnf, I)

    if num_sat > max_sat:
      max_sat = num_sat
      best_var = [var]
    elif num_sat == max_sat:
      best_var.append(var)

    I.flipAssignment(var)

  if best_var != []:
    flip_var = random.choice(best_var)
    I.flipAssignment(flip_var)
```

5.2 Preprocessing

5.2.1 Prefix Notation

The scope of the code is to get a hard grasp on the subject in question. It means that it would be too simplistic to just implement directly the local search algorithms. For this reason, we have imagined how a piece of real working software should behave, and we tried to realize the whole workflow. Therefore, the first step is to take as input a general string representing a propositional formula, on which we do not make any assumption at all. In fact, it would be irrealistic to ask a user translating the formula into the CNF form (it can be quite non-trivial and time-consuming even for short formulas with few variables), or to ask whatever kind of strict notation (we can imagine that input to this code is the output of other processes).

This is why the first functions defined in this section are uniformOperatorNotation, which changes boolean operators expressed in all the most common notations in order to uniform them with respect to a unique chosen one, and checkInputValidity, useful for checking the syntactic validity of the input. Differently from what we have done, this last step can also be implemented by traducing directly the grammar:

$$\begin{aligned}
&F ::= p\\
&F ::= (F)\\
&F ::= \neg F\\
&F ::= F\; op\; F\\
&op ::= \wedge \mid \vee \mid \rightarrow \mid \leftrightarrow \mid \oplus \mid \uparrow \mid \downarrow
\end{aligned}$$

However, the function checkInputValidity allows us to significantly reduce the number of ”production” functions (with a short and manageable cascade of if-then cases), and especially it perfectly fits the following code and the strong modular approach of the sections in which the code is divided.

```
# It homologates different representations for the operators in a
     single notation.
def uniformOperatorNotation(string):
  string = re.sub('nor', '\downarrow', string)
  string = re.sub('nand', '\uparrow', string)
  string = re.sub('iff|if and only if|<->|<=>', '\leftrightarrow', string)
  string = re.sub('implies|->|=>', '\rightarrow', string)
  string = re.sub('xor|\^', '\oplus', string)
  string = re.sub('and|\*|&', '\wedge', string)
  string = re.sub('or|\|\||\+|\|', '\vee', string)
  string = re.sub('not|~|-', '\neg', string)
  return string

# It returns False if the formula is ill-formed with respect to
    the syntax.
def checkInputValidity(string, stack, i):
  if (string[i] in symbols):
    if (i == 0 or string[i-1] == ')' or string[i-1] in bin_symbols):
      return False
  if (string[i] in bin_symbols and i == len(string)-1):
    return False
  if (string[i] == ')'):
    if (i == len(string)-1 or (i != 0 and string[i-1] in un_symbols)):
      return False
  if (string[i] == '('):
    if (i == 0 or ')' not in stack or string[i-1] in bin_symbols):
      return False
  return True
```

Finally, we have one of the core functions, namely infixToPrefix. It takes the formula as input and turns it into prefix notation (also known as Polish notation). Prefix notation consists in placing an operator before its operands (regardless of the fact that the operator is unary or binary). This is way less convenient for human readability with respect to the more common infix notation, but it has other advantages, such as the possibility of evaluating a formula in linear order using queues (or stacks) and the removal of possible ambiguities derived from the execution of the operators. These ambiguities are often solved using parentheses and conventionally established operator priorities. Prefix notation allows us to get rid of them both. Given the fact that our objective is to have a formula in CNF form, prefix notation can be seen as the perfect pre-processing step, because the peculiar placement of the operands turns out to be so helpful in later transformation.

```
# It turns the formula into prefix notation, using a stack to
    manage operators (with priority and parentheses) while
    scanning the formula.
# It return the formula as a list and a random interpretation for
    the variables (it returns None if something goes wrong).
def infixToPrefix(string):
  string = uniformOperatorNotation(string).replace(" ", "")[::-1]

```

```
regex = re.compile("^[^(\neg|\wedge|\uparrow|\vee$|\downarrow|\
    rightarrow|\leftrightarrow|\oplus|\(|\))]*")
stack = []

I = Assignment({}, [])
res = ''
i = 0

if len(re.findall("\(", string)) != len(re.findall("\)", string)):
  return None

while i < len(string):
  if not checkInputValidity(string, stack, i):
    return None

  operand = re.search(regex, string[i:]).group(0)

  if operand != '':
    I.getAssignment().update({operand[::-1]: random.choice([True, False])
        })
    res += operand + ' '
    i += len(operand)
  elif string[i] in symbols:
    while (stack != [] and getOperatorPriority(string[i]) <
        getOperatorPriority(stack[-1])):
      res += stack[-1] + ' '
      stack.pop()
    stack.append(string[i])
    i += 1
  elif string[i] == ')':
    stack.append(string[i])
    i += 1
  elif string[i] == '(':
    while (stack[-1] != ')'):
      res += stack[-1] + ' '
      stack.pop()
    stack.pop()
    i += 1

for op in stack[::-1]:
  res += op + ' '

return res[::-1].strip().split(' '), I
```

5.2.2 CNF

Now it is possible to make the CNF conversion of the formula taken as input. How can we do that? The function convertToCNF is a recursive function in which we complete the construction in a bottom-up fashion. When a variable is encountered, we directly return a CNF object with one clause and just one literal, that is the variable itself. For what concern the operators, we use an external function operatorHandler for the case of $\wedge$ and $\vee$, and we reconduct all the other cases to these two "base cases". Anyway, in every situation we identify the operands to which the operator is applied calling recursively convertToCNF, and returning an extra integer to recognize the end of one operand and the beginning of another one within the input (that is an undistinct "flow" of element).

```python
# It takes a formula expressed in prefix notation (as a list) and
    returns the CNF (as an object of the class CNF).
# It is a recursive function that handles each propositional
   operator (negated case are treated separately).
# The index returned with the CNF formula is useful at each step
   to discriminate where an operand ends within the list.
def convertToCNF(string, i):
  if string[i] not in symbols:
    return CNF([Clause([Literal(string[i], True)])]), i
  if string[i] == '\wedge' or string[i] == '\vee':
    operand1, last = convertToCNF(string, i+1)
    operand2, last = convertToCNF(string, last+1)
    return operatorHandler(string[i], operand1, operand2), last
  if string[i] == '\rightarrow':
    string[i] = '\vee'
    string.insert(i+1, '\neg')
    operand1, last = convertToCNF(string, i+1)
    operand2, last = convertToCNF(string, last+1)
    return operatorHandler(string[i], operand1, operand2), last
  if string[i] == '\leftrightarrow':
    string[i] = '\wedge'
    string.insert(i+1, '\vee')
    operand1, last1 = convertToCNF(string, i+2)
    string.insert(last1+1, '\neg')
    operand2, last2 = convertToCNF(string, last1+1)
    first_half = operatorHandler(string[i+1], operand1, operand2)
    string[last2+1:last2+1] = ['\vee', '\neg'] + string[i+2:last1+1] +
        string[last1+2:last2+1]
    second_half, last = convertToCNF(string, last2+1)
    return operatorHandler(string[i], first_half, second_half), last
  if string[i] == '\oplus':
    string[i] = '\neg'
    string.insert(i+1, '\leftrightarrow')
    return convertToCNF(string, i)
  if string[i] == '\uparrow':
    string[i] = '\neg'
    string.insert(i+1, '\wedge')
    return convertToCNF(string, i)
  if string[i] == '\downarrow':
    string[i] = '\neg'
    string.insert(i+1, '\vee')
    return convertToCNF(string, i)
  if string[i] == '\neg':
    if string[i+1] not in symbols:
      return CNF([Clause([Literal(string[i+1], False)])]), i+1
    if string[i+1] == '\neg':
      return convertToCNF(string, i+2)
    if string[i+1] == '\wedge':
      string[i] = '\vee'
      string[i+1] = '\neg'
      operand1, last = convertToCNF(string, i+1)
      string.insert(last+1, '\neg')
      operand2, last = convertToCNF(string, last+1)
      return operatorHandler(string[i], operand1, operand2), last
    if string[i+1] == '\vee':
      string[i] = '\wedge'
      string[i+1] = '\neg'
      operand1, last = convertToCNF(string, i+1)
      string.insert(last+1, '\neg')
      operand2, last = convertToCNF(string, last+1)
      return operatorHandler(string[i], operand1, operand2), last
    if string[i+1] == '\rightarrow':
      string[i] = '\wedge'
      del string[i+1]
      operand1, last = convertToCNF(string, i+1)
      string.insert(last+1, '\neg')
      operand2, last = convertToCNF(string, last+1)
      return operatorHandler(string[i], operand1, operand2), last
    if string[i+1] == '\leftrightarrow':
      string[i] = '\wedge'
      string[i+1] = '\vee'
      operand1, last1 = convertToCNF(string, i+2)
      operand2, last2 = convertToCNF(string, last1+1)
      first_half = operatorHandler(string[i+1], operand1, operand2)
      string[last2+1:last2+1] = ['\vee', '\neg'] + string[i+2:last1+1] + ['
          \neg'] + string[last1+1:last2+1]
      second_half, last = convertToCNF(string, last2+1)
      return operatorHandler(string[i], first_half, second_half), last
    if string[i+1] == '\oplus':
      string[i] = '\leftrightarrow'
      del string[i+1]
      return convertToCNF(string, i)
    if string[i+1] == '\uparrow':
      string[i] = '\wedge'
      del string[i+1]
      return convertToCNF(string, i)
    if string[i] == '\downarrow':
      string[i] = '\vee'
      del string[i+1]
      return convertToCNF(string, i)
```

In detail, we use the following equivalences:

$$
\begin{aligned}
\neg\neg A &\equiv A \\
\neg(A \wedge B) &\equiv \neg A \vee \neg B \\
\neg(A \vee B) &\equiv \neg A \wedge \neg B \\
A \vee A &\equiv A \\
A \wedge A &\equiv A \\
A \vee \neg A &\equiv True \\
A \vee (B \wedge C) &\equiv (A \vee B) \wedge (A \vee C) \\
A \rightarrow B &\equiv \neg A \vee B \\
A \leftrightarrow B &\equiv (\neg A \vee B) \wedge (A \vee \neg B) \\
A \oplus B &\equiv \neg(A \leftrightarrow B) \\
A \uparrow B &\equiv \neg(A \wedge B) \\
A \downarrow B &\equiv \neg(A \vee B)
\end{aligned}
$$

The operand $\wedge$ takes two CNF objects as operands and returns a CNF object in which the list of clauses is just the concatenation of the initial lists of clauses. The operand $\vee$ is handled using the distributive law. We take into consideration the simplifications derived from idempotent laws for $\vee$ and $\wedge$, and from complement law for $\vee$. We avoid applying the idempotent law for $\wedge$ because it would preclude us from carrying out early stopping analysis and, in case, expanding the program implementing algorithms for Max-SAT.

```
# It takes as input an operator \wedge or \vee and two CNF
    formulas, and performs the operator on them returning the
    resulting CNF formula.
# The \vee case deletes a literal if it is duplicated in a clause
    , and deletes a clause if it contains opposite literals.
def operatorHandler(operator, operand1, operand2):
  if operator == '\wedge':
    clauses = []
    for C1 in operand1.getClauses():
      same_clause = False
      for C2 in operand2.getClauses():
        if C1.equalTo(C2):
          same_clause = True
      if not same_clause:
        clauses.append(C1)
    clauses = clauses + operand2.getClauses()
    return CNF(clauses)
  elif operator == '\vee':
    clauses = []
    for C1 in operand1.getClauses():
      for C2 in operand2.getClauses():
        clause = []
        del_clause = False
        for x1 in C1.getLiterals():
          same_lit = False
          for x2 in C2.getLiterals():
            if x1.equalTo(x2):
              same_lit = True
            if x1.oppositeOf(x2):
              del_clause = True
              break
          if not same_lit:
            clause.append(x1)
          if del_clause:
            break
        if not del_clause:
          clauses.append(Clause(clause + C2.getLiterals()))
    return CNF(clauses)
```

5.2.3 Random instances

We can now enter a formula and turn it into Conjunctive Normal Form. Anyway, even if this method gives more possibility to the user, it is often unpractical to test the following algorithms against long and complex formulas. If the given input is too easy to solve, or humanly biased, we could not appreciate the evolution and the struggle of the computation. Furthermore, real case scenarios are way too compounded than what a user might think. This is why, inspired by the benchmarks of the annual International SAT Competition, we add a function to randomly create formulas having a large number of variables and clauses. It is possible to state also how many literals must be in each clause (useful to create k-SAT problems) and to choose if this number is an upper bound or the exact number of literals. An alternative way could be creating a function to read directly the mentioned benchmarks.

```python
# It generates random CNF formulas with n_vars (number of
    variables) and n_clauses (number of clauses).
# Last two parameters are used to establish how many literals a
    clause contain and whether this is a strict requirement.
def genRandomInstances(n_vars, n_clauses, k_lit_in_C=3, k_exact=True):
  cnf = CNF([])
  I = Assignment({}, [])

  for i in range(n_clauses):
    clause = Clause([])
    k = k_lit_in_C if k_exact else random.randint(1, k_lit_in_C)

    vars = [str(var) for var in range(1, n_vars+1)]

    for j in range(k):
      var_name = random.choice(vars)
      vars.remove(var_name)

      lit = Literal(var_name, random.choice([True,False]))
      clause.addLiteral(lit)

    cnf.addClause(clause)

  for var in range(1, n_vars+1):
    I.getAssignment().update({str(var): random.choice([True, False])})

  return cnf, I
```

5.3 Unit Propagation and Pure Literals Rule

Unit propagation and Pure Literals Rule are often associated with DPLL rather than local search methods. This is due to the different nature of the two approaches: they use respectively an incremental-state formulation and a complete-state formulation for the interpretations. It leads to the possibility, for DPLL, of re-computing both Unit propagation and Pure Literals Rule at each recursive call. So why use them here?

A simple consideration is that, even if we cannot call these two techniques during local searches, nothing prevents us from using them as a pre-processing procedure. In this way, we hope to set permanently the assignment of as many variables as possible before the execution of the real search. It has a double benefit. First, it guides the following algorithm, restricting the number of variables to be assigned, and in turn the search space to be explored. Then, in a lucky situation, it can also find a satisfying interpretation before the actual search, and cause an early stopping of the entire program. Obviously, in all real cases of complex instances, this idea becomes quickly useless. It does not mean that, in some of the state-of-the-art algorithms, these simple ideas are not exploited in a more compound picture to speed up the computation and obtain effective strategies.

Both Unit propagation and Pure Literals Rule are implemented using a similar line of action. We ”scan” the CNF formula to check if we can apply them. In positive cases, we execute the simplification, we add the variable to the list fixed, and we call recursively the function. In fact, the application of these rules can lead to the formation of a new situation where the same rules can be applied a second time.

```python
# It performs Unit Propagation on the formula until it’s possible
    .
# If there is a clause with only a literal, the function set the
    assignment of that literal to satisfy the clause.
# If an assignment is performed, the variable involved is added
    to fixed list and the function is called recursively.
def unitPropagation(cnf, I):
  new_assignment = False
  print("Unit propagation")
  print("Fixed vars:")
  print(I.getFixedVars())

  for C in cnf.getClauses():
    lit = []
    satisfied = False

    if len(I.getFixedVars()) == len(I.getAssignment().keys()):
      break

    for x in C.getLiterals():
      if x.getName() in I.getFixedVars() and x.getValue(I):
        satisfied = True
      elif x.getName() not in I.getFixedVars():
        lit.append(x)

    if len(lit) == 1 and not satisfied:
      I.modifyAssignment(lit[0].getName(), lit[0].getPolarity())
      I.addFixedVar(lit[0].getName())
      new_assignment = True

  if new_assignment:
    unitPropagation(cnf, I)
```

```python
# It performs Pure Literals Rule on the formula until it’s
```

```
    possible.
# First, the clauses is filtered because we won't consider the
    already satisfied ones (satisfied by the variables in fixed).
# We maintain a polarity_dict to check if a var appears always
    with the same polarity, in which case that var is assigned
    consistently.
# If an assignment is performed, the variable involved is added
    to fixed list and the function is called recursively.
def pureLiterals(cnf, I):
  print("Pure Literals")
  print("Fixed vars:")
  print(I.getFixedVars())

  polarity_dict = {}
  new_assignment = False

  clauses = cnf.getClauses()
  sat_clauses = []

  for C in clauses:
    for x in C.getLiterals():
      if x.getName() in I.getFixedVars() and x.getValue(I) and C not in
          sat_clauses:
        sat_clauses.append(C)

  clauses = list(filter(lambda C: C not in sat_clauses, clauses))

  for C in clauses:
    for x in C.getLiterals():
      if x.getName() in polarity_dict:
        if x.getPolarity() != polarity_dict.get(x.getName()):
          polarity_dict.update({x.getName(): None})
      else:
        polarity_dict.update({x.getName(): x.getPolarity()})

  for pair in polarity_dict.items():
    if len(I.getFixedVars()) == len(I.getAssignment().keys()):
      break

    if pair[1] != None and pair[0] not in I.getFixedVars():
      I.modifyAssignment(pair[0], pair[1])
      I.addFixedVar(pair[0])
      new_assignment = True

  if new_assignment:
    pureLiterals(cnf, I)
```

5.4 SAT Algorithms

5.4.1 GSAT

Now, we can get into the core of the implementation. Here, we translate what we presented in previous chapters, exploiting the function defined before.

The first function is GSAT. Its main step is performed by bestFlip, which gives us the best interpretation among the successors. The computation is bounded by the parameters max_tries and max_flip. If we use small values, we can remain stuck in a local minimum or, even worse, we can never reach a minimum at all.

```
# It implements the base variant of GSAT algorithm.
# The variables max_restart and max_flips control respectively
    outer and inner loops.
# The core step (greedy hill-climbing) is performed by the
    function bestFlip.
def GSAT(cnf, I, max_restart, max_flips):
  for i in range(max_restart):
    if i == 0:
      print("Starting GSAT...")
    else:
      print("Restarting...")

    randomInterpretation(I)

    print("Number of satisfied clauses {}".format(numberSatisfiedClauses(
        cnf, I)))

    if isFormulaSatisfied(cnf, I):
      return True

    for j in range(max_flips):
      bestFlip(cnf, I)

      print("Iteration {} - Number of satisfied clauses {}".format(j,
          numberSatisfiedClauses(cnf, I)))
      if isFormulaSatisfied(cnf, I):
        return True

  return False
```

5.4.2 WalkSAT

WalkSAT shares most of the parameters with GSAT. This time we can choose also the probability associated with the random walk. Again, the main step is performed by bestFlip, but now we restrict the set of variables to those inside a randomly-picked unsatisfied clause. Probabilistically, we let the process flip a variable that is not necessarily the best one.

```python
# It implements the base variant of WalkSAT algorithm.
# As for GSAT, the variables max_restart and max_flips control respectively outer and inner loops.
# On the contrary wrt GSAT, bestFlip is called only in some iterations (probabilistically) and only on variables from a chosen clause.
# The clause is chosen among the unsatisfied ones by the function pickUnsatisfiedClause
def WalkSAT(cnf, I, prob, max_restart, max_flips):
  for i in range(max_restart):
    if i == 0:
      print("Starting WalkSAT...")
    else:
      print("Restarting...")

    randomInterpretation(I)

    print("Number of satisfied clauses {}".format(numberSatisfiedClauses(cnf, I)))

    if isFormulaSatisfied(cnf, I):
      return True

    for j in range(max_flips):
      C = pickUnsatisfiedClause(cnf, I)

      if random.random() < prob:
        lits = [x for x in C.getLiterals() if x.getName() not in I.getFixedVars()]
        if lits != []:
          x = random.choice(lits)
          I.flipAssignment(x.getName())
      else:
        vars = [x.getName() for x in C.getLiterals()]
        bestFlip(cnf, I, vars)

      print("Iteration {} - Number of satisfied clauses {}".format(j, numberSatisfiedClauses(cnf, I)))
      if isFormulaSatisfied(cnf, I):
        return True

  return False
```

5.4.3 SASAT

SASAT is the implementation of the Simulated Annealing approach to the SAT problem. Here, there are two variants. The first one is a base Simulated Annealing process controlled by temperature parameters (max_temp and min_temp). In each iteration, we compute the gain obtained by flipping the interpretation of a variable and, with a certain probability, we choose whether to flip it or not. The second variant includes a random walk step in the choice of the assignment, combining the idea behind both SASAT and WalkSAT.

```python
# It implements the base variant of Simulated Annealing algorithm
    .
# The variables max_restart and min_temp control respectively
    outer and inner loops.
# Temperature starts by max_temp and degrades with time passing
    by depending on the decay rate.
# For each variable, its assignment is probabilistically flipped
    based on temperature and delta (increasing number of
    satisfied clauses).
def SASAT(cnf, I, max_restart, max_temp, min_temp, decay_rate):
  for i in range(max_restart):
    if i == 0:
      print("Starting SASAT...")
    else:
      print("Restarting...")

    randomInterpretation(I)

    print("Number of satisfied clauses {}".format(numberSatisfiedClauses(
        cnf, I)))

    if isFormulaSatisfied(cnf, I):
      return True

    temp = max_temp
    j = 0

    while temp > min_temp:
      # Update temperature
      temp = max_temp * math.exp(-1*j*decay_rate)

      for var in I.getAssignment().keys():
        if var not in I.getFixedVars():
          # Compute for each (non-fixed) var the gain (delta)
          num_sat = numberSatisfiedClauses(cnf, I)

          I.flipAssignment(var)
          delta = numberSatisfiedClauses(cnf, I) - num_sat
          # Compute probability and hold the new interpretation according to that value
          prob = 1 / (1 + math.exp((-1*delta) / temp))
          if random.random() > prob:
            I.flipAssignment(var)

      print("Iteration {} - Number of satisfied clauses {}".format(j, numberSatisfiedClauses(cnf, I)))
      if isFormulaSatisfied(cnf, I):
        return True
      j += 1

  return False

# It implements a variant of Simulated Annealing algorithm with Random Walks.
# In this case, with a certain probability we use simulated annealing approach;
# Otherwise, we flip the variable if the gain is greater than zero.
def SASAT_RW(cnf, I, prob, max_restart, max_temp, min_temp, decay_rate):
  for i in range(max_restart):
    if i == 0:
      print("Starting SASAT...")
    else:
      print("Restarting...")

    randomInterpretation(I)

    print("Number of satisfied clauses {}".format(numberSatisfiedClauses(cnf, I)))

    if isFormulaSatisfied(cnf, I):
      return True

    temp = max_temp
    j = 0

    while temp > min_temp:
      # Update temperature
      temp = max_temp * math.exp(-1*j*decay_rate)

      for var in I.getAssignment().keys():
        if var not in I.getFixedVars():
          # Compute for each (non-fixed) var the gain (delta)
          num_sat = numberSatisfiedClauses(cnf, I)

          I.flipAssignment(var)
          delta = numberSatisfiedClauses(cnf, I) - num_sat

          # Hold the new interpretation according to probabilities
             obtained by random walk and delta
          if random.random() < prob:
            if random.random() > 1 / (1 + math.exp((-1*delta) / temp)):
              I.flipAssignment(var)
          else:
            if delta < 0:
              I.flipAssignment(var)

      print("Iteration {} - Number of satisfied clauses {}".format(j,
          numberSatisfiedClauses(cnf, I)))
      if isFormulaSatisfied(cnf, I):
        return True
      j += 1

  return False
```

5.4.4 Genetic SLS

The next algorithm is Genetic SLS. It is an interpretation of the Genetic Algorithm methods used in conjunction with stochastic local search. As explained in the dedicated chapter, there exist plenty of different ways to construct a Genetic Algorithm depending on the design choices made in each peculiar phase. Our implementation starts with a population of num_pop different interpretations. In order to manage a bit-string representation, two auxiliary methods are added to the Assignment class respectively for the serialization and the deserialization of an interpretation in a string of bits. Each interpretation is associated with its related fitness function, which gives us a measure of "how good" it is (how many clauses are satisfied under that interpretation). Then, we start the actual algorithm.

First, we perform the parent selection for the reproduction step. An interpretation is selected proportionally with its fitness function, and we implement this idea through a so-called roulette wheel selection exploiting the cumulative values of fitness values. After selecting two parents, we perform the crossover and generate a "child" string, randomly choosing the genes (bit) from the parents. Finally, with a certain probability (mut_prob) we include mutations inside the child string to add an entropy factor and move forward the exploration of the search space. At the end of the reproduction phase, we can perform the stochastic local search starting from the new generated interpretation and using the function bestFlip until it gives us positive results. Repeating this procedure for each child produces a new population of interpretations that can replace the old ones. During the replacement, we consider an elitism mechanism that keeps the best interpretations from the old population and discards the worst interpretations from the new population.

```python
# It implements a Genetic Algorithm with Stochastic Local Search.
# Population is a list of lists: each sublist has three
    components, i.e. a serialized interpretation,
# the related number of satisfied clauses, and the cumulative
    percentage value of fitness function (useful for selection).
def GLS(cnf, I, num_pop, epochs, mutation_prob, elitism_prop):
  print("Starting Genetic Local Search...")

  population = []
  n = 0
  cumulative_fitness = 0.0

  # New population
  for i in range(num_pop):
    randomInterpretation(I)

    if isFormulaSatisfied(cnf, I):
      return True

    num_sat = numberSatisfiedClauses(cnf, I)
    population.append([I.serialize(), num_sat, cumulative_fitness])
    n += num_sat

  for elem in population:
    elem[2] = elem[1] / n + cumulative_fitness
    cumulative_fitness = elem[2]

  # Init epochs
  for epoch in range(epochs):
    print("Epoch {}".format(epoch))
    children = []

    # Generation of new individuals
    for i in range(num_pop):
      parents = []

      # Parent selection
      for j in range(2):
        parent = ''

        select_prob = random.random()

        for elem in population:
          if elem[2] > select_prob:
            parent = elem[0]
            break

        parents.append(parent)

      # Crossover
      child = ''
      for gene in range(len(parents[0])):
        child += random.choice([parents[0][gene], parents[1][gene]])

      # Mutation
      mutation_prob = 0.5
      mut_child = ''

      if random.random() < mutation_prob:
        for gene in range(len(child)):
          if random.random() < 0.5:
            mut_child += str(int(not bool(int(child[gene]))))
          else:
            mut_child += child[gene]
      else:
        mut_child = child

      # SLS for the new individual
      I.deserialize(mut_child)

      num_sat = numberSatisfiedClauses(cnf, I)
      improvement = True

      while improvement:
        bestFlip(cnf, I)

        if numberSatisfiedClauses(cnf, I) > num_sat:
          num_sat = numberSatisfiedClauses(cnf, I)
        else:
          improvement = False

      if isFormulaSatisfied(cnf, I):
        return True

      children.append([I.serialize(), num_sat])

    # Generational replacement with elitism
    population = sorted(population, key=lambda x:x[1], reverse=True)
    children = sorted(children, key=lambda x:x[1], reverse=True)

    print("Number of satisfied clauses {}".format(children[0][1]))

    elitism_prop = 0.2
    del population[int(elitism_prop * num_pop):]
    del children[num_pop - int(elitism_prop * num_pop):]

    n = 0
    cumulative_fitness = 0.0

    for elem in population:
      n += elem[1]

    for elem in children:
      population.append([elem[0], elem[1], cumulative_fitness])
      n += elem[1]

    for elem in population:
      elem[2] = elem[1] / n + cumulative_fitness
      cumulative_fitness = elem[2]

  return False
```

5.5 Max-SAT Algorithms

5.5.1 Johnson's Algorithm

As we have seen, both Max-SAT and Model Counting problems can be solved using a heterogeneous set of techniques. This helps us to make an experience with algorithms that differs a lot from the previous ones and that makes use of probability from a new perspective. The first algorithm implemented is Johnson's Algorithm. It is an approximation algorithm and it is based on a weighting scheme associated with the clauses to be solved. In each iteration, we select a variable and we choose whether to set it to True or False depending on the sum of the weights related to the clauses containing that variable. Once we make the assignment, we delete the satisfied clauses and we update the weights of the other ones.

```
# It implements Johnson's Algorithm for Max-SAT problems.
# The dictionary "weights" stores a list for each clause: the
    first element is the actual weight of the clause;
# the second and the third elements are lists of variables that
    occur respectively with positive and negative polarity in
    that clause.
def Johnson(cnf, I):
  weights = {}

  # Creating weights dictionary
  for C in cnf.getClauses():
    pos_occ = []
    neg_occ = []
    for x in C.getLiterals():
      if x.getPolarity():
        pos_occ.append(x.getName())
      else:
        neg_occ.append(x.getName())

    weights.update({C: [1, pos_occ, neg_occ]})

  # Assignment
  for var in I.getAssignment().keys():
    pos_set = []
    neg_set = []
    pos_w = 0
    neg_w = 0

    # Splitting clauses according to the polarity of current var
        and summing up weights for the two groups
    for elem in weights.items():
      if var in elem[1][1]:
        pos_set.append(elem[0])
        pos_w += elem[1][0]
      elif var in elem[1][2]:
        neg_set.append(elem[0])
        neg_w += elem[1][0]

    # Setting the assignment and adjusting weights dictionary
    if pos_w >= neg_w:
      I.modifyAssignment(var, True)

      for pos_C in pos_set:
        del weights[pos_C]

      for neg_C in neg_set:
        l = weights.get(neg_C)
        l[0] = l[0]*2
        weights.update({neg_C: l})

    else:
      I.modifyAssignment(var, False)

      for neg_C in neg_set:
        del weights[neg_C]

      for pos_C in pos_set:
        l = weights.get(pos_C)
        l[0] = l[0]*2
        weights.update({pos_C: l})
```

5.5.2 Bidirectional Bounds Refinement

The second kind of approximation algorithm is based on the use of lower and upper bounds. In this case, we have implemented a Bidirectional Bounds Refinement method, that consists of the iterative assignment of a variable and the consequent update of the bounds. The lower bound represents the number of satisfied clauses, the upper bound the number of satisfiable clauses. In each iteration, we select a variable and we try both the truth assignments. Depending on how the bounds are modified in the two, we can observe which choice gives us the greatest gain. If both truth assignments are good, we choose probabilistically one of them with a bias on the best one.

```
# It implements a Bidirectional Bounds Refinement algorithm for
    Max-SAT.
# In each iteration, it maintains a mean value to compare which
    truth value must be assigned to a var.
def BBR(cnf, I):
  n = len(cnf.getClauses())
  mean = n / 2

  for var in I.getAssignment().keys():
    I.addFixedVar(var)

    # Compute the new mean and gain when var is True
    I.modifyAssignment(var, True)
    pos_lb = numberSatisfiedClausesByFixedVars(cnf, I)
    pos_ub = n - numberUnsatisfiableClausesByFixedVars(cnf, I)
    pos_mean = (pos_lb + pos_ub) / 2
    pos_gain = pos_mean - mean

    # Compute the new mean and gain when var is False
    I.modifyAssignment(var, False)
    neg_lb = numberSatisfiedClausesByFixedVars(cnf, I)
    neg_ub = n - numberUnsatisfiableClausesByFixedVars(cnf, I)
    neg_mean = (neg_lb + neg_ub) / 2
    neg_gain = neg_mean - mean

    mean = neg_mean

    # Choose between holding the "false" interpretation for var
        or switching to a "true" one (updating mean)
    if neg_gain < 0:
      I.modifyAssignment(var, True)
      mean = pos_mean
    elif pos_gain > 0:
      if random.random() < (pos_gain / (pos_gain + neg_gain)):
        I.modifyAssignment(var, True)
        mean = pos_mean
```

5.5.3 Look-Ahead SLS

Max-SAT can be also solved by using the stochastic local search techniques of SAT problems. To show it, we have presented the idea of multiple flips and here we can implement the Look-Ahead SLS. It is constructed on the top of the GSAT paradigm, but it is likely to be able to escape from local minima and obtain better results for Max-SAT. In each iteration, it tries to execute bestFlip to increase the number of satisfied clauses. When it fails, we randomly store a set of variables, called first-level variables, and we try to flip their assignments iteratively. This can temporarily return a worse interpretation, but then we execute again the bestFlip function, obtaining a double flip (look-ahead) and possibly exploring new areas of the search space.

```
# It implements a GSAT-like SLS with Look-Ahead mechanism.
# num_first_lvl is used to determine how many variables must be
    considered before the actual look-ahead flip.
def LookAhead(cnf, I, max_restart, max_flips, num_first_lvl):
  n = len(cnf.getClauses())
  best_assignment = I.getAssignment()
  max_sat = numberSatisfiedClauses(cnf, I)

  for i in range(max_restart):
    if i == 0:
      print("Starting SLS...")
    else:
      print("Restarting...")

    randomInterpretation(I)
    curr_sat = numberSatisfiedClauses(cnf, I)

    if isFormulaSatisfied(cnf, I):
      return n

    for j in range(max_flips):
      # Perform the best flip
      bestFlip(cnf, I)
      num_sat = numberSatisfiedClauses(cnf, I)

      if num_sat > curr_sat:
        curr_sat = num_sat
      else:
        # If best flip does not increase the number of satisfied
            clauses, try to look-ahead
        curr_assignment = I.getAssignment()
        first_lvl_lit = []

        # Select a set of "first level" literals
        k = 0
        while len(first_lvl_lit) < num_first_lvl and k < 2*num_first_lvl:
          C = pickUnsatisfiedClause(cnf, I)
          lit = random.choice(C.getLiterals())

          if lit not in first_lvl_lit:
            first_lvl_lit.append(lit)

          k += 1

        # Flip a "first level" literal and perform the best flip.
        for x in first_lvl_lit:
          I.flipAssignment(x.getName())

          bestFlip(cnf, I)
          num_sat = numberSatisfiedClauses(cnf, I)

          if num_sat > curr_sat:
            curr_sat = num_sat
            break
          else:
            I.setAssignment(curr_assignment)

      print("Iteration {} - Number of satisfied clauses {}".format(j,
          curr_sat))
      if isFormulaSatisfied(cnf, I):
        return n

    if curr_sat > max_sat:
      best_assignment = I.getAssignment()
      max_sat = curr_sat

  I.setAssignment(best_assignment)
  return max_sat
```

5.6 Model Counting Algorithms

5.6.1 ExactCount

Finally, we present three methods to get solutions for the Model Counting instances. They push us towards a new way of thinking about the problem and experimenting with probability. The first function is actually a deterministic function that counts the number of models for a given formula with a brute force iteration over the interpretations space. It is realized by simulating the scan of the truth table that can be constructed for the input formula. Obviously, this is not useful in practical cases because of the great inefficiency of the brute force approach. Anyway we can exploit this function to get insights, solve the simpler cases and, most importantly, use it as a layer on top of which constructing the following techniques.

```python
# It counts the number of models for a given formula.
def exactCount(cnf, I):
  vars = [v for v in I.getAssignment().keys() if v not in I.getFixedVars()]
  n_vars = len(vars)
  count = 0

  for var in vars:
    I.modifyAssignment(var, False)

  # Simulate the generation of the "rows" present in the truth
      table of the formula
  prec = 0
  for k in range(2**len(vars)):
    for idx, switch in enumerate(format(k ^ prec, "b")):
      if switch == '1':
        I.flipAssignment(vars[n_vars-1 - idx])

    prec = k

    if isFormulaSatisfied(cnf, I):
      count += 1

  return count
```

5.6.2 ApproxCount

When the number of variables is too big, we can't try out all the possible interpretations for a formula. This means we can't come out with the real number of models existing for that formula, and we are forced to think of a way of getting an approximation of the count. ApproxCount does so by exploiting the idea of probabilistic sampling. In each iteration, we sample k satisfying interpretations (solution for the formula) and we select a variable. Then we check how many times the variable is assigned True or False inside the sampled solutions and we set the assignment for that variable, reducing the dimensionality of the problem. We repeat this idea until the formula is not feasible for ExactCount, that is the number of non-fixed variables is quite small. At the end, we return the exact count performed on the remaining formula times the multiplier we have updated for each assignment that tells us how to scale the number of models for the entire formula.

```python
# It implements ApproxCount for the Model Counting problem.
# The multiplier computation is based on at most k solutions (k
    control the quality of the approximation).
def ApproxCount(cnf, I, k):
  multiplier = 1

  n_left_vars = len([v for v in I.getAssignment().keys() if v not in I.
      getFixedVars()])

  # While the formula is not feasible for ExactCount
  while n_left_vars > 20:
    i = 0
    samples = []

    # Sample at most k solutions (satisfying interpretations)
    while len(samples) < k and i < 2*k:
      if random.random() < 0.5:
        sol = WalkSAT(cnf, I, 0.25, 100, 100)
      else:
        sol = SASAT_RW(cnf, I, 0.25, 100, 1.0, 0.1, 0.01)

      if sol and I.getAssignment() not in samples:
        samples.append(I.getAssignment().copy())

      i += 1

    if samples == []:
      break

    # Choose a variable and count how many times it appears
        positive and negative in sampled solutions
    var = random.choice([v for v in I.getAssignment().keys() if v not in I.
        getFixedVars()])

    num_true = 0
    num_false = 0

    for sample in samples:
      if sample.get(var):
        num_true += 1
      else:
        num_false += 1

    # Set the variable and update the multiplier
    if num_true > num_false:
      I.modifyAssignment(var, True)
      multiplier *= len(samples) / num_true
    else:
      I.modifyAssignment(var, False)
      multiplier *= len(samples) / num_false

    # Simplify
    I.addFixedVar(var)
    unitPropagation(cnf, I)
    pureLiterals(cnf, I)

    n_left_vars = len([v for v in I.getAssignment().keys() if v not in I.
        getFixedVars()])

  return multiplier * exactCount(cnf, I)
```

5.6.3 SampleCount

The last proposed method is SampleCount. It shares its core idea with ApproxCount, but it is more robust in returning a correct lower bound on the number of models with high confidence. Again we first sample k solutions from the solution space of the formula, but the variable to be assigned is no more chosen randomly among the left ones. Rather we choose the most balanced variable, that is the variable assigned more equally to True and False in the sampled solutions. The multiplier associated with the most balanced variable is no more computed as the ratio of the number of samples, but it is always approximated to 2. At the end of a count, we repeat the procedure (max_tries times) and we store the best lower bound.

```
# It implements SampleCount for the Model Counting problem.
def SampleCount(cnf, I, k, max_tries):
  lb = 2**len(I.getAssignment().keys())

  # Iterate for max_tries time to adjust the lower bound
  for j in range(max_tries):
    I.setFixedVars([])
    n_left_vars = len(I.getAssignment().keys())
    s = 0

    # While the formula is not feasible for ExactCount
    while n_left_vars > 20:
      s += 1

      # Sample at most k solutions (satisfying interpretations)
      i = 0
      samples = []

      while len(samples) < k and i < 2*k:
        if random.random() < 0.5:
          sol = WalkSAT(cnf, I, 0.25, 100, 100)
        else:
          sol = SASAT_RW(cnf, I, 0.25, 100, 1.0, 0.1, 0.01)

        if sol and I.getAssignment() not in samples:
          samples.append(I.getAssignment().copy())

        i += 1

      if samples == []:
        break

      # Choose a balanced variable (early stop when the balance is
           acceptable)
      left_vars = [v for v in I.getAssignment().keys() if v not in I.
          getFixedVars()]
      balanced_var = left_vars[0]
      balance = len(samples)
      num_true_bal = 0
      num_false_bal = 0

      for var in left_vars:
        num_true = 0
        num_false = 0

        for sample in samples:
          if sample.get(var):
            num_true += 1
          else:
            num_false += 1

        curr_balance = abs(num_true - num_false)

        if curr_balance <= balance:
          num_true_bal = num_true
          num_false_bal = num_false
          balance = curr_balance
          balanced_var = var

        if balance / len(samples) < 0.1:
          break

      # Set the chosen variable's assignment
      if num_true_bal == 0:
        I.modifyAssignment(balanced_var, False)
      elif num_false_bal == 0:
        I.modifyAssignment(balanced_var, True)
      elif random.random() < 0.5:
        I.flipAssignment(balanced_var)

      # Simplify
      I.addFixedVar(balanced_var)
      unitPropagation(cnf, I)
      pureLiterals(cnf, I)

      n_left_vars = len([v for v in I.getAssignment().keys() if v not in I.
          getFixedVars()])

    # Compute the approximated count and adjust the lower bound
    count = 2**s * exactCount(cnf, I)

    if count < lb:
      lb = count

  return lb
```

5.7 Comparison

This section sums up some tests conducted using the described algorithms. In order to avoid totally biased and unreliable results, all tests were repeated multiple times against the same input parameters and we have computed the mean of these values. The objective is not to establish the superiority of one methodology over the others (they can be improved in several directions), but to have a quick insight of possible useful parameters.

It is important to note that each result must be contextualized. Good results can often be achieved through a trade-off with running time, and there are cases, such as Max-SAT algorithms, in which a surprising scalable power is way better than a slightly more precise approximation. Sometimes, as happens with SampleCount, we decide to renounce a better result in order to have higher confidence with respect to the correctness of the result itself.

The input formulas are always randomly generated 3-SAT instances, with 100 variables and 400 clauses. For Model Counting algorithms we have reduced them to have 20 variables and 80 clauses, so that we have had the possibility of performing a preliminary brute-force counting to compare the approximated results from our implementations.

SAT

Algorithm	max_tries	max_flips	rw_prob	Restarts	Time
GSAT	100	100	–	62	299
GSAT	100	200	–	23	233
WalkSAT	100	100	0.25	21	7
WalkSAT	100	100	0.10	33	12

Algorithm	max_tries	decay_rate	rw_prob	Restarts	Time
SASAT	100	0.10	–	9	22
SASAT	100	0.05	–	6	28
SASAT_RW	100	0.10	0.25	14	33
SASAT_RW	100	0.10	0.10	9	21

Algorithm	population	mutation	elitism	Restarts	Time
GLS	30	0.33	0.20	17	348
GLS	20	0.25	0.20	13	187

Max-SAT

Algorithm	max_tries	max_flips	first_lvl_vars	# Sat Clauses
Johnson	–	–	–	390/400
BBR	–	–	–	381/400
Look-Ahead	20	50	1	397/400
Look-Ahead	20	50	10	399/400

Model Counting

Algorithm	num_samples	max_tries	% Models Found
ApproxCount	50	–	0.76
SampleCount	50	5	0.50

Bibliography

[1] Hachemi Bennaceur. "A comparison between SAT and CSP techniques". In: *Constraints* 9.2 (2004), pp. 123–138.

[2] Gregory F Cooper. "The computational complexity of probabilistic inference using Bayesian belief networks". In: *Artificial intelligence* 42.2-3 (1990), pp. 393–405.

[3] Lorenz Demey, Barteld Kooi, and Joshua Sack. "Logic and Probability". In: *The Stanford Encyclopedia of Philosophy.* Ed. by Edward N. Zalta. Summer 2019. Metaphysics Research Lab, Stanford University, 2019.

[4] Marcelo Finger and Glauber De Bona. "Probabilistic satisfiability: Logic-based algorithms and phase transition". In: *Twenty-Second International Joint Conference on Artificial Intelligence.* 2011.

[5] Huimin Fu et al. "An improved genetic algorithm for solving 3-SAT problems based on effective restart and greedy strategy". In: *2017 12th International Conference on Intelligent Systems and Knowledge Engineering (ISKE).* IEEE. 2017, pp. 1–6.

[6] Carla P Gomes, Ashish Sabharwal, and Bart Selman. "Model counting". In: *Handbook of satisfiability.* IOS press, 2008.

[7] Carla P Gomes et al. "From Sampling to Model Counting." In: *IJCAI.* Vol. 2007. 2007, pp. 2293–2299.

[8] Jens Gottlieb, Elena Marchiori, and Claudio Rossi. "Evolutionary algorithms for the satisfiability problem". In: *Evolutionary computation* 10.1 (2002), pp. 35–50.

[9] Audun Jøsang. "A logic for uncertain probabilities". In: *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems* 9.03 (2001), pp. 279–311.

[10] Kalev Kask and Rina Dechter. "A graph-based method for improving GSAT". In: *AAAI/IAAI, Vol. 1.* 1996, pp. 350–355.

[11] Jon Kleinberg and Eva Tardos. *Algorithm Design.* 1st ed. Pearson, 2006.

[12] Stuart Russell and Peter Norvig. *Artificial Intelligence. A Modern Approach.* 3rd ed. Pearson, 2010.

[13] Bart Selman, Henry A Kautz, Bram Cohen, et al. “Noise strategies for improving local search”. In: *AAAI*. Vol. 94. 1994, pp. 337–343.

[14] William M Spears. “Simulated annealing for hard satisfiability problems.” In: *Cliques, Coloring, and Satisfiability* 26 (1993), pp. 533–558.

[15] Phil Sung. *Maximum satisfiability.* 2006.

[16] *The International SAT Competition Web Page.*

[17] Wei Wei and Bart Selman. “A new approach to model counting”. In: *International Conference on Theory and Applications of Satisfiability Testing.* Springer. 2005, pp. 324–339.

[18] David Williamson. *A simple, greedy approximation algorithm for MAX SAT.*

[19] Jiongzhi Zheng, Jianrong Zhou, and Kun He. “Farsighted Probabilistic Sampling based Local Search for (Weighted) Partial MaxSAT”. In: *arXiv preprint arXiv:2108.09988* (2021).