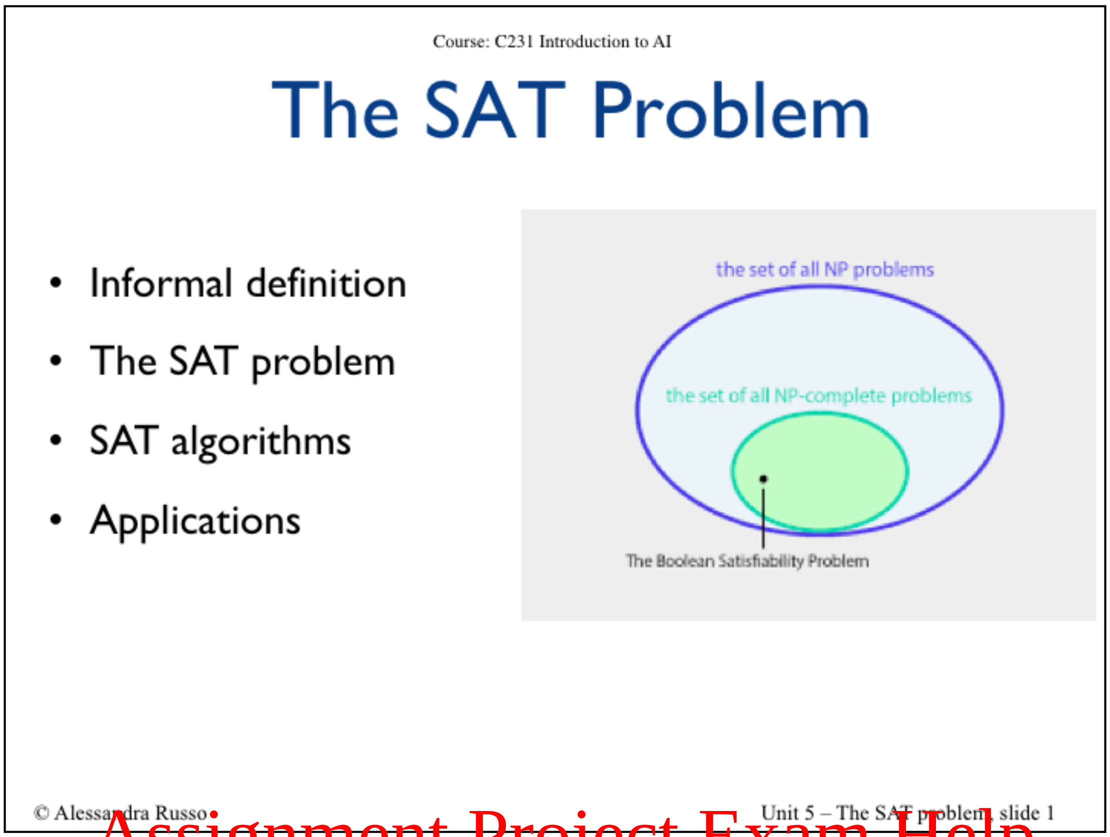
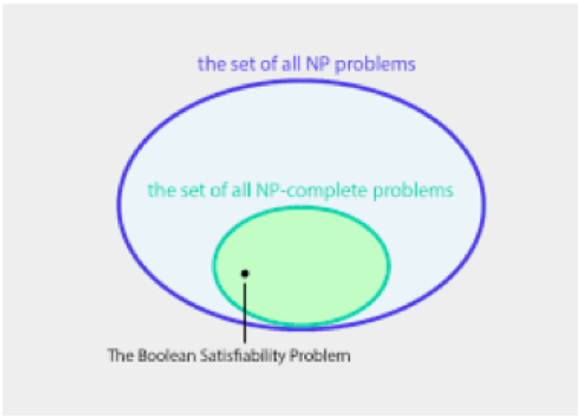


Course: C231 Introduction to AI

The SAT Problem

- Informal definition
- The SAT problem
- SAT algorithms
- Applications


the set of all NP problems
the set of all NP-complete problems
The Boolean Satisfiability Problem

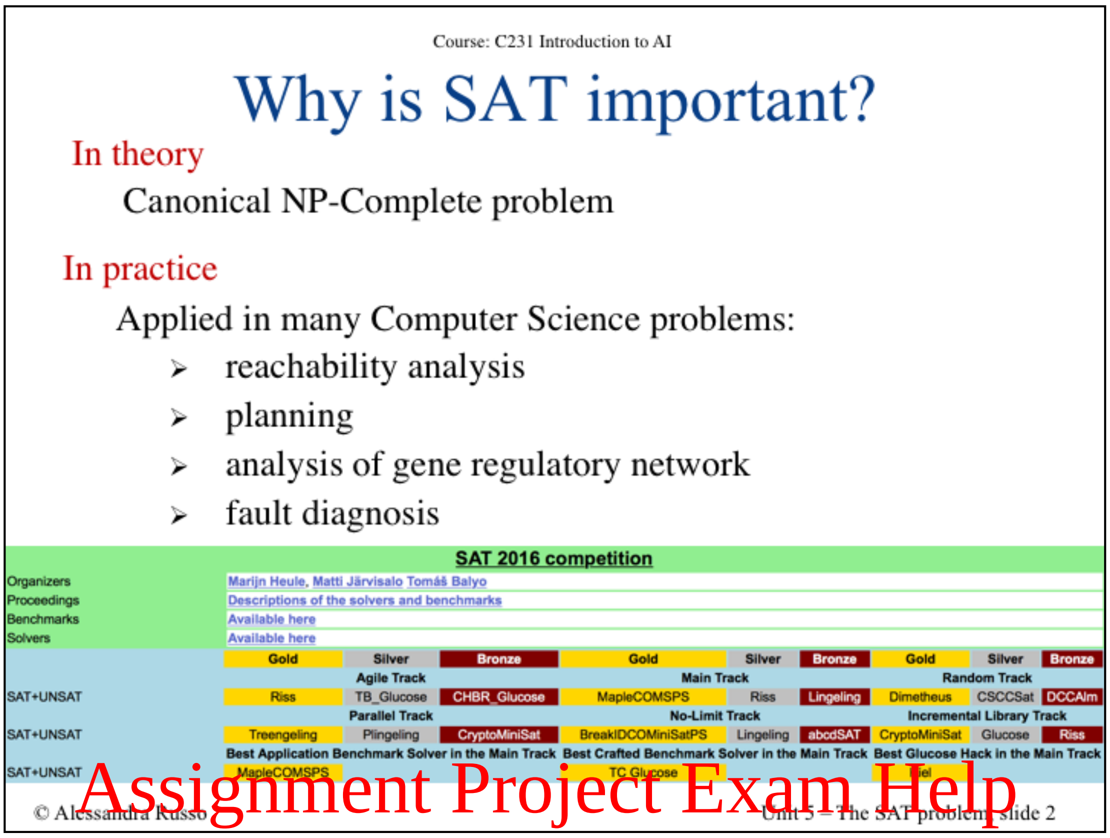
© Alessandra Russo Unit 5 – The SAT problem, slide 1

In the past lecture, we have introduced the notion of planning and we have looked into more details how planning problems can be formalised as Abductive Event Calculus problems and solved using an abductive reasoning algorithm. There are many other ways for solving planning problems and one of these is by formulating a planning problem into a Boolean Satisfiability Problem. In this lecture we will introduce what a Boolean Satisfiability problem, also known as SAT problem, is, and we will give the formal definition and explore basic algorithms for solving the SAT problem.

SAT problem is one of the core computer science problems, which is gaining more and more relevance in AI. This is because nowadays more problems are being solved faster by SAT solvers than other means. In computer science, problems can be classified based on the time and space complexity of the algorithm used to compute an answer for every instance of the problem. For instance, you might consider a computational problem to be "Is n a prime number?" An instance of this problem is given when we pick a particular value of n. So the class of this problem depends on the time and space complexity of the algorithm that we might use in computing an answer for each value of n.

In complexity theory, a class of decision problems is the known class of **NP problems**. These are problems that can be solved in polynomial time by a non-deterministic Turing machine. This class contains a specific class of decision problems called **NP-complete** problems**.** These are the hardest NP decision problems, but they have the property that every NP decision problem can be reduced to an NP-complete problem in polynomial time. So if we had an algorithm that is able to solve an NP problem in polynomial time we would be able to solve all NP problem in polynomial time as these can be reduced to the NP-complete problem in polynomial time and then solved in polynomial time. The overall time complexity will still be polynomial.

The big interest in the Boolean Satisfiability problem started when Cook-Levin proved **Boolean Satisfiability decision problem to be NP-complete (1971).** This meant that every NP problems including optimization problems, are at most as difficult to solve as the SAT problem. Despite the large progress made in developing faster algorithms for SAT, there is no algorithm that can efficiently solve all Boolean SAT problems in polynomial time, and it is generally believed that no such algorithm exists. Trying to come up with such a algorithm would be equivalent to solve the famous open problem in the theory of computing of whether P=NP.



Course: C231 Introduction to AI

Why is SAT important?

In theory

Canonical NP-Complete problem

In practice

Applied in many Computer Science problems:

- ➢ reachability analysis
- ➢ planning
- ➢ analysis of gene regulatory network
- ➢ fault diagnosis

SAT 2016 competition

Organizers	Marijn Heule, Matti Järvisalo Tomáš Balyo								
Proceedings	Descriptions of the solvers and benchmarks								
Benchmarks	Available here								
Solvers	Available here								
	Gold	Silver	Bronze	Gold	Silver	Bronze	Gold	Silver	Bronze
	Agile Track			Main Track			Random Track		
SAT+UNSAT	Riss	TB_Glucose	CHBR_Glucose	MapleCOMSPS	Riss	Lingeling	Dimetheus	CSCCSat	DCCAlm
	Parallel Track			No-Limit Track			Incremental Library Track		
SAT+UNSAT	Treengeling	Plingeling	CryptoMiniSat	BreakIDCOMiniSatPS	Lingeling	abcdSAT	CryptoMiniSat	Glucose	Riss
	Best Application Benchmark Solver in the Main Track			Best Crafted Benchmark Solver in the Main Track			Best Glucose Hack in the Main Track		
SAT+UNSAT	MapleCOMSPS			TC Glucose			[illegible]		

© Alessandra Russo Unit 5 – The SAT problem, slide 2

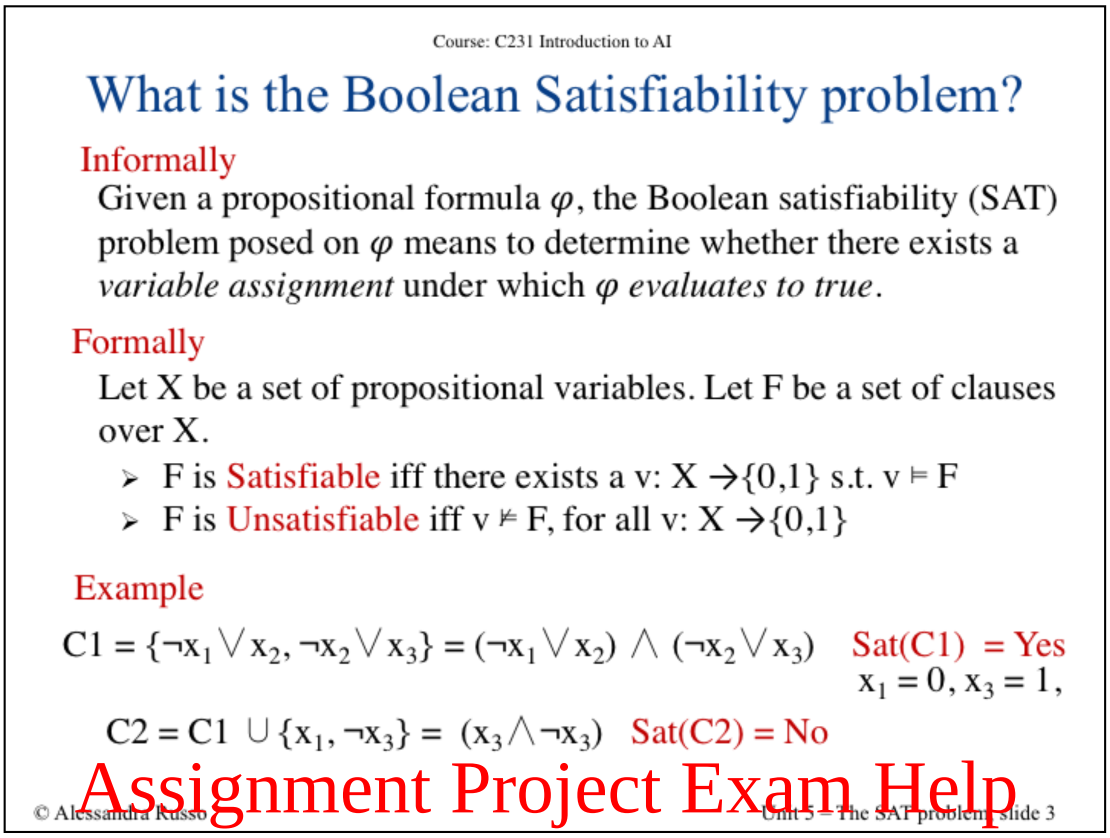
The SAT problem is one of the core topics of Computer Science with strong relevance in Artificial Intelligence. The past decade has seen an enormous progress in the performance of Boolean satisfiability (SAT) solvers . Despite the worst-case exponential run time of all known algorithms, satisfiability solvers are increasingly leaving their mark as a general-purpose tool in many areas of computer science.

From a theoretical point of view, it is important because it represents a canonical NP-complete problem, as explained in the first slide. So trying to solve the SAT problem in an efficient manner can shed light on how to solve any other NP problem.

From a practical point of view, SAT is used in many Computer Science problems. In the context of formal methods, it is important for solving the **reachability problem** in both hardware and software verification.

In Artificial Intelligence, it is used for solving planning and scheduling problems. In bioinformatics for analysis of genetic regulatory networks. In design automation, it supports problems such as fault diagnosis and noise analysis. Another example of a recent application of SAT solvers is in computing stable models used in the answer set programming paradigm, a powerful knowledge representation and reasoning approach that we will be covering in this course the next two weeks. In these applications (planning, verification, and answer set programming) the translation into a propositional representation (the "SAT encoding") is done automatically, and is hidden from the user. The user only deals with the appropriate higher-level representation language of the application domain.

Annual SAT competitions have led to the development of dozens of clever implementations of such solvers and challenging hand-crafted benchmark problems. Modern SAT solvers provide a "black-box" procedure that can often solve hard structured problems with over a million variables and several million constraints.



What is the Boolean Satisfiability problem?

Informally

Given a propositional formula φ, the Boolean satisfiability (SAT) problem posed on φ means to determine whether there exists a *variable assignment* under which φ *evaluates to true*.

Formally

Let X be a set of propositional variables. Let F be a set of clauses over X.

- F is Satisfiable iff there exists a v: X →{0,1} s.t. $v \models F$
- F is Unsatisfiable iff $v \not\models F$, for all v: X →{0,1}

Example

$C1 = \{\neg x_1 \vee x_2, \neg x_2 \vee x_3\} = (\neg x_1 \vee x_2) \wedge (\neg x_2 \vee x_3)$ Sat(C1) = Yes $x_1 = 0, x_3 = 1,$

$C2 = C1 \cup \{x_1, \neg x_3\} = (x_3 \wedge \neg x_3)$ Sat(C2) = No

NP-complete problems were considered practically unsolvable. But since 1990, there have been continuous breakthroughs in SAT solving algorithms. This has been also leading to new results in state space search, with applications in construction of intelligent systems, and more recently with impacts in other areas, including probabilistic reasoning and machine learning. Study algorithms for Boolean SAT solving will provide you with idea on how to tackle other problems in computer science and AI.

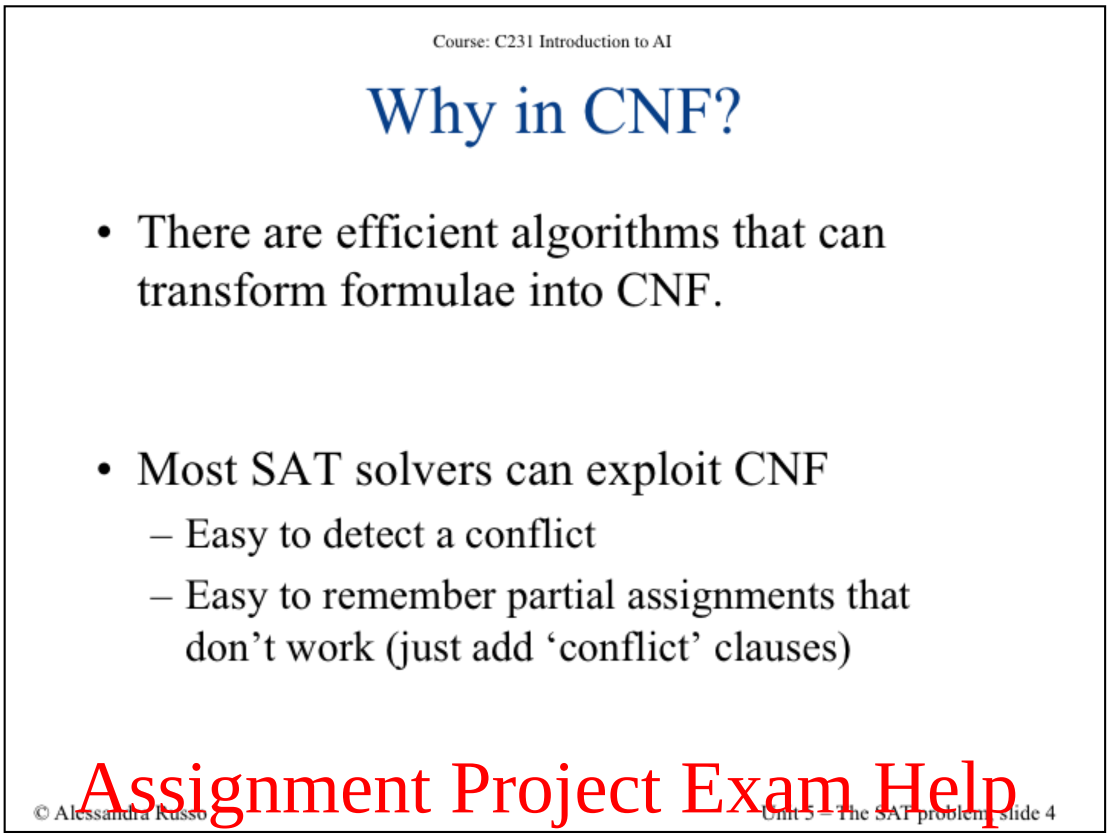
But, let's now define what the Boolean Satisfiability (SAT) problem is. In brief it is the problem of deciding (i.e. which requires a yes/no answer) if **there is an assignment to the variables of a Boolean formula φ such that the formula is satisfied** (or which makes the formula true).

Most algorithms for the Boolean SAT problem (also referred to as SAT solvers) use a *conjunctive normal form* (CNF) representation of the Boolean formula. The given formula is in this case represented as *a conjunction of clauses*, where each clause is a disjunction of literals, and a literal can be a propositional variable or its negation.

The slide gives the formal definition of a satisfiability problem in terms of formulae represented as set of clauses, or equivalently represented in Conjunctive normal form (CNF), conjunction of clauses.

For instance, if we consider the set of clauses $C1 = \{(\neg x_1 \vee x_2), (\neg x_2 \vee x_3)\}$, the satisfiability problem applied to C1 would return answer Yes and an assignment that satisfies the set C1 would be for instance, $x_1 = 0$ and $x_3 = 1$. If now we consider the set of clauses $C2 = C1 \cup \{x_1, \neg x_3\}$, we would have that the SAT problem would return in this case the answer No, as clearly there is no assignment that can satisfies x_3 and $\neg x_3$ at the same time. This is to show you that solving the SAT(C1) is essentially a decision problem on three variables x_1, x_2 and x_3, because obviously determining the truth value of each of these variables would also give the truth value of their respective negations.

In practice, one is not only interested in the decision ("yes/no") problem, but also in finding an actual satisfying assignment if there exists one. All practical satisfiability algorithms, known as SAT solvers, do produce such an assignment if it exists.

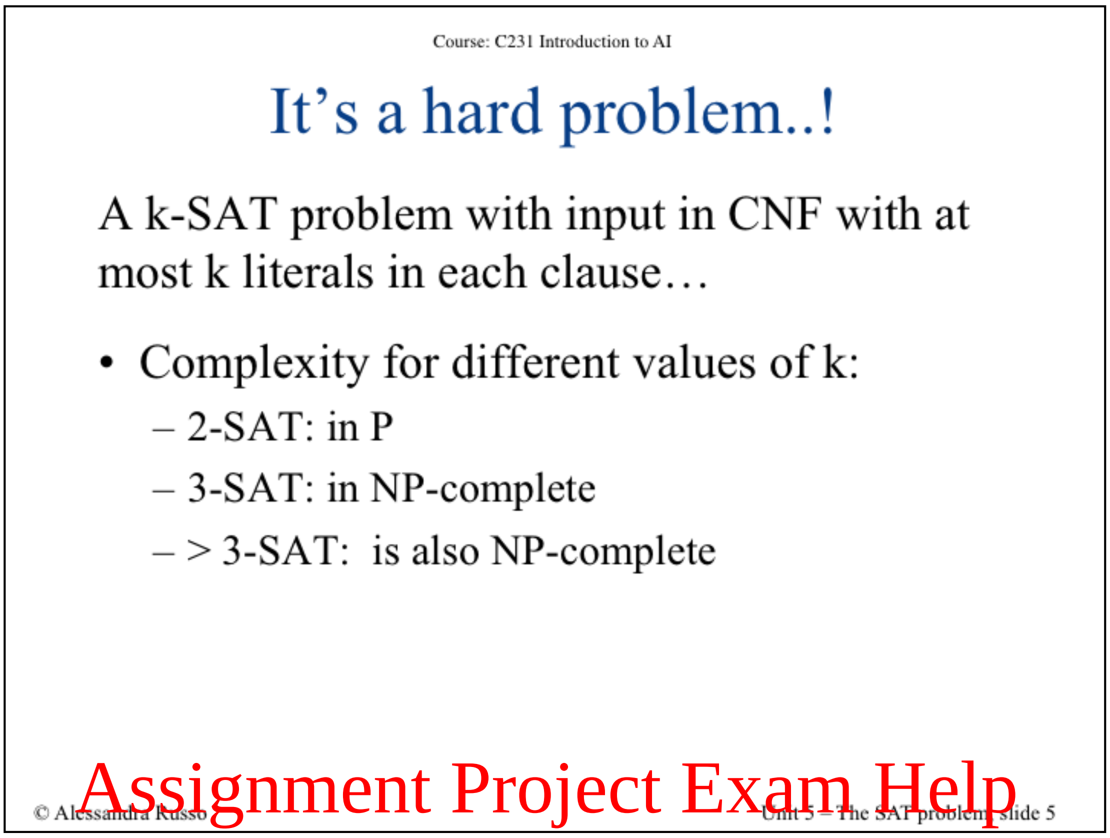


Why in CNF?

- There are efficient algorithms that can transform formulae into CNF.
- Most SAT solvers can exploit CNF
  - Easy to detect a conflict
  - Easy to remember partial assignments that don't work (just add 'conflict' clauses)

The use of CNF representation of a given formula in input allows for different variations of SAT solver algorithms, which in turn take advantage of the fact that it is easier to detect conflicts among clauses and in within clauses, as well as remember during the search for assignments, those assignments that don't work.

On the other end, the translation to SAT problems into propositional CNF form generally leads to a substantial increase in problem representation. However, large SAT encodings are no longer an obstacle for modern SAT solvers. In fact, for many combinatorial search and reasoning tasks, the translation to SAT followed by the use of a modern SAT solver is often more effective than a custom search engine running on the original problem formulation. The explanation for this phenomenon is that SAT solvers have been engineered to such an extent that their performance is difficult to duplicate, even when one tackles the reasoning problem in its original representation.



It's a hard problem..!

A k-SAT problem with input in CNF with at most k literals in each clause…

- Complexity for different values of k:
  - 2-SAT: in P
  - 3-SAT: in NP-complete
  - > 3-SAT: is also NP-complete

But despite the various tricks, the SAT problem is still a difficult computational task.

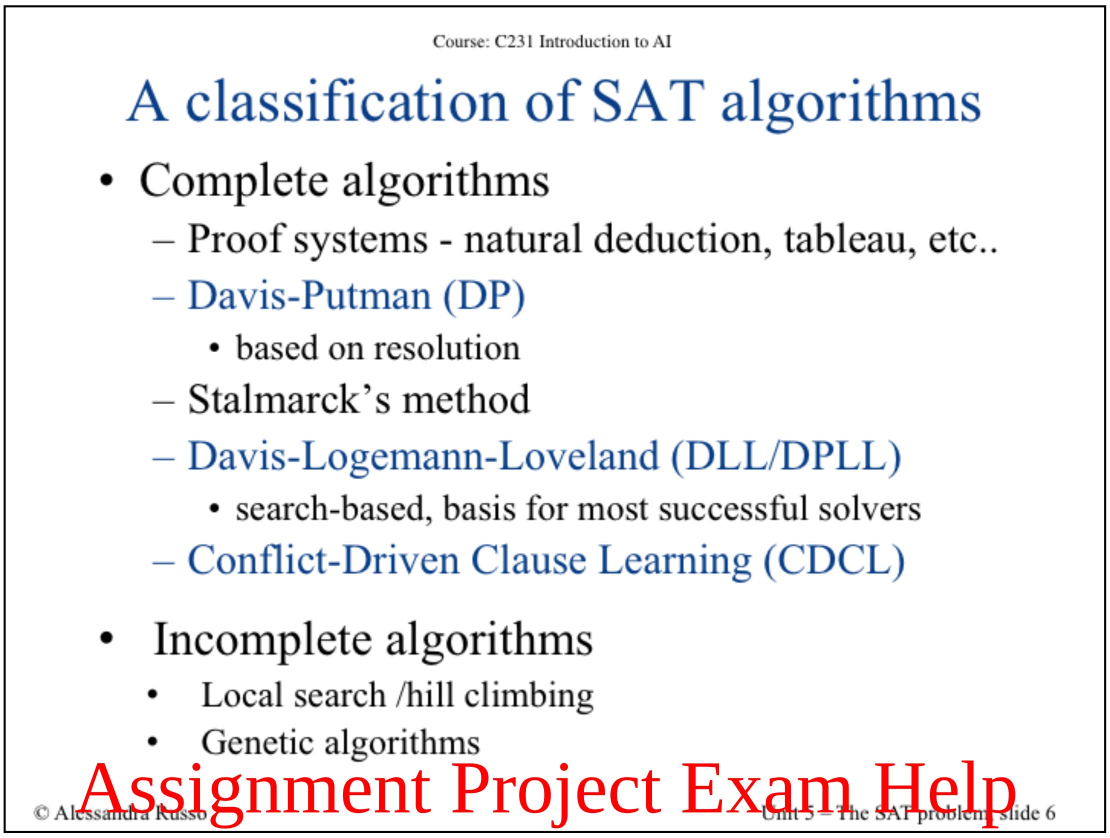
Classifications of this problem have been studied with respect to the number of literals that can appear in each clause of the given set. In particular, we talk about 2-SAT problem as the satisfiability problem where each clause in the given set of clauses is restricted to a disjunction of two literals. Similarly, 3-SAT is the problem where each clause is a disjunction of three literals and so on.

2-SAT has been shown to be in the class of P problems, which means that it is possible to solve such problem in polynomial time with a deterministic Turing machine.

3-SAT problems have been shown to be in the class of NP-complete problems, so that can be solved in polynomial time by the a nondeterministic Turing machine.

Complexity of k-SAT problems where each clause has at most k literals is also NP-complete. In fact, intuitively, a k-SAT problem, for $k \geq 3$, would be not easier that a 3-SAT problem but also not harder than a SAT problem. Both 3-SAT and SAT have been proved to be NP-complete, so k-SAT must be NP-complete too. Formally, it has been shown that a k-SAT problem can be transformed into a 3-SAT problem by rewriting every clause with $k>3$ literals into a set of k-2 clauses with 3 literals by means of new dummy variables. For instance, if we have a clause $\{a,b,c,d\}$ we could rewrite it into the set of clauses $\{\{a,b,x_1\}\{\neg x_1,c,d\}\}$where x_1 is a newly introduced dummy variable that is not included in the language of the original set of clauses. Although the transformed set of clauses is much bigger than the initial one the growth is polynomial in the number of clauses in the given initial set.

During this lecture we will look at two main types of SAT solvers, where our problems will be typically a mix of binary and ternary clauses.


Course: C231 Introduction to AI

A classification of SAT algorithms

- Complete algorithms
  - Proof systems - natural deduction, tableau, etc..
  - Davis-Putman (DP)
    - based on resolution
  - Stalmarck's method
  - Davis-Logemann-Loveland (DLL/DPLL)
    - search-based, basis for most successful solvers
  - Conflict-Driven Clause Learning (CDCL)
- Incomplete algorithms
  - Local search /hill climbing
  - Genetic algorithms

© Alessandra Russo Unit 5 – The SAT problem, slide 6

A complete solution method for the SAT problem is one that, given the input formula F, either produces a satisfying assignment for F or proves that F is unsatisfiable. There are different methods for addressing the SAT solving problem, that is checking whether a given formula is satisfiable. In the first year you have seen the truth table method as a mechanism for constructing assignments an checking which assignment makes the formula true. Although this is clearly a simple method, it is also the most inefficient one. The tables tend to grow very fast, the number of rows grows exponentially in the number of variables, and the number of columns grow linearly in the number of subformulae of the given formula.

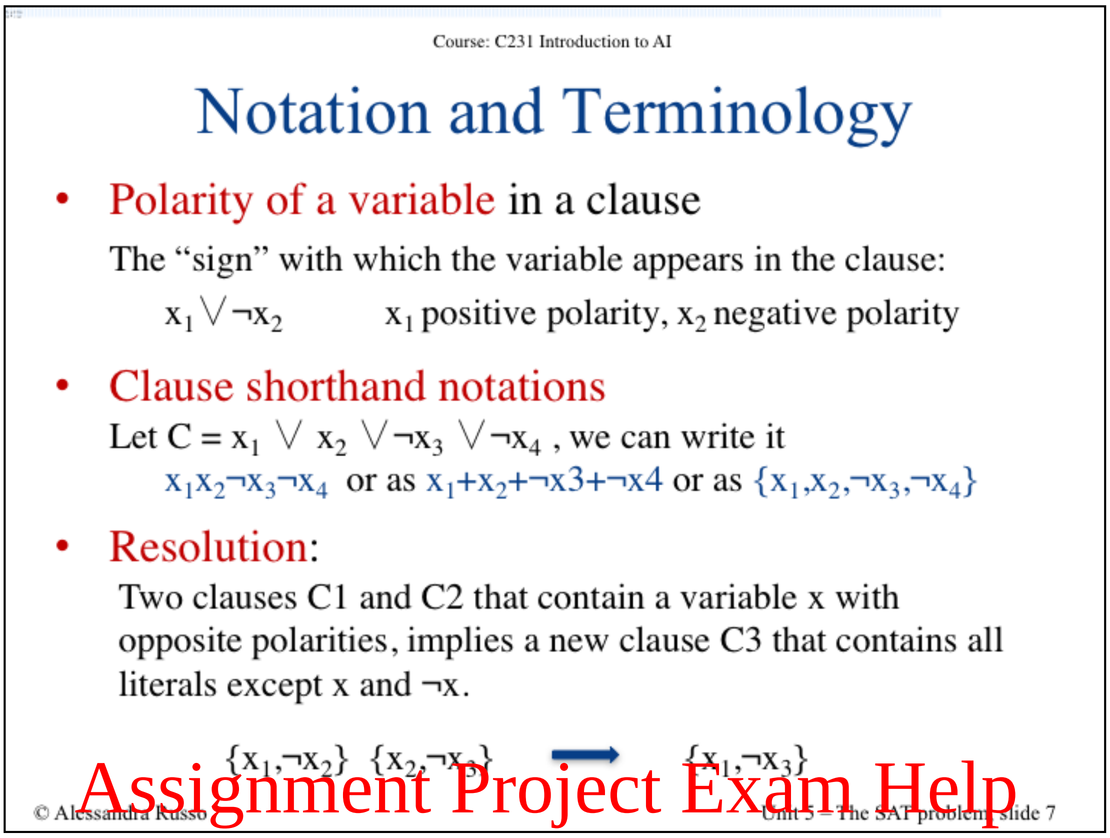
There are clearly better methods!

Sat solvers algorithms can be grouped into complete and incomplete methods. Incomplete algorithms cannot guarantee that eventually the algorithm with return with un satisfiable (No answer) or satisfiable (Yes answer) and an assignment.

Complete algorithms are instead guaranteed to do so. This slide lists some of the complete SAT solving algorithms. The most basic one is the DP algorithm, which is based on resolution. A subsequent version was proposed by the same authors, referred to as the DPLL algorithm. One of the most surprising things of recent practical progress of SAT solvers is that the best complete methods still remain to be variants of a process introduced several decades ago: **the DPLL procedure**, which performs a backtrack search in the space of partial truth assignments. A key feature of DPLL is efficient pruning of the search space based on falsified clauses.

In the rest of this lecture we will present the DP and the DPLL algorithms.

For the rest of this lecture that an input formula F has already be transformed into clausal form.



Course: C231 Introduction to AI

Notation and Terminology

- Polarity of a variable in a clause

  The "sign" with which the variable appears in the clause:

  $x_1 \vee \neg x_2$ x_1 positive polarity, x_2 negative polarity

- Clause shorthand notations

  Let $C = x_1 \vee x_2 \vee \neg x_3 \vee \neg x_4$, we can write it

  $x_1 x_2 \neg x_3 \neg x_4$ or as $x_1 + x_2 + \neg x3 + \neg x4$ or as $\{x_1, x_2, \neg x_3, \neg x_4\}$

- Resolution:

  Two clauses C1 and C2 that contain a variable x with opposite polarities, implies a new clause C3 that contains all literals except x and ¬x.

  $\{x_1, \neg x_2\}$ $\{x_2, \neg x_3\}$ $\longrightarrow$ $\{x_1, \neg x_3\}$

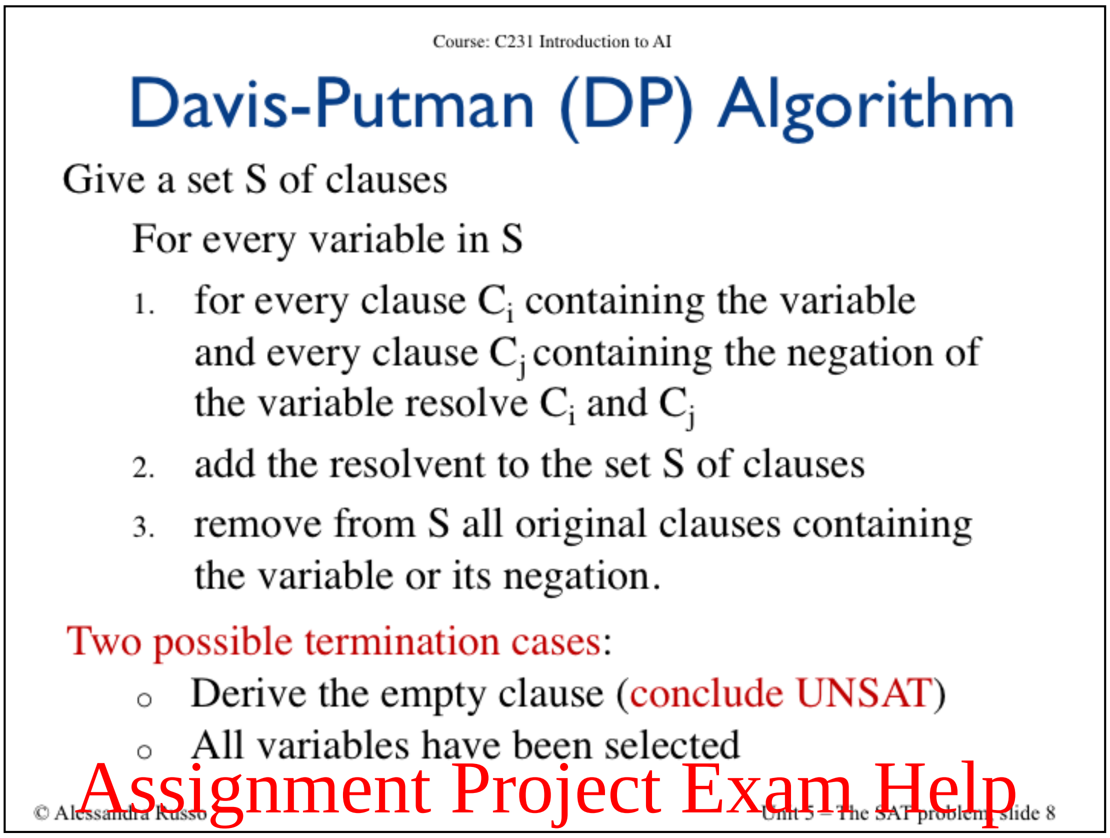
© Alessandra Russo Unit 5 – The SAT problem slide 7

Before introducing the SAT algorithms, let's see some basic notation and terminologies that we will be using when describing the algorithms. To simplify notation, we might represent a clause as a set of literals. Some textbooks represent them as a sum of variables where negated variables are sometime is written as "dashed variables", or a variable with a "line above" it to indicated that it is negated in the clause. In this notation disjunction is written as a "+" and conjunction as a "multiplication". So you might find notations like

(a+b)(b'+c)(a+d') to express a conjunction of clauses of the form:

$(a \vee b) \wedge (\neg b \vee c) \wedge (a \vee \neg d)$.

We will use the set notation introduced in this slide to represent a clause and a set of sets to represent a conjunction of clauses.


Course: C231 Introduction to AI

Davis-Putman (DP) Algorithm

Give a set S of clauses

For every variable in S

1. for every clause C_i containing the variable and every clause C_j containing the negation of the variable resolve C_i and C_j
2. add the resolvent to the set S of clauses
3. remove from S all original clauses containing the variable or its negation.

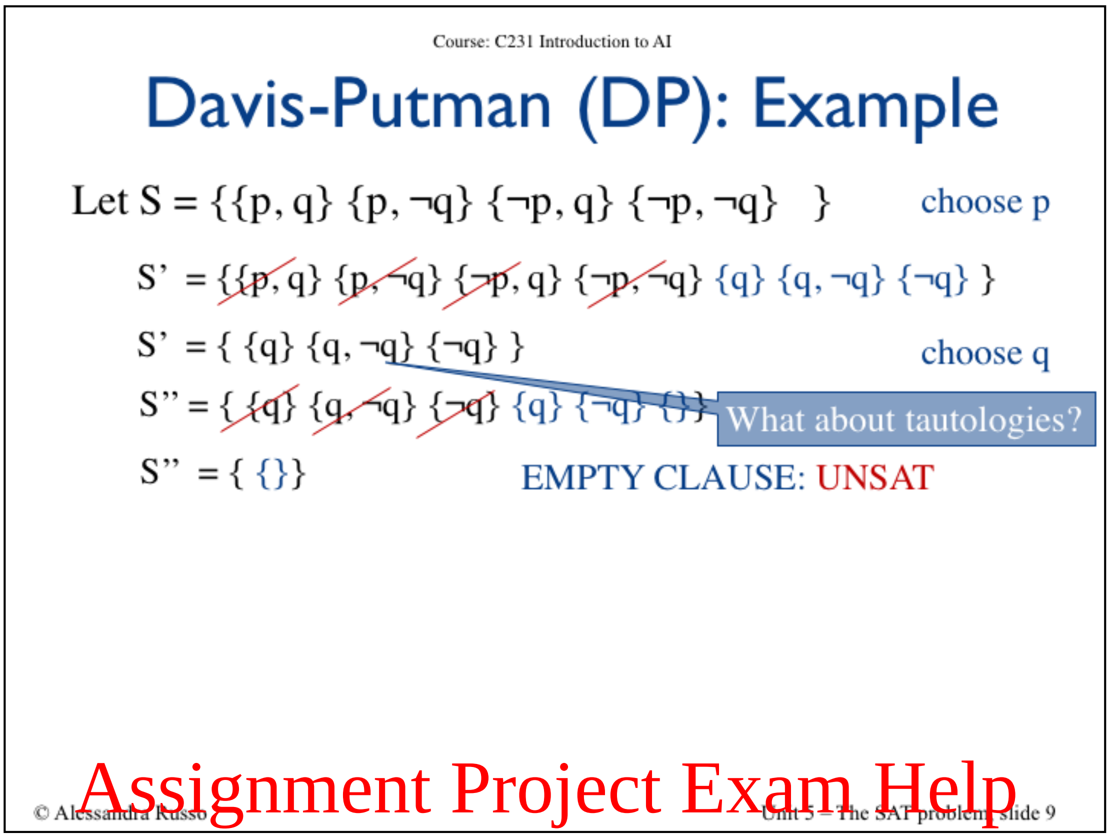
Two possible termination cases:

- Derive the empty clause (conclude UNSAT)
- All variables have been selected

© Alessandra Russo Unit 5 – The SAT problem, slide 8

The simplest notion of the Davis Putman algorithm is based on resolution. The input to this algorithm is a set of clauses. The output is either an empty clause or an empty set of clauses (that is a **no clause** case), or a set of clauses that cannot be further resolve. In the first case, we say that the given set of clauses is **unsatisfiable**. In the latter two cases, we say that the given set of clauses is satisfiable. When the algorithm stops with a set of variables (positive or negative) that cannot be further resolved, these variables will need to be satisfiable in the final assignment that is constructed. We say that these variable inform a **partial assignment.** So a full assignment that satisfies the given set of clauses exists and would have to be constructed by extending this partial assignment to all the variables that appear in the given set of clauses.

The basic step in this algorithm is the resolution, which we have seen before in the course. Essentially assuming that we have already our formula transformed in clausal normal form, the algorithm iterates through the variables that appear in this set of clauses. In particular, it chooses a variable and selects all pairs of clauses that contain this variable positively and negatively (respectively). It adds the resolvents to the given set of clauses and removes from the set any clause includes the selected variable(either positively or negatively). It iterates this step until (i) either the empty clause is generated, or (ii) there are no more clauses (so the given set is now empty), or all variables have been considered. In these latter two cases we say that the initially given set S of clauses is satisfiable, in the first case we can say that the given set S of clauses is unsatisfiable.



Course: C231 Introduction to AI

Davis-Putman (DP): Example

Let S = {{p, q} {p, ¬q} {¬p, q} {¬p, ¬q} } choose p

S' = {~~{p, q}~~ ~~{p, ¬q}~~ ~~{¬p, q}~~ ~~{¬p, ¬q}~~ {q} {q, ¬q} {¬q} }

S' = { {q} {q, ¬q} {¬q} } choose q

S'' = { ~~{q}~~ ~~{q, ¬q}~~ ~~{¬q}~~ {q} {¬q} {}}

What about tautologies?

S'' = { {}} EMPTY CLAUSE: UNSAT

© Alessandra Russo Unit 5 – The SAT problem, slide 9

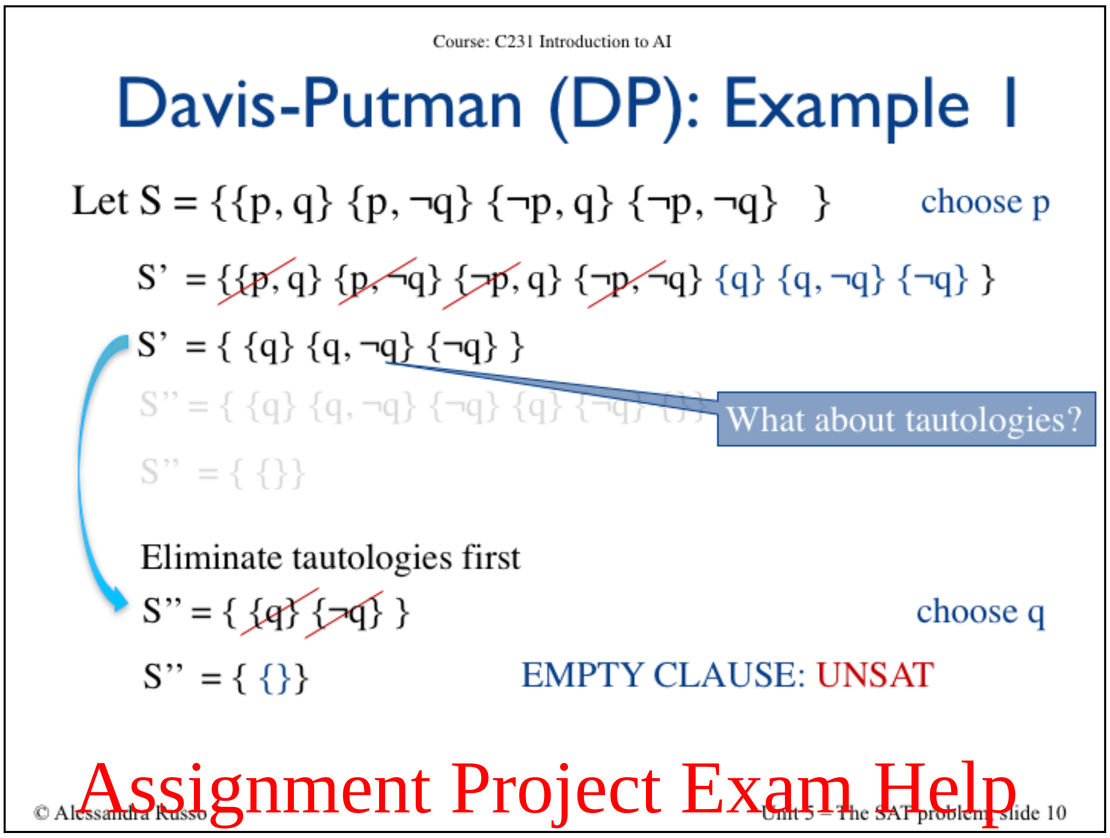
Consider the example in this slide. The algorithm generates the empty clauses, i.e. the initial set of clauses has been reduced to a set that include the empty clause (i.e. empty set). So the given set S of clauses is unsatisfiable.

You can easily see that this algorithm is really not efficient. Given a set of clauses, at any iteration we might end up in the worse case scenario with exponential blow-up in the size of the given set.

This algorithm has been extended into a more efficient version called DPLL, or DLL. Before introducing the various parts of this more efficient SAT solver, let's make few observations on this example. During the resolution step, it is possible for instance that the generated clause is (or includes) a tautology. This is normally a clause of the form $(p \vee \neg p)$, which is obviously always true independent on the truth value of p. A possible obvious simplification could be for instance to remove tautology clauses before choosing the next variable during the computation.

For instance, in the example above, the generated set S' of clauses includes the tautology {q,¬q}. This can be removed before choosing the next variable q, resulting then in less resolution steps and at the same time in a smaller set of clauses.

This is shown in the next slide.



Course: C231 Introduction to AI

Davis-Putman (DP): Example 1

Let S = {{p, q} {p, ¬q} {¬p, q} {¬p, ¬q} } choose p

S' = {{~~p~~, q} {~~p~~, ¬q} {~~¬p~~, q} {~~¬p~~, ¬q} {q} {q, ¬q} {¬q} }

S' = { {q} {q, ¬q} {¬q} }

S'' = { {q} {q, ¬q} {¬q} {q} {¬q} {} }

S'' = { {}}

What about tautologies?

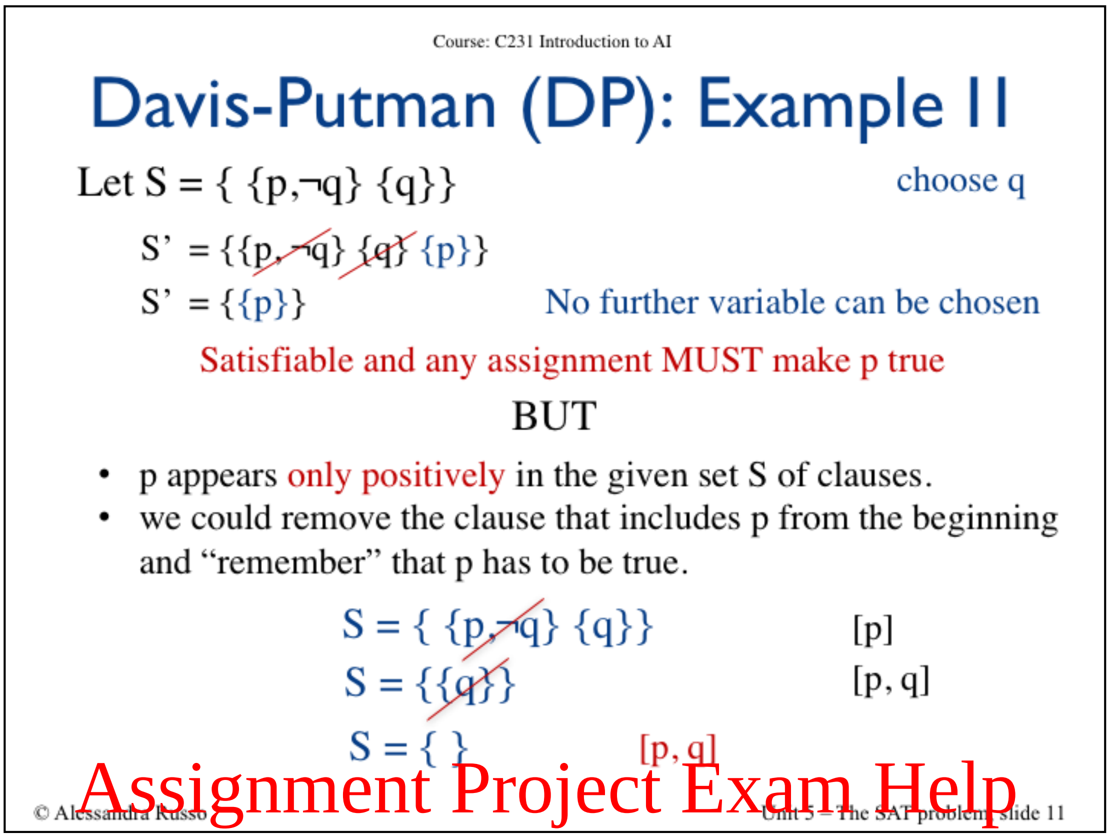
Eliminate tautologies first

S'' = { ~~{q}~~ ~~{¬q}~~ } choose q

S'' = { {}} EMPTY CLAUSE: UNSAT

© Alessandra Russo Unit 5 – The SAT problem, slide 10

Useful rule to apply in order to make the algorithm simpler is to remove tautologies from the set current set of clauses.


Course: C231 Introduction to AI

Davis-Putman (DP): Example II

Let S = { {p,¬q} {q}} choose q

$S' = \{\{p, \cancel{\neg q}\}\ \cancel{\{q\}}\ \{p\}\}$

$S' = \{\{p\}\}$ No further variable can be chosen

Satisfiable and any assignment MUST make p true

BUT

- p appears only positively in the given set S of clauses.
- we could remove the clause that includes p from the beginning and "remember" that p has to be true.

$S = \{\ \cancel{\{p, \neg q\}}\ \{q\}\}$ [p]

$S = \{\cancel{\{q\}}\}$ [p, q]

$S = \{\ \}$ [p, q]

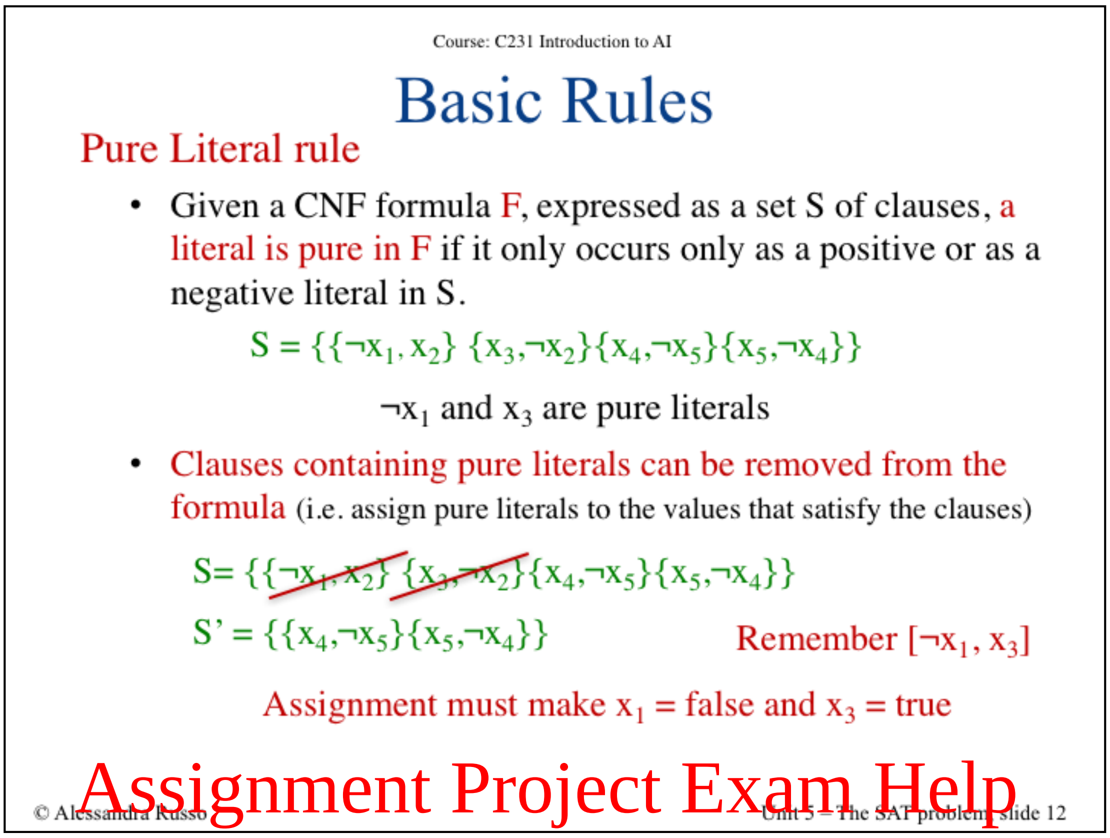
© Alessandra Russo Unit 5 – The SAT problem, slide 11

You can consider now this second example. In this case the DP algorithm would return the list [p] of variables (left over) and the answer Satisfiable, stating that the given set S is satisfiable provided that p is true.

But, the language of the given set of clauses in this case is {p, q}. So to construct a full assignment we would have to determine the value of q as well. Looking at the given set of clauses, q must be true. So the only assignment that makes the given set satisfiable is {p=true, q=true). Reconstructing this full assignment may be complex: the output that the algorithm would give when the answer is Satisfiable, might only include some of the variables in the language. So is there a more efficient way of checking the satisfiability?

We notice here that p appears in the given set S only positively. So we could in this case remember that p has to be true and since we remember it, we could also eliminate all the clauses that include p, since they would automatically be true, once we know that p is true. In this case, the set S of given clauses will only retain the clause S = {{q}}. Now q is a pure literal in S and we can remove the clause {q} from S and remember also q. We are left with an empty set of clauses. So we can conclude that S is satisfiable provided that p is true and q is true. We don't need to apply any resolution.

Further simplification rules can be apply to make the DP algorithm more efficient. Let's see other two important rules. We will then present the new version of the Davis Putman algorithm, called the DLL, or DPLL , which was proposed few years later by M. Davis, G. Logemann and D. Loveland and which makes use of these simplification rules.


Course: C231 Introduction to AI

Basic Rules

Pure Literal rule

- Given a CNF formula F, expressed as a set S of clauses, a literal is pure in F if it only occurs only as a positive or as a negative literal in S.

  $S = \{\{\neg x_1, x_2\}\ \{x_3, \neg x_2\}\{x_4, \neg x_5\}\{x_5, \neg x_4\}\}$

  $\neg x_1$ and x_3 are pure literals

- Clauses containing pure literals can be removed from the formula (i.e. assign pure literals to the values that satisfy the clauses)

  $S= \{\cancel{\{\neg x_1, x_2\}}\ \cancel{\{x_3, \neg x_2\}}\{x_4, \neg x_5\}\{x_5, \neg x_4\}\}$

  $S' = \{\{x_4, \neg x_5\}\{x_5, \neg x_4\}\}$ Remember $[\neg x_1, x_3]$

  Assignment must make x_1 = false and x_3 = true

© Alessandra Russo Unit 5 – The SAT problem, slide 12

To make the SAT solver more efficient, we can use the notion of "pure literals". Given a set S of clauses, a pure literal is a literal that occurs only with one polarity (it is either only positive, in every clause it appears, or it's only negative in every clauses it appears) in the set S.

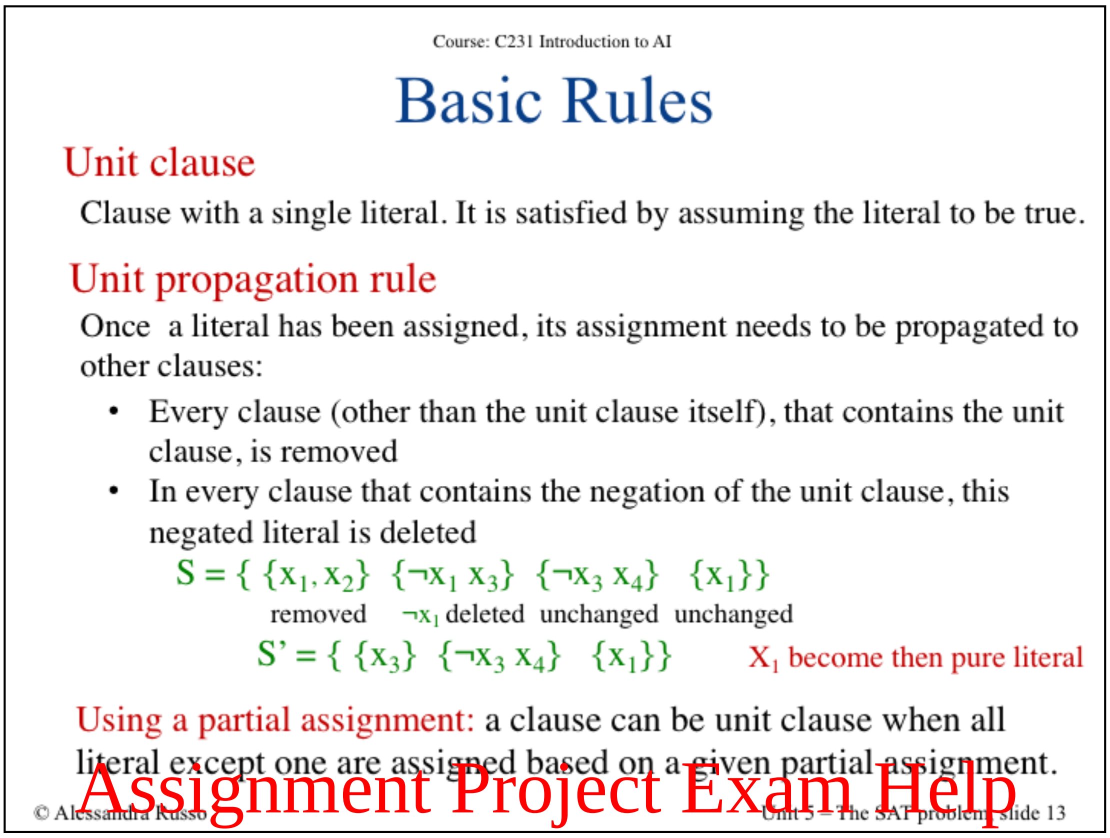
In the above given example, the literal $\neg x_1$ occurs only negatively in S, and the literal x_3 occurs only positively in S. So both $\neg x_1$ and x_3 are pure literals in S.

When checking the satisfiability of S, we can "safely" remove all those clauses that include pure literals. For instance we can simplify the set S by removing the clauses $\{\neg x_1, x_2\}$ and $\{x_3, \neg x_1\}$. We would get in this case the new set $S' = \{\{x_4, \neg x_5\}\{x_5, \neg x_4\}\}$.

To prove satisfiability of S, it would be sufficient now to prove the satisfiability of S'. Why?

We want to show that **S is satisfiable <=> S' is satisfiable** where S' is obtained from S by removing clauses containing pure literals. Clearly **S is satisfiable => S' is satisfiable** is true. Let's reason by contradiction and assume S' is not satisfiable but S is satisfiable. There is no assignment to all the variables that appear in S' that would make the clauses in S' true. Since $S' \subseteq S$, we can safely say that there is no assignment on the variables of S that would make all the clauses of S true. So S is unsatisfiable leading to a contradiction. Let's consider the other case.
S' is satisfiable => S is satisfiable. S' is satisfiable. So there is an assignment on all the variables in S' that satisfies all the clauses in S'. Let C be a clause in S\S'. C includes at least one literal l, whose variable does not appear in S' and that is pure in S. So there is no other clause C1 in S\S' that includes $\neg$l. Since the variable in l is not included in S' we can assign to it whatever truth value needed to make the literal l true, without falsifying any of the clauses in S'. Such an assignment also makes the clause C, that includes l, satisfied. Let's extend the assignment constructed for S' with these true values assignments to each variable of a pure literal, so to make the pure literals true. This extended assignment would now make all the clauses in S\S' that include pure literals satisfied, and by construction it also satisfies all the clauses in S' true. Hence, it satisfies S. So S is satisfiable. S\S'.

In the example above we can simplify S to S', show that S' is unsatisfied as DP would generate the empty clause from S’ and therefore conclude that S is unsatisfied.



Course: C231 Introduction to AI

Basic Rules

Unit clause

Clause with a single literal. It is satisfied by assuming the literal to be true.

Unit propagation rule

Once a literal has been assigned, its assignment needs to be propagated to other clauses:

- Every clause (other than the unit clause itself), that contains the unit clause, is removed
- In every clause that contains the negation of the unit clause, this negated literal is deleted

$S = \{ \{x_1, x_2\} \; \{\neg x_1 \; x_3\} \; \{\neg x_3 \; x_4\} \; \{x_1\}\}$

removed $\neg x_1$ deleted unchanged unchanged

$S' = \{ \{x_3\} \; \{\neg x_3 \; x_4\} \; \{x_1\}\}$ X_1 become then pure literal

Using a partial assignment: a clause can be unit clause when all literal except one are assigned based on a given partial assignment.

© Alessandra Russo Unit 5 – The SAT problem, slide 13

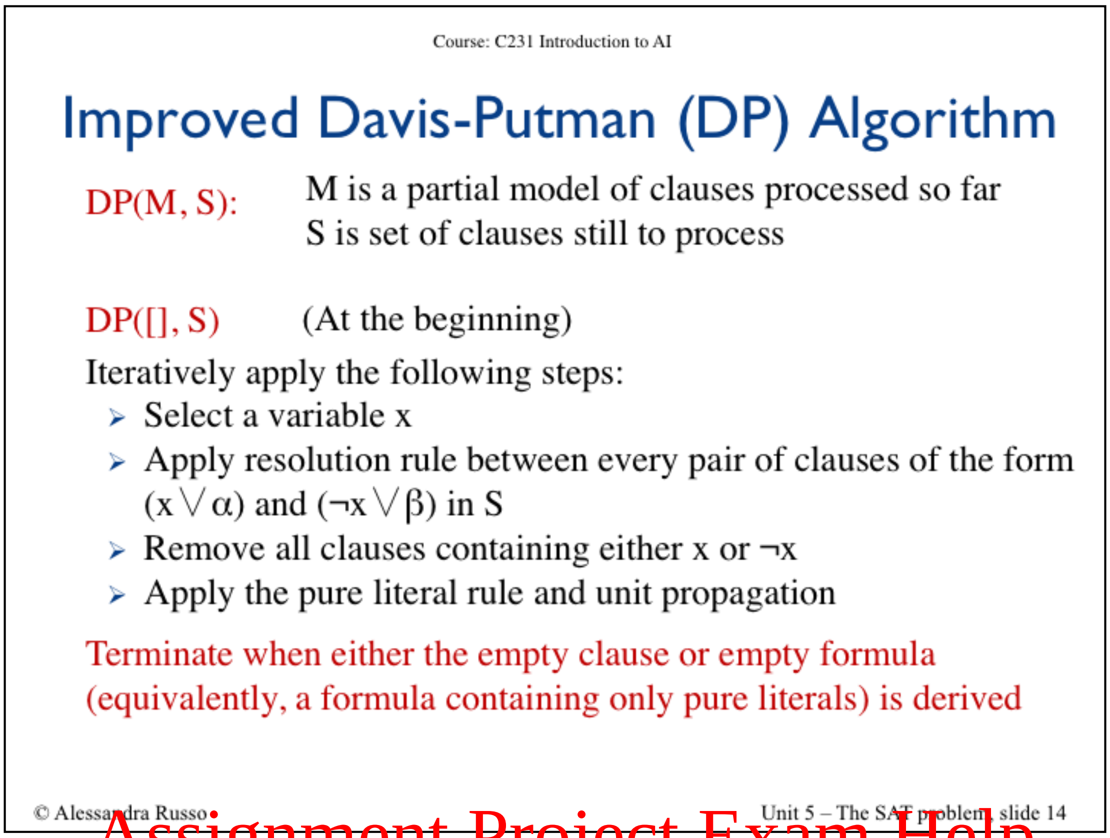
Another important rule used to make the DP algorithm more efficient is the notion of **unit clause** and its related **unit propagation rule**.

The unit clause is a clause that has one single literal l. As the SAT algorithm is trying to satisfy all the given clauses, a unique clause can be satisfied by just assigning to the variable in this literal l the truth value needed to make the literal true. For instance a unique clause $\{\neg x_1\}$ is simply satisfied by assuming x_1 to be false. This implicit assignment can be used to evaluate other clauses in the given set that include this literal. This is done using the following **unit propagation rules**:

1. every clause (other than the unit clause itself), in the given set S that contains the literal l is removed, since now the clause is satisfied;
2. in every clause, in the given set S, that contains ¬l, this negated literal is deleted, since we know ¬l is false and it cannot contribute to the satisfiability of the clause.

Two important observations. The notion of unit clause applies also to clauses whose literals are all assigned except for one. During the SAT algorithm a partial assignment is constructed (see examples below) and therefore literals in the clauses may already have truth value based on this partial assignment. When all literals in a clause are false under the partial assignment expect for one, this clause can be treated as a unit clause. So the unit propagation rules can be applied to the other clauses with respect to this unassigned literal. After the unit propagation rules are applied, the literal in question will clearly result to be a pure literal (since all its negated occurrences in all the other clauses have been removed). So the two unit propagation rules can be followed by a pure literal rule: that is the unit clause in question can be removed, but the literal needs to be added to the partial model that has been constructed.

A second important observation is that the unit propagation rule 2 given above seems to be a special type of resolution rule: one in which one clause has only one literal. However, please note that it is not a full resolution rule are the unit clause cannot be removed, unless the literal is stored in a partial model. **So don't get confused!**



Improved Davis-Putman (DP) Algorithm

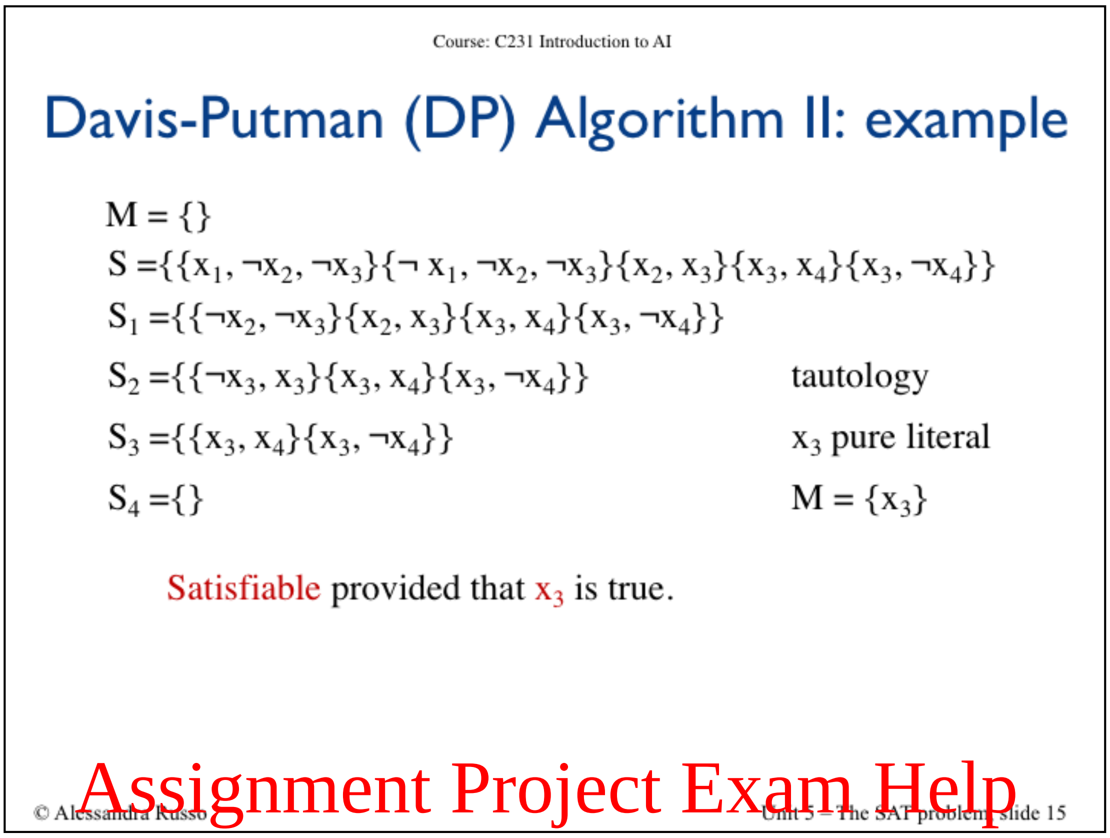
DP(M, S): M is a partial model of clauses processed so far
S is set of clauses still to process

DP([], S) (At the beginning)

Iteratively apply the following steps:

- Select a variable x
- Apply resolution rule between every pair of clauses of the form $(x \vee \alpha)$ and $(\neg x \vee \beta)$ in S
- Remove all clauses containing either x or $\neg x$
- Apply the pure literal rule and unit propagation

Terminate when either the empty clause or empty formula (equivalently, a formula containing only pure literals) is derived


Course: C231 Introduction to AI

Davis-Putman (DP) Algorithm II: example

$M = \{\}$

$S = \{\{x_1, \neg x_2, \neg x_3\}\{\neg x_1, \neg x_2, \neg x_3\}\{x_2, x_3\}\{x_3, x_4\}\{x_3, \neg x_4\}\}$

$S_1 = \{\{\neg x_2, \neg x_3\}\{x_2, x_3\}\{x_3, x_4\}\{x_3, \neg x_4\}\}$

$S_2 = \{\{\neg x_3, x_3\}\{x_3, x_4\}\{x_3, \neg x_4\}\}$ tautology

$S_3 = \{\{x_3, x_4\}\{x_3, \neg x_4\}\}$ x_3 pure literal

$S_4 = \{\}$ $M = \{x_3\}$

Satisfiable provided that x_3 is true.

© Alessandra Russo Unit 5 – The SAT problem, slide 15

Let's consider the example of set S of clauses given in this slide and apply the DP algorithm described in the previous slide.

The given set does not include any pure literal or unit clause and at the start the partial model is empty since no clause has been considered yet.

The only possible step to do in this case is to pick a variable and apply a resolution step.

We have chosen x_1 and the only resolution steps involving this variable is between the first two clauses on the left.

This gives us the new set S_1 of clauses.

Again no pure literals and no unit clauses, so we can pick a second variable and try to apply again resolution.

We choose now x_2 and the only resolution steps involving this variable is between the firs two clauses on the left of S_1.

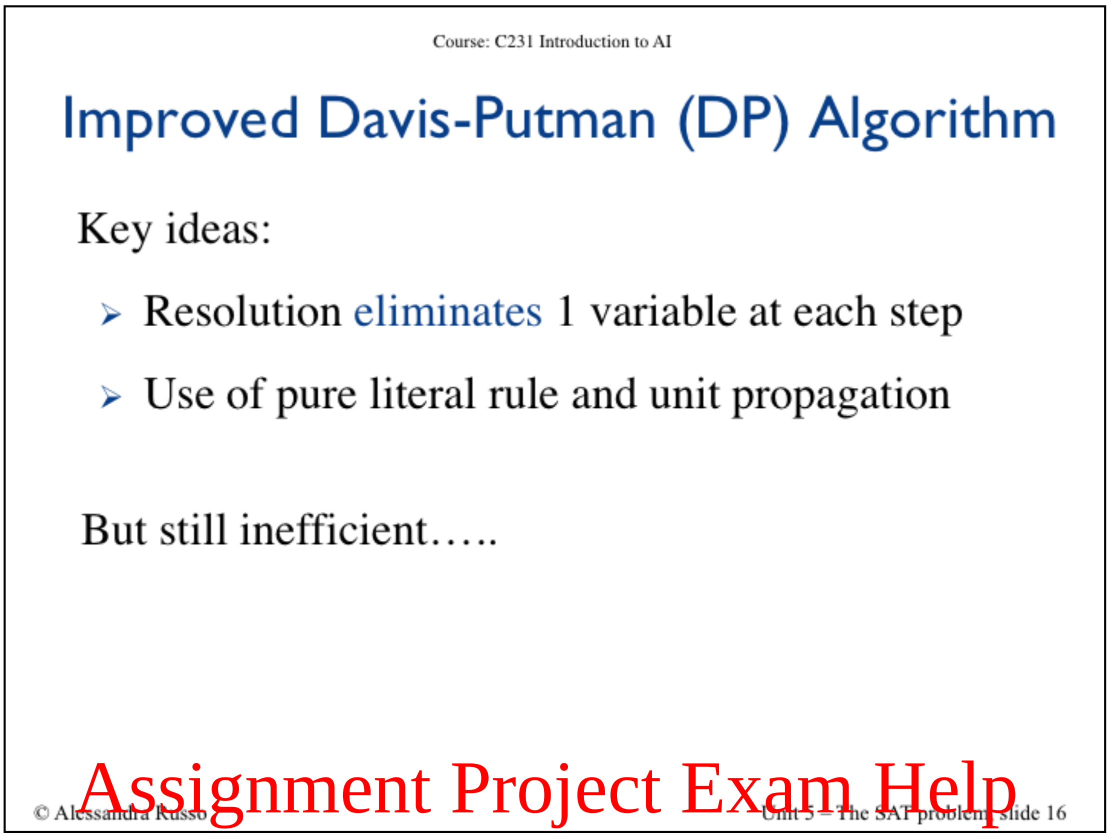
This gives us the new set S_2 of clauses.

Now S_2 includes a tautology clause, $\{\neg x_3, x_3\}$, which can be removed, giving the new set S_3.

S_3 includes now x_3 as pure literal. So the pure literal rule can be applied and both clauses get removed from S_3 and x_3 cam be added to the partial model M.

This gives us now a empty set S_4 of clauses and a partial model $M = \{x_3\}$.

The algorithm stops returning "Yes satisfiable" with partial model $M = \{x_3\}$, which means that the given set of clauses is satisfiable provided that x_3 is true.

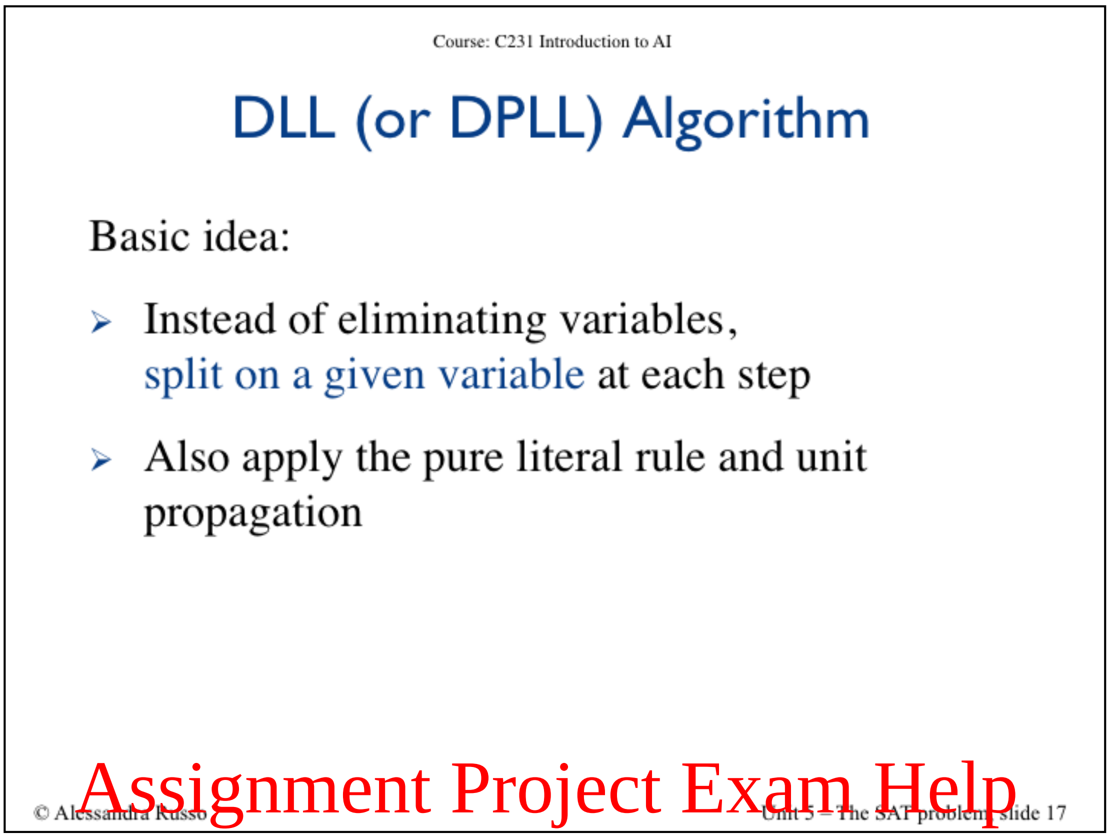


Improved Davis-Putman (DP) Algorithm

Key ideas:

- Resolution eliminates 1 variable at each step
- Use of pure literal rule and unit propagation

But still inefficient…..



Course: C231 Introduction to AI

DLL (or DPLL) Algorithm

Basic idea:

- Instead of eliminating variables, split on a given variable at each step
- Also apply the pure literal rule and unit propagation

© Alessandra Russo

Unit 5 – The SAT problem, slide 17

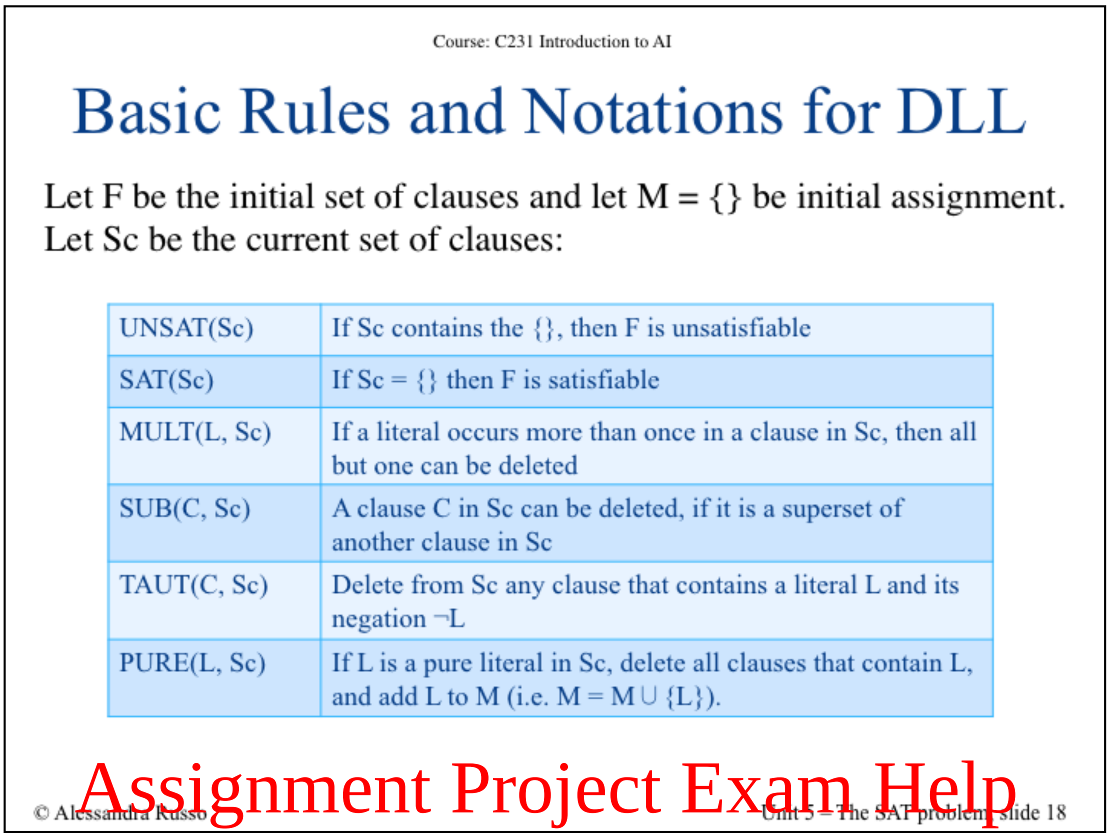
Few years after the DP algorithm, Davis Putnam, G. Logemann and D. Loveland proposed a new SAT solving algorithm that was since then known as the DLL (or DPLL) algorithm.

This algorithm still constitutes today the backbone of many of the very efficient SAT solvers.

The key difference between DP and DLL is that instead of eliminating one variable at each iteration (by applying resolution), we keep the variable but we split on the two possible vales that this variable might have in the final satisfiability assignment.

The same mechanisms of pure literal rule and unit propagation are applied in DLL as in DP.

To give a full account of the algorithm let's summarise all the "simplification rules" that can be applied at each step before applying the splitting rule. All of these rules could also equally applied to the DP algorithm.



Course: C231 Introduction to AI

Basic Rules and Notations for DLL

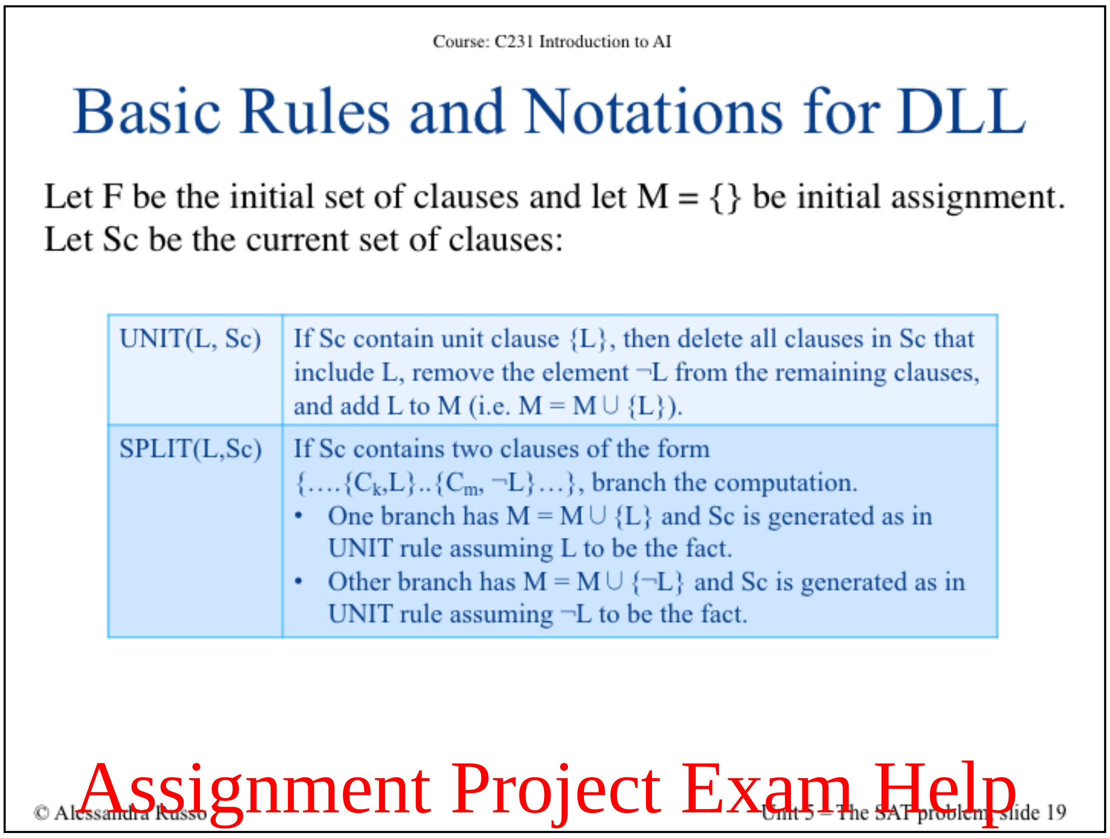
Let F be the initial set of clauses and let M = {} be initial assignment. Let Sc be the current set of clauses:

UNSAT(Sc)	If Sc contains the {}, then F is unsatisfiable
SAT(Sc)	If Sc = {} then F is satisfiable
MULT(L, Sc)	If a literal occurs more than once in a clause in Sc, then all but one can be deleted
SUB(C, Sc)	A clause C in Sc can be deleted, if it is a superset of another clause in Sc
TAUT(C, Sc)	Delete from Sc any clause that contains a literal L and its negation ¬L
PURE(L, Sc)	If L is a pure literal in Sc, delete all clauses that contain L, and add L to M (i.e. M = M ∪ {L}).

© Alessandra Russo   Unit 5 – The SAT problem, slide 18

The first two rules above are the terminating cases of the algorithm, which also applies to the improved DP algorithm that we saw before.

The remaining rules are simple simplifications that can be applied to the current set of clauses, being this the initial set of clauses the given formula has been transformed into, or any of the set of clauses generated during the procedure.


Course: C231 Introduction to AI

Basic Rules and Notations for DLL

Let F be the initial set of clauses and let M = {} be initial assignment. Let Sc be the current set of clauses:

UNIT(L, Sc)	If Sc contain unit clause {L}, then delete all clauses in Sc that include L, remove the element ¬L from the remaining clauses, and add L to M (i.e. M = M ∪ {L}).
SPLIT(L,Sc)	If Sc contains two clauses of the form {….{C_k,L}..{C_m, ¬L}…}, branch the computation. • One branch has M = M ∪ {L} and Sc is generated as in UNIT rule assuming L to be the fact. • Other branch has M = M ∪ {¬L} and Sc is generated as in UNIT rule assuming ¬L to be the fact.

© Alessandra Russo Unit 5 – The SAT problem, slide 19

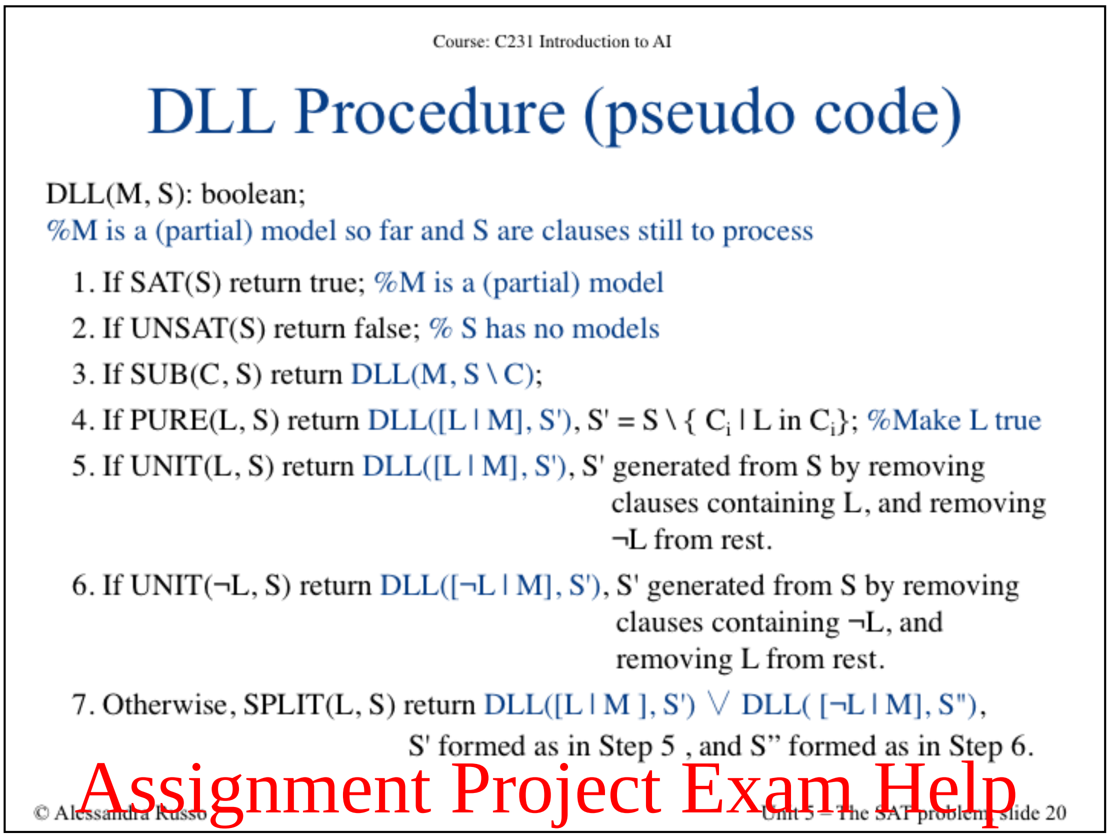
These are the main rules for the DLL algorithm.

Note that the case of UNIT(L, Sc) does not necessarily need to add the literal to M. This is because in practice once the new set of clauses is generated from Sc by the application of unit propagation rule the literal in the unit clause will then become a pure literal and the PURE rule will be applied to it so removing this unit cause too and adding the literal to the partial model M. For convenience we have added the addition of the literal to M in the UNIT rule as well to shorten the derivation.

The splitting step creates a branching in the search tree for the construction of the satisfying (partial) assignment. This is where DLL differs from DP. The literal that is chosen to perform the split is not deleted (via resolution), as in the DP algorithm, but assumed on one branch to be positive and on the other branch to be negative. This assumption is added to the current partial model M of that branch. This is essentially equivalent to have in the current set of clauses Sc a fact literal that corresponds to the assumption made.

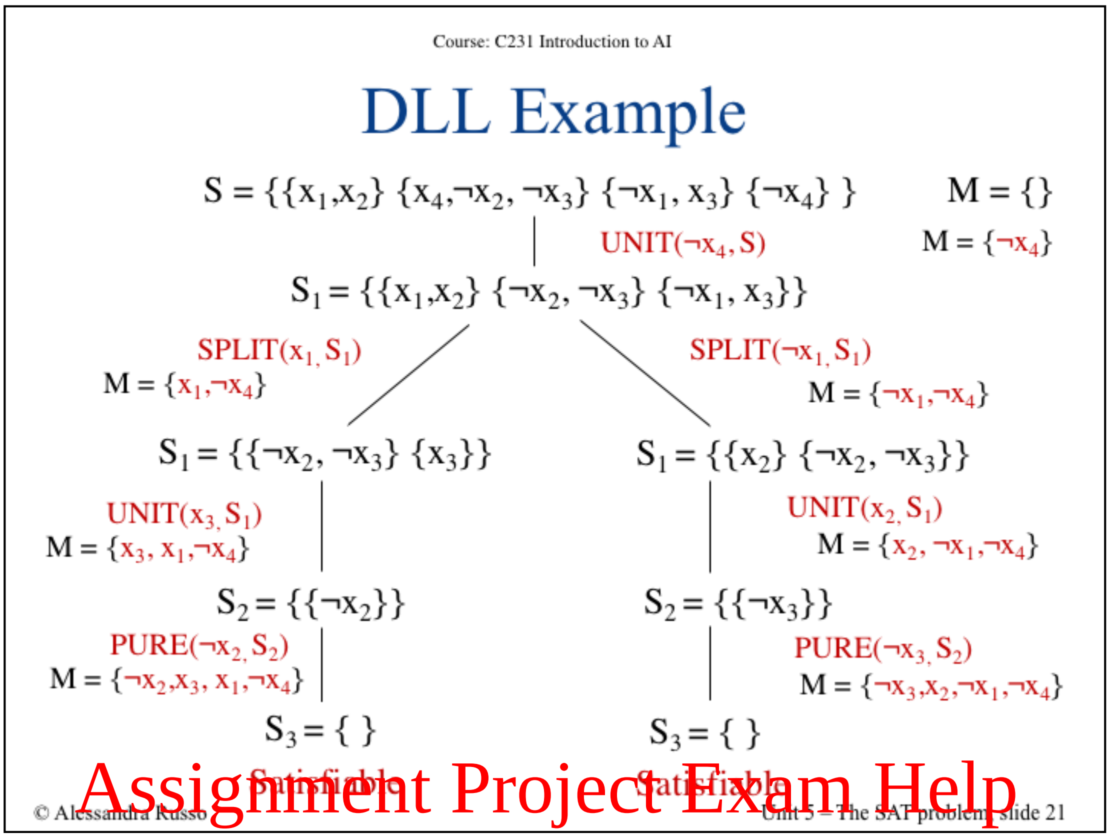
So, in the left branch the chosen literal L is added to M, M = M ∪ {L}, and the unit rule is applied to Sc as if L was a fact (or unit clause) in Sc. On the right branch the negation of this literal is added to M, M = M ∪ {¬L}, and the unit rule is applied to Sc as if ¬L was a fact (or unit clause) in Sc.

The algorithm proceeds by developing both branches.



DLL Procedure (pseudo code)

```
DLL(M, S): boolean;
%M is a (partial) model so far and S are clauses still to process
  1. If SAT(S) return true; %M is a (partial) model
  2. If UNSAT(S) return false; % S has no models
  3. If SUB(C, S) return DLL(M, S \ C);
  4. If PURE(L, S) return DLL([L | M], S'), S' = S \ { C_i | L in C_i}; %Make L true
  5. If UNIT(L, S) return DLL([L | M], S'), S' generated from S by removing
                                           clauses containing L, and removing
                                           ¬L from rest.
  6. If UNIT(¬L, S) return DLL([¬L | M], S'), S' generated from S by removing
                                           clauses containing ¬L, and
                                           removing L from rest.
  7. Otherwise, SPLIT(L, S) return DLL([L | M ], S') ∨ DLL( [¬L | M], S''),
                      S' formed as in Step 5 , and S'' formed as in Step 6.
```



Course: C231 Introduction to AI

DLL Example

$S = \{\{x_1,x_2\}\ \{x_4,\neg x_2, \neg x_3\}\ \{\neg x_1, x_3\}\ \{\neg x_4\}\ \}$ $M = \{\}$

UNIT($\neg x_4$, S) $M = \{\neg x_4\}$

$S_1 = \{\{x_1,x_2\}\ \{\neg x_2, \neg x_3\}\ \{\neg x_1, x_3\}\}$

SPLIT(x_1, S_1) $M = \{x_1,\neg x_4\}$

$S_1 = \{\{\neg x_2, \neg x_3\}\ \{x_3\}\}$

UNIT(x_3, S_1) $M = \{x_3, x_1,\neg x_4\}$

$S_2 = \{\{\neg x_2\}\}$

PURE($\neg x_2$, S_2) $M = \{\neg x_2,x_3, x_1,\neg x_4\}$

$S_3 = \{\ \}$

Satisfiable

SPLIT($\neg x_1$, S_1) $M = \{\neg x_1,\neg x_4\}$

$S_1 = \{\{x_2\}\ \{\neg x_2, \neg x_3\}\}$

UNIT(x_2, S_1) $M = \{x_2, \neg x_1,\neg x_4\}$

$S_2 = \{\{\neg x_3\}\}$

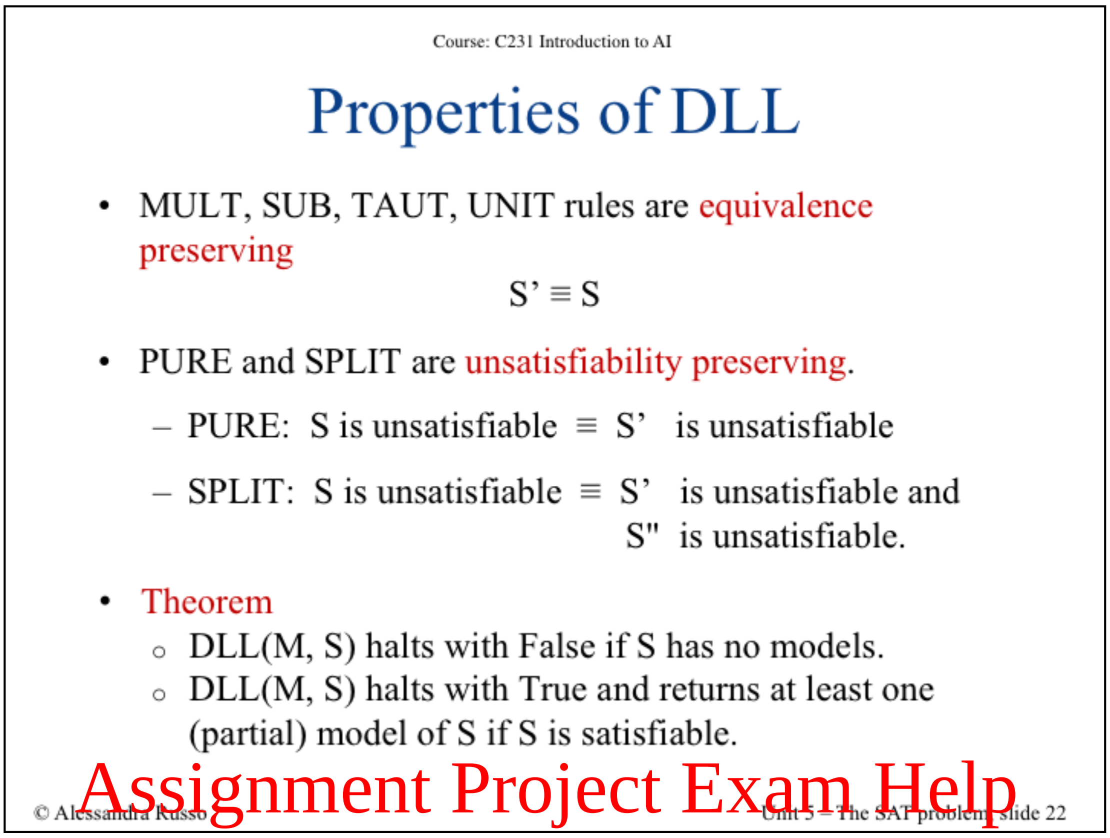
PURE($\neg x_3$, S_2) $M = \{\neg x_3,x_2,\neg x_1,\neg x_4\}$

$S_3 = \{\ \}$

Satisfiable

© Alessandra Russo Unit 5 – The SAT problem, slide 21

This is an example of application of the DLL algorithm to the given set S of clauses. Note that on the set of clauses S1 a pure literal rule could have been applied instead of the Unit rule in both branches. This would have lead though to the same result of satisfiability and same assignments. This is because the assignments in both branches are complete model. It is not always the case that the assignment M constructed during the DLL procedure constitutes a full model. Often the outcome is a partial assignment (or partial model). So alternative applications of the rules may lead to alternative assignments M at the end.


Course: C231 Introduction to AI

Properties of DLL

- MULT, SUB, TAUT, UNIT rules are equivalence preserving

$$S' \equiv S$$

- PURE and SPLIT are unsatisfiability preserving.
  - PURE: S is unsatisfiable ≡ S' is unsatisfiable
  - SPLIT: S is unsatisfiable ≡ S' is unsatisfiable and S" is unsatisfiable.
- Theorem
  - DLL(M, S) halts with False if S has no models.
  - DLL(M, S) halts with True and returns at least one (partial) model of S if S is satisfiable.

© Alessandra Russo — Unit 5 – The SAT problem, slide 22

To illustrate the statement in the theorem, consider an initial formula

$F = A \vee B$.

This is already in clausal form.

So the DLL(M, S) can be applied directly to F with initial model M={} and S=F.

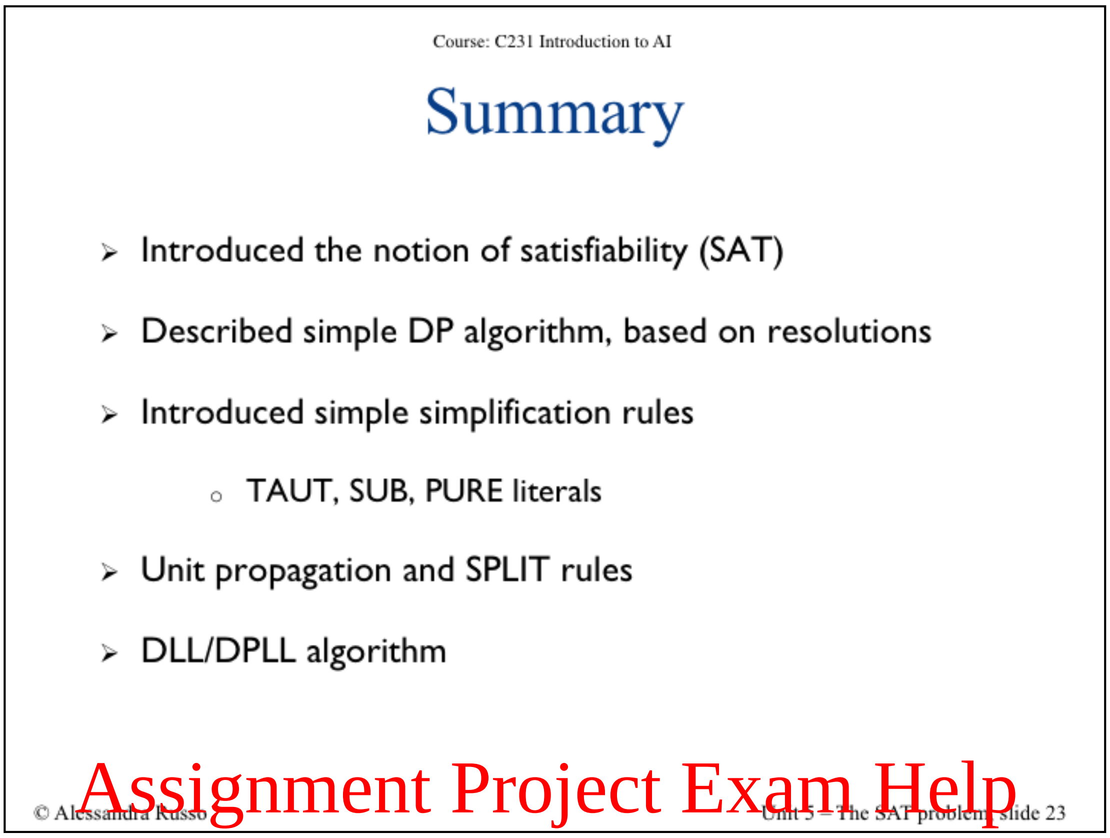
We can apply the PURE literal rule on either A or B. In the first case, we get

DLL({A}, {}) and in the second case we get DLL({B}, {}).

In each case the DLL stops returning True and the partial model M1 = {A}, or M2 = {B} respectively. Clearly {A} is a model of the initial formula F and {B} is also a model of the initial formula {F}. Both M1 and M2 are partial model as they do not specify the truth value of B (in M1), and of A in M2, respectively.

In general, any variable that is the language of the initial formula and not assigned in M at the end of the execution, can get any of the possible truth values without affecting the satisfiability of the initial formula.

For instance, M1 can be extended to either {A, B} or {A, ¬B}. Both these assignments are full models of F and satisfy F. Similarly M2 can be extended to give the two models {B, A}, {B, ¬A}, which also satisfy F.


Course: C231 Introduction to AI

Summary

- Introduced the notion of satisfiability (SAT)
- Described simple DP algorithm, based on resolutions
- Introduced simple simplification rules
    - TAUT, SUB, PURE literals
- Unit propagation and SPLIT rules
- DLL/DPLL algorithm

© Alessandra Russo Unit 5 – The SAT problem, slide 23