

1. SAT Solving Algorithms

Program Verification

ETH Zurich, Spring Semester 2017

Alexander J. Summers

Satisfiability Checking

- The **SAT problem**:

Given a propositional formula with free variables, can we find a way to assign the free variables to make the formula true?

- i.e. is the formula *satisfiable*? If so, *how*?
- Applications: test generation, planning, circuit design, program synthesis, hardware / software verification, bounded model checking
- Can we build an *efficient* tool to automatically answer this question?
 - *Such tools are called Satisfiability (SAT) solvers*
 - *SAT Solving is the classic NP-complete problem (Cook '71)*

Exponential Complexity?

- Since SAT Solving is *NP-complete*, is there hope?
- Perhaps surprisingly, many efficient SAT solvers exist
- Naïve algorithm: enumerate all assignments to n variables
- Worst case complexity is (for all known algorithms) exponential
 - in the *number of propositional variables* in the problem (e.g. 2^n assignments)
- *But...*
 - average cases encountered *in practice* can be handled (much) faster
 - real problem instances will *not* be random: exploit implicit structure
 - some variables will be tightly correlated with each other
 - some variables will be irrelevant for the difficult parts of the search

Propositional Logic

- Fix an alphabet of *propositional variables* p, q, r, p_1, p_2, \dots
- Define *propositional formulas* (\neg binds tighter than \wedge , etc.)
 $A, B ::= p \mid \top \mid \perp \mid \neg A \mid A \wedge B \mid A \vee B \mid A \Rightarrow B \mid A \Leftrightarrow B$
- A *propositional model* M maps propositional variables to truth values
- A formula A is *satisfied by a model* M , written $M \models A$, by usual semantics:
 $M \models p$ iff $M(p)$. $M \models \top$ always. $M \models \perp$ never. $M \models \neg A$ iff $M \not\models A$
- A formula A is *valid* iff for *all* models M : $M \models A$
- A formula A is *satisfiable* iff for *some* model M : $M \models A$
- A formula A is *unsatisfiable* iff not satisfiable (equivalently, $\neg A$ is valid)
- A *entails* B (written $A \models B$) iff for *all* models M : if $M \models A$ then $M \models B$
- A and B *equivalent* (written $A \equiv B$) iff for *all* models M : $M \models A$ iff $M \models B$

Propositional Equivalences

- Some important propositional logic equivalences (for all A, B, C):

$$\neg\neg A \equiv A \quad \text{and} \quad \neg\top \equiv \perp \quad \text{and} \quad \neg A \equiv A \Rightarrow \perp$$

$$A \wedge B \equiv B \wedge A \quad \text{and} \quad A \vee B \equiv B \vee A \quad \text{and} \quad A \Leftrightarrow B \equiv B \Leftrightarrow A$$

$$A \wedge \top \equiv A \quad \text{and} \quad A \wedge \perp \equiv \perp \quad \text{and} \quad A \vee \top \equiv \top \quad \text{and} \quad A \vee \perp \equiv A$$

$$(A \wedge B) \vee C \equiv (A \vee C) \wedge (B \vee C) \quad \text{and} \quad (A \vee B) \wedge C \equiv (A \wedge C) \vee (B \wedge C)$$

$$\neg(A \vee B) \equiv \neg A \wedge \neg B \quad \text{and} \quad \neg(A \wedge B) \equiv \neg A \vee \neg B$$

$$A \Rightarrow B \equiv (\neg A \vee B) \equiv \neg(A \wedge \neg B) \equiv \neg B \Rightarrow \neg A$$

$$A \Leftrightarrow B \equiv (A \Rightarrow B) \wedge (B \Rightarrow A) \equiv (A \wedge B) \vee (\neg A \wedge \neg B)$$

$$A \Rightarrow B \vee C \equiv A \wedge \neg B \Rightarrow C \quad \text{and} \quad A \wedge B \Rightarrow C \equiv A \Rightarrow \neg B \vee C$$

Make sure that you're comfortable with understanding and using these

(Equi-)Satisfiability

- Validity, unsatisfiability, equivalence and entailment questions can all be *reduced* to satisfiability questions (how?)
- We explore algorithms which answer SAT questions:
 - take a formula A as input
 - either return *unsat* (if A is unsatisfiable), or return *sat*, and (optionally) a model M such that $M \models A$
- A and B are *equi-satisfiable* iff either *both* or *neither* are satisfiable
- SAT solving algorithms typically:
 - rewrite the input formula into an *equi-satisfiable* one in standard form (CNF)
 - rewrite the formula into further *equi-satisfiable* CNF formulas
 - (often) perform a back-tracking search to explore different potential models
 - terminate when the current formula is clearly equivalent to true or false

Conjunctive Normal Form

- A *literal* is a variable or the negation of one ($p, \neg p, \dots$)
 - For a literal l we write $\sim l$ for the negation of l cancelling double negations
- A *clause* is a disjunction of (any finite number of) literals
 - e.g. $p \vee \neg q$, $q \vee r \vee \neg r$, q are all clauses
 - the *empty clause* (0 disjuncts) is defined to be \perp (why?)
 - a *unit clause* is just a single literal (exactly 1 disjunct)
 - a variable p *occurs positively in a clause* iff p is one of the clause's disjuncts
 - a variable p *occurs negatively in a clause* iff $\neg p$ is one of the clause's disjuncts
- A formula A is in *conjunctive normal form (CNF)* iff it is a conjunction of (any finite number of) clauses
 - i.e. a conjunction of disjunctions of literals
 - an *empty conjunction* (0 disjuncts) is defined to be \top (why?)
 - e.g. $(p \vee \neg q) \wedge (q \vee r \vee \neg r)$, $(p \wedge \neg q)$, q are in CNF ($p \vee \neg q \wedge q \vee r \vee \neg r$ is not - why?)

Conversion to CNF

- Any propositional formula has *equi-satisfiable CNF representation(s)*
- One approach: rewrite directly via equivalences
 - 1) Rewrite all $A \Rightarrow B$ to $\neg A \vee B$, and all $A \Leftrightarrow B$ to $(\neg A \vee B) \wedge (A \vee \neg B)$
 - 2) push all negations inwards, e.g. rewrite $\neg(A \vee B)$ to $\neg A \wedge \neg B$
 - 3) Rewrite all $\neg \neg A$ to A
 - 4) eliminate \top and \perp , e.g. rewrite $(A \vee \perp)$ to A , remove clauses containing \top
 - 5) distribute disjunctions over conjunctions,
e.g. rewrite $A \vee (B \wedge C)$ as $(A \vee B) \wedge (A \vee C)$
 - 6) Simplify: remove duplicate clauses, literals from clauses (why OK?)
In fact, each *variable* need only occur in each clause *at most once* (why?)
- Each step preserves *equivalence* with the original formula (needed?)
- What might be a practical problem with this approach?

Davis-Putnam Algorithm

- First attempt at a better-than-brute-force SAT algorithm (1960)
 - original algorithm tackles first-order logic; we present the propositional case
- We assume an input formula A in CNF, equivalently representable as:
 - a set of clauses; a set of sets of literals
- Rewrite the set of clauses until the *set* is empty, or *a clause* is empty
 - in the former case, *sat*, in the latter case, *unsat*
 - in the *sat* case we could also return a model; we omit this for now
- We will present the algorithm as working on a formula A , but will
 - consider clauses *up to reorderings* of their disjuncts (i.e. as sets)
 - we assume that each clause contains each variable *at most once*
 - requires an extra simplification step for input clauses & new ones generated

Davis-Putnam Rules

- If A is \top then return **sat**
- If A contains an empty clause \perp then return **unsat**
- If a variable p occurs either *only positively* or *only negatively* in A , delete all clauses of A in which p occurs (*pure literal rule*)
- If l is a *unit clause* in A , then update A (*unit propagation*) by:
 - *removing* all clauses which have l as a disjunct, and
 - updating all clauses in A containing $\sim l$ as a disjunct by *removing* that disjunct
- If a variable p occurs both *positively* and *negatively* in clauses of A :
 - Let $C_{\text{pos}} = \{A_1 \vee p, A_2 \vee p, \dots\}$ be the clauses in A in which p occurs *positively*, and let $C_{\text{neg}} = \{B_1 \vee \neg p, B_2 \vee \neg p, \dots\}$ be those in which p occurs *negatively*
 - Remove these two sets of clauses from A , and *replace* them with the new set $\{A_i \vee B_j \mid A_i \vee p \in C_{\text{pos}}, B_j \vee \neg p \in C_{\text{neg}}\}$

The Resolution Rule

- The last rule (called the *resolution rule*) is the most complex
- Consider the simplest case: $C_{\text{pos}} = \{A_1 \vee p\}$, $C_{\text{neg}} = \{B_1 \vee \neg p\}$
- If we choose p to be true, we will have to make B_1 true somehow
- Symmetrically, if we make p false, A_1 must be made true
- Therefore *any* model of the two original clauses must satisfy $A_1 \vee B_1$
 - conversely, any model of $A_1 \vee B_1$ can be adapted to a model of the two (how?)
- The new set of clauses from resolution can be seen as enumerating the *consequences of branching* over the choice of p as true/false
- Repeated application of this rule can grow the formula exponentially
 - effectively, we enumerate the search space for a model, *in the formula itself*
 - may consume *exponential memory*: impractical for large input problems

Davis-Putnam-Logemann-Loveland Algorithm

- The *DPLL* algorithm (1962) is an alternative to Davis-Putnam
 - Improved versions of DPLL are the basis for almost all modern SAT solvers
- Explores the search space of potential models, and *backtracks*
 - no resolution rule; potentially-exponential formula growth is avoided
- We will define the algorithm as building up a *partial model* M
 - A *partial model* assigns truth values to only some variables; a partial function
 - We will represent partial models by finite sets of literals (e.g. $M=\{p, \neg r\}$)
- The algorithm returns either (sat, M) or *unsat*
 - in the former case, M will be a model for the input formula
- The algorithm state is a pair (M, A)
 - The partial model M is our current attempt at building a model to satisfy A
 - The formula A will always be equi-satisfiable with the original

DPLL Rules

- If A is \top then return (sat, M)
- If A contains an empty clause \perp then return unsat
- If variable p occurs *only positively (negatively)* in A , delete all clauses of A in which p occurs, and update M to $M \cup \{p\}$ (to $M \cup \{\neg p\}$)
- If l is a *unit clause* in A :
 - update M to $M \cup \{l\}$, and
 - *remove* all clauses from A which have l as a disjunct, and
 - update all clauses in A containing $\sim l$ as a disjunct by *removing* that disjunct
- The rules above are all the same in DP (with model construction)
 - we replace the *resolution rule* to get DPLL...



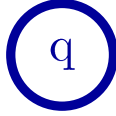

DPLL Rules

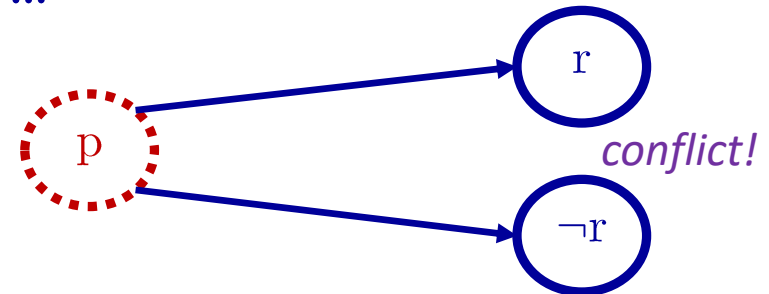
- If A is \top then return (sat, M)
- If A contains an empty clause \perp then return unsat
- If variable p occurs *only positively (negatively)* in A , delete all clauses of A in which p occurs, and update M to $M \cup \{p\}$ (to $M \cup \{\neg p\}$)
- If l is a *unit clause* in A :
 - update M to $M \cup \{l\}$, and
 - *remove* all clauses from A which have l as a disjunct, and
 - update all clauses in A containing $\sim l$ as a disjunct by *removing* that disjunct
- If a variable p occurs both *positively* and *negatively* in clauses of A :
 - Apply the algorithm to $(M \cup \{p\}, A \wedge p)$: if we get (sat, M') then return this
 - otherwise, apply the algorithm to $(M \cup \{\neg p\}, A \wedge \neg p)$ and return the result
 - here, p or $\neg p$ are called *decision literals* (choosing $\neg p$ first is also allowed)

DPLL Search Space

- The *decision literal* rule is the only one which may need undoing
 - uses of this rule can be viewed as a *search tree of decision literals*
 - other rules describe *consequences* of the decisions already made
 - backtracking should always reverse the choice of a single *decision literal*
 - the *consequences* of a backtracked *decision literal* must also be removed
- The choice of decision literal is underspecified, in the algorithm
- e.g. $(p \vee q) \wedge (\neg p \vee q \vee r) \wedge (\neg p \vee q \vee \neg r) \wedge (\neg p \vee \neg q \vee r) \wedge (\neg p \vee \neg q \vee \neg r)$
 - choosing p first as decision literal is slower than choosing e.g. $\neg p$
- Ideally, we will choose *relevant* literals as decision literals
 - those which lead us quickly to a model, or to unsatisfiability (& back-tracking)
- Is there a way to tune the exploration of this search space *on the fly*?
 - exploit problem-specific information seen so far...?

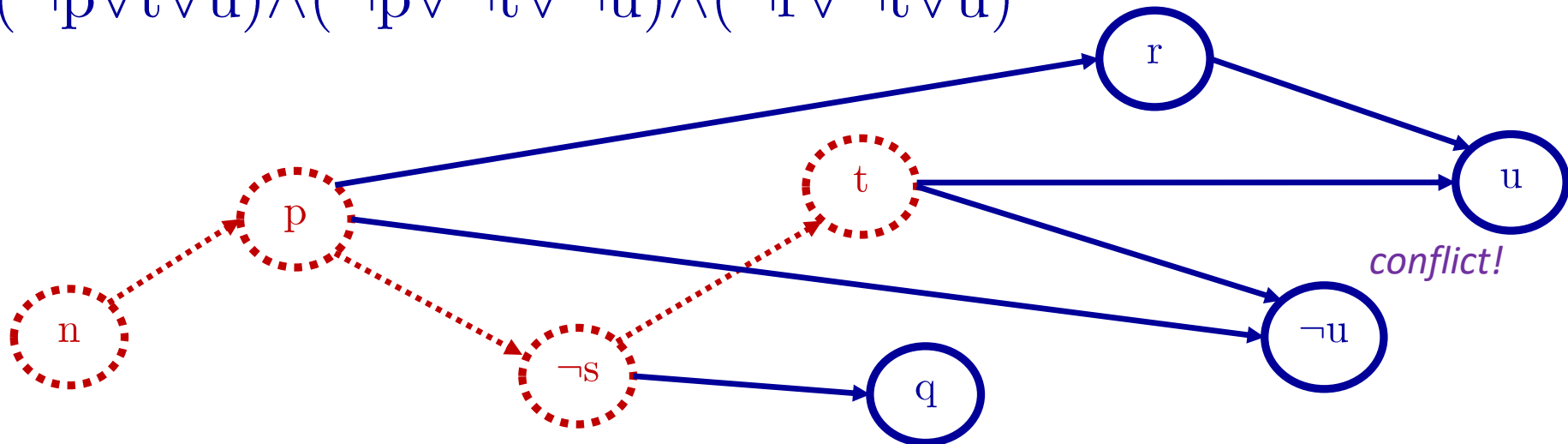
Implication Graph

- We can visualise the search for a model with a graph, as follows:
- We use dashed red  nodes to indicate *decision literals*
 - we use dashed red edges  to indicate the *order of decisions* made
- We add blue  nodes for literals added due to *unit propagation*
 - modify the algorithm to *remember* any literals removed from clauses
 - when a unit clause is selected, we check it for *previously removed* literals
 - each such literal was removed due to adding its negation to the model so far
 - the current unit propagation step is *possible due to* these previous choices
 - we record such dependencies via blue  edges
 - e.g. with an input including $(\neg p \vee r) \wedge (\neg p \vee \neg r)$...
 - when we add both a literal and its negation, we record a conflict (need to backtrack)



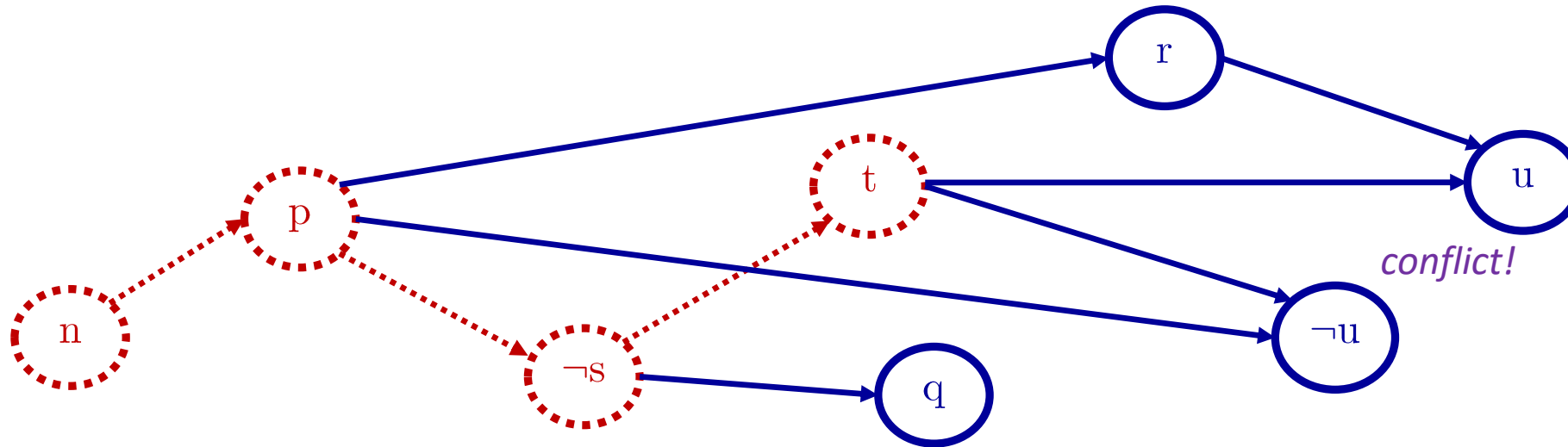
Implication Graph

- We can visualise the search for a model with a graph, as follows:
- We use dashed red \textcircled{p} nodes to indicate *decision literals*
 - we use dashed red edges $\cdots\rightarrow$ to indicate the *order of decisions* made
- We add blue \textcircled{q} nodes for literals added due to *unit propagation*
 - for each disjunct *previously removed* (if any) from the unit clause, add an edge \longrightarrow from the node for the negation of the disjunct to the new node
- e.g. $(n \vee p) \wedge (\neg n \vee p \vee q) \wedge (\neg n \vee p \vee \neg q) \wedge (\neg p \vee r) \wedge (\neg u \vee t) \wedge (\neg r \vee \neg s \vee t) \wedge (q \vee s) \wedge (\neg p \vee t \vee u) \wedge (\neg p \vee \neg t \vee \neg u) \wedge (\neg r \vee \neg t \vee u)$



Implication Graph - Observations

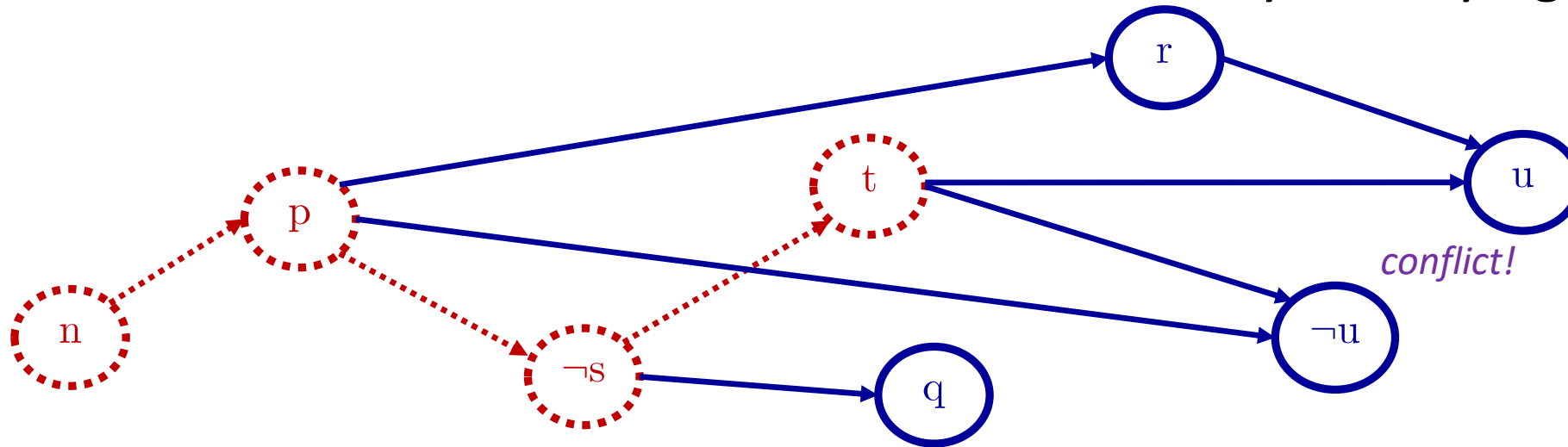
- e.g. $(n \vee p) \wedge (\neg n \vee p \vee q) \wedge (\neg p \vee r) \wedge (\neg u \vee t) \wedge (\neg r \vee \neg s \vee t) \wedge (q \vee s) \wedge (\neg p \vee t \vee u) \wedge (\neg p \vee \neg t \vee \neg u) \wedge (\neg r \vee \neg t \vee u)$



- Only *some* decision literals (here **p** and **t**) are relevant for a conflict
 - we could rediscover the same conflict in (many) other search branches
 - backtracking later to switch **¬s** will be wasteful: the same conflict will exist
- Idea: it would have been better to branch on **t** before **s**, here...

Extension: Back-jumping

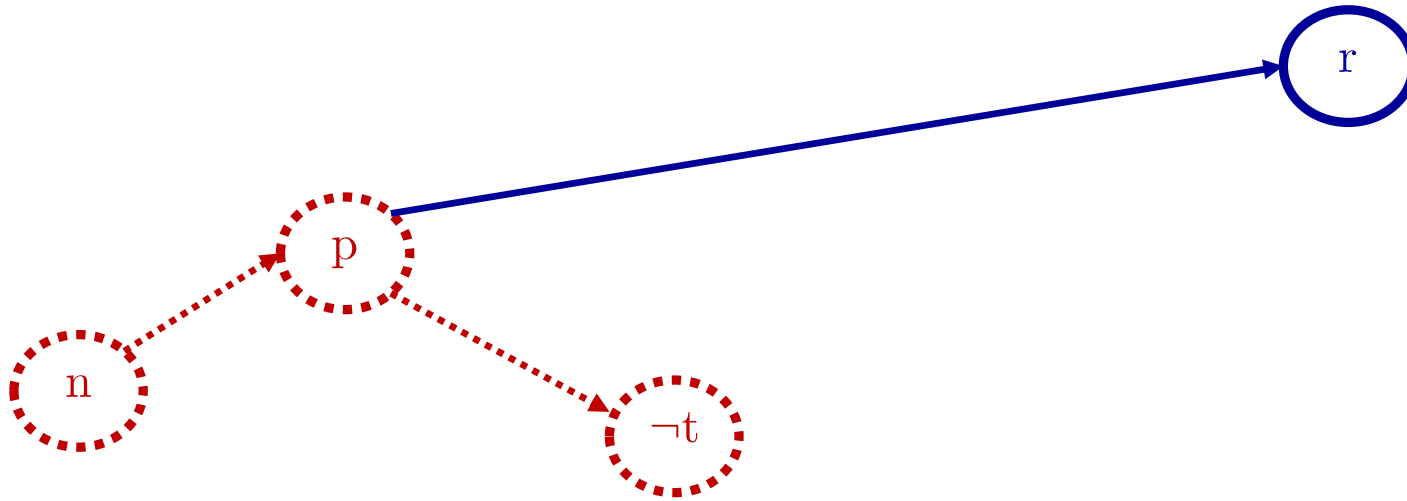
- Keep decision literals in an explicit stack (cf. $\cdots \rightarrow$ edges)
- When a conflict is found, undo one decision by modifying the stack...



- Identify the *relevant* decision literals (here p and t)
- Pop *only one relevant literal* + as many others as possible (t and $\neg s$)

Extension: Back-jumping

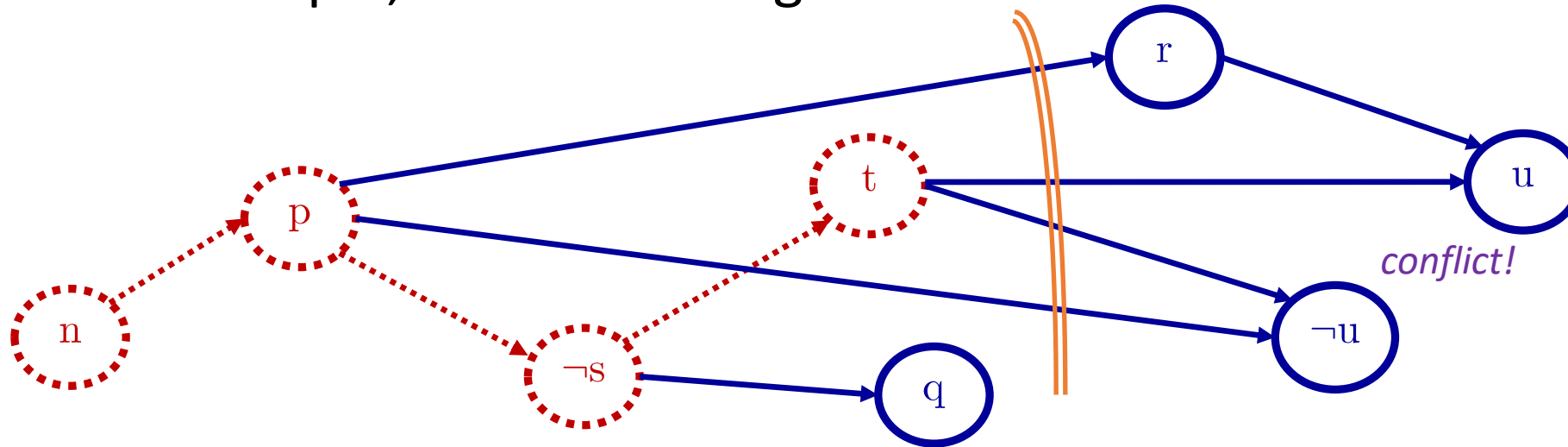
- Keep decision literals in an explicit stack (cf. $\cdots \rightarrow$ edges)
- When a conflict is found, undo one decision by modifying the stack...



- Identify the *relevant* decision literals (here p and t)
- Pop *only one relevant literal* + as many others as possible (t and $\neg s$)
- Push the *negation of the relevant literal* popped ($\neg t$) and continue

Extension: Clause Learning

- Back-jumping prunes local subtrees, but conflict could still repeat
- For example, after switching n to $\neg n$ we could rebuild the conflict



- This is avoided by *clause learning* when a conflict is reached:
 - draw a **boundary** dividing conflicting literals from the decision literals (a “cut”)
 - for the nodes with outgoing edges crossing the boundary, compute the disjunction of the negations of their literals (here, $\neg p \vee \neg t$)
 - when back-jumping, *add this new clause* to the formula to be satisfied

Conflict-Driven Clause Learning Algorithm

- *Conflict-Driven Clause Learning (CDCL)* is the extension of DPLL with:
 - back-jumping (decision literals as an explicit stack) and clause learning
- Clause learning can be performed with various strategies
 - Different ways to cut the implication graph lead to different clauses
 - e.g. “1-UIP strategy”: working backwards from conflicting literals, find the first (“latest”) node which is on all paths from the decision literal to be changed.
- Learned clauses are always implied by the original clause set
 - but extra clauses can *trigger unit propagation earlier*, pruning branches
 - on the other hand, too many clauses may slow down algorithm operations
 - In practice, clauses are also *periodically removed* during SAT solver runs
- CDCL algorithms are used in almost all modern SAT solvers
 - strictly better than DPLL in practice (can be the same on random examples)

SAT Solving Research

- SAT Solving has been an active CS research area for >40 years
- SAT research involves modifying all aspects of the algorithms, e.g.
 - improved *encodings* of the input formula into CNF form
 - techniques for *simplifying* the formula before and during processing
 - heuristics for choosing the *order* of decision literals (e.g. recent activity)
 - deciding which clauses to *learn* (which boundary in the implication graph?)
 - deciding when to *discard* learned clauses (too many will slow operations)
 - choosing to *restart* the search in a different order (retaining learned clauses)
 - specialised *data structures* for fast operations (especially unit propagation)
 - work on *parallel SAT solving* (unlike DPLL, splitting CDCL problems is hard)
- There are *SAT competitions* where industrial and academic tools race
 - different categories: industrial problems, random problems, hand-crafted etc.

SAT Solving Algorithms - Summary

- We have seen three different (but related) algorithms: DP, DPLL, CDCL
- All work on a CNF representation of a propositional formula
- DPLL beats DP by avoiding exponential memory consumption
- CDCL beats DPLL by pruning the search space on the fly
- A wide variety of problem domains are tackled by modern SAT solvers
 - In practice, *how to encode* a problem as a SAT problem is a critical concern
 - Different encodings yield (exponentially) different performance in practice
- In the next section: encoding varieties of problems as SAT problems

SAT Solving Algorithms – Some References

- *Handbook of Satisfiability*. Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh (2009)
- DP: *A Computing Procedure for Quantification Theory*. Martin Davis, Hilary Putnam (1960).
- DPLL: *A Machine Program for Theorem Proving*. Martin Davis, George Logemann, Donald Loveland (1962).
- CDCL:
 - *GRASP-A New Search Algorithm for Satisfiability*. J.P. Marques-Silva, Karem A. Sakallah (1996)
 - *Using CSP look-back techniques to solve real world SAT instances*. Roberto J. Bayardo Jr., Robert C. Schrag (1997)
- General SAT Developments:
 - *Chaff: engineering an efficient SAT solver*. Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, Sharad Malik
 - *SAT-solving in practice*. Koen Claessen, Niklas Een, Mary Sheeran, Niklas Sörensson (2008)
 - *Successful SAT Encoding Techniques*. Magnus Björk (2009)
- Other teaching material:
 - *SAT Solvers: Theory and Practice*. Clark Barrett
 - *SAT-Solving: From Davis-Putnam to Zchaff and Beyond (SAT Basics)*. Lintao Zhang
- Latest SAT-solving Competition: *Proceedings of SAT Competition 2016*. Tomáš Balyo, Marijn J. H. Heule