

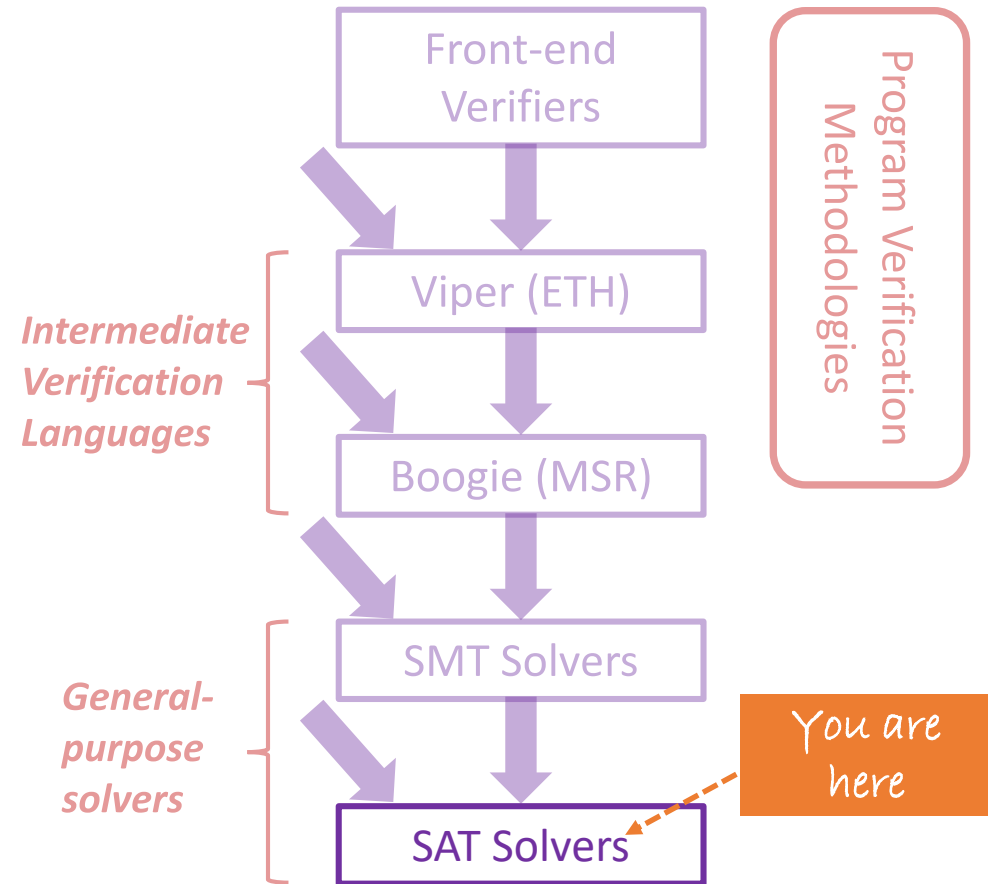
1. SAT Solving Algorithms

Program Verifiers and Program Verification

UBC Winter Term 1 (2023)

Alexander J. Summers

Next up: SAT Solving Algorithms



SAT solving : the problem

- The **SAT problem**:

Given a propositional formula containing constant symbols, can we find a way to assign the constants to make the formula true?

- i.e. is the formula *satisfiable*? If so, *how*?
 - $p \wedge (\neg q \vee \neg p)$ *satisfiable*: choose p true and q false
 - $p \wedge (\neg q \vee \neg p) \wedge q$ *unsatisfiable*
 - $(p \Rightarrow \neg r \wedge \neg(t \Leftrightarrow u)) \wedge (\neg p \Rightarrow \neg n \vee (s \wedge n)) \wedge ((\neg q \wedge (r \vee t)) \vee (q \wedge s)) \wedge (u \Rightarrow t) \wedge (t \Rightarrow u \vee r)$?
 - ... what about formulas with *thousands or millions of constant symbols*... ?
- This week: how are *SAT solving algorithms* made *efficient in practice*?
 - *Why can this ever work?* SAT solving is the *classic NP-complete problem*
 - We'll cover *four simple but powerful ideas* employed in modern algorithms

Example SAT Solving Applications

Circuit Synthesis

Inputs + outputs
Design constraints
Behaviour specification

SAT solver

Gate-level design

A thick purple arrow points from the input text on the left to the output text on the right.

Hardware Verification

Circuit design
Inputs + outputs
Error specification

SAT solver

Error-causing inputs (bug)

A thick purple arrow points from the input text on the left to the output text on the right.

Automated Planning

Transition system
Goal states
Number of steps

SAT solver

Sequence of steps

A thick purple arrow points from the input text on the left to the output text on the right.

Dependency Analysis

Software packages
Dependencies
Version constraints

SAT solver

Compatible set of packages

A thick purple arrow points from the input text on the left to the output text on the right.

Solving Puzzles as SAT Problems

Any finite-state puzzle can be encoded as a SAT problem

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

State space
Physical constraints
Puzzle rules
Given inputs



*Puzzle
solution*

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. naïve enumeration yields 2^n candidate models to check
- *But...*
 - average cases encountered *in practice* can be handled (much) faster
 - real problem instances will usually *not* be random: exploit implicit structure
 - some variables will be *tightly correlated* with each other, some *irrelevant*
 - efficient algorithms tune search *dynamically*, based on formula structure

Revision: Propositional Logic

- Fix an alphabet of *(Boolean) constant symbols* p, q, r, p_1, p_2, \dots
- Define *propositional formulas* (\neg binds tighter than \wedge , tighter than \vee , 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 is a partial map from constant symbols to truth values
- A formula A is *satisfied by a model* M , written $M \models A$, via usual semantics:
 $M \models p$ iff $M(p)$. $M \models \top$ always. $M \models \perp$ never. $M \models \neg A$ iff $M \not\models A$. etc. ...
- 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 $M \models B$
 - Important example property: a formula A is *unsatisfiable* iff $A \models \perp$
- A and B *equivalent* (written $A \equiv B$) iff for *all* models M : $M \models A$ iff $M \models B$

Revision: 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

For propositional logic lecture notes, see e.g. CPSC 121 course materials, or e.g. Ian Hodkinson's excellent Logic course material (p. 53-58):

https://web.archive.org/web/20221118043551/https://www.doc.ic.ac.uk/~imh/teaching/140_logic/140.pdf

Revision: Conjunctive Normal Form

- A *literal* is a constant 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 constant p *occurs positively in a clause* iff p is one of the clause's disjuncts
 - constant 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?)

Reduction to Satisfiability

- Validity, unsatisfiability, entailment and equivalence questions can all be *reduced* to (un)satisfiability questions (mini exercise: prove this!)
- A is *valid* iff $\neg A$ is *unsatisfiable*
- A *entails* B ($A \models B$) iff $A \wedge \neg B$ is *unsatisfiable*
- A and B are *equivalent* iff A *entails* B and B *entails* A ,
 - or, equivalently (why?) $A \wedge \neg B$ is *unsatisfiable* and $B \wedge \neg A$ is *unsatisfiable*
 - or, equivalently (why?) $A \wedge \neg B \vee B \wedge \neg A$ is *unsatisfiable*

SAT Solving Algorithm Structure

- We explore algorithms which take a formula A as input and:
 - either return **unsat** (if A is unsatisfiable), or return **sat**, and a model M such that $M \models A$
- Such SAT solving algorithms typically combine three kinds of steps:
 - rewrite formula A into an **equivalent** one (e.g. in a standard form such as CNF)
 - terminate when the current problem is clearly equivalent to true or false
 - split (e.g. via **back-tracking search**) the problem of finding a model for A into any number of sub-problems for formulas A_1, A_2, \dots such that:
 - Any model found for **any one of the sub-problems** can be converted into a model for A
 - If **no model exists for any** of the sub-problems A_1, A_2, \dots , then no model exists for A
- Often, a model is built up by tracking a current **partial model** M
 - A **partial model** assigns truth values to only some constants; a partial function
 - We represent partial models using **finite sets of literals** (e.g. $M = \{p, \neg r\}$)
 - Tracks pairs (M, A) of current partial model M and formula A to satisfy

Simple 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. $A \vee (B \wedge C)$ to $(A \vee B) \wedge (A \vee C)$
 - 6) Remove duplicate clauses, duplicate literals from clauses (why is this OK?)

In fact, each *constant* need only occur in each clause *at most once* (why?)
- Each step preserves *equivalence* with the original formula (needed?)
- Can you see a potential practical problem with this approach?
 - (see the *exercises* for an improved approach, due to Tseitin)

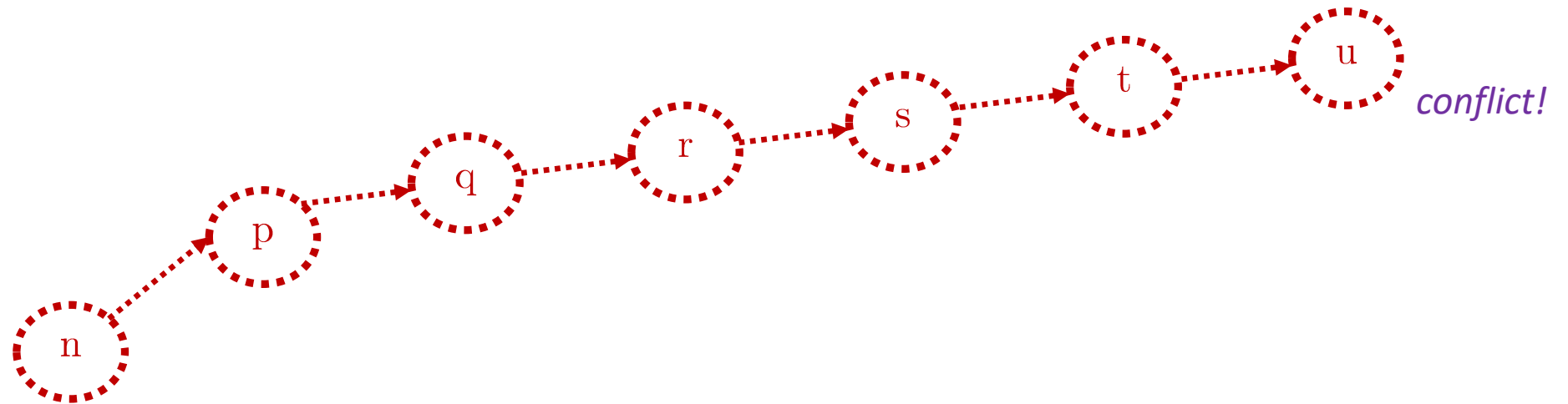
Solving SAT problems

- Is the following formula satisfiable (7 constants: n, p, q, r, s, t, u)?

$$(p \Rightarrow \neg r \wedge \neg(t \Leftrightarrow u)) \wedge (\neg p \Rightarrow \neg n \vee (s \wedge n)) \wedge \\ ((\neg q \wedge (r \vee t)) \vee (q \wedge s)) \wedge (u \Rightarrow t) \wedge (t \Rightarrow u \vee r)$$

- How would you check this *by hand*? What about some *algorithm*?
- One observation: *checking* a solution is much easier than finding one
 - e.g. evaluate the formula for given true/false choices with a linear pass
- Naïve algorithm: *enumerate* all such choices, then check if any work...

Naïve backtracking search



$$(p \Rightarrow \neg r) \wedge \neg(t \Leftrightarrow u) \wedge$$

$$(\neg p \Rightarrow \neg n \vee (s \wedge n)) \wedge$$

$$((\neg q \wedge (r \vee t)) \vee (q \wedge s)) \wedge$$

$$(u \Rightarrow t) \wedge$$

$$(t \Rightarrow u \vee r)$$

- This choice *fails* (e.g. $p \Rightarrow \neg r$ is not true)
- A simple naïve search will flip the *last* decision **u** on the stack, and try again
- But the *reason* for this failure is *further up*
- Can we efficiently *detect failures earlier*?

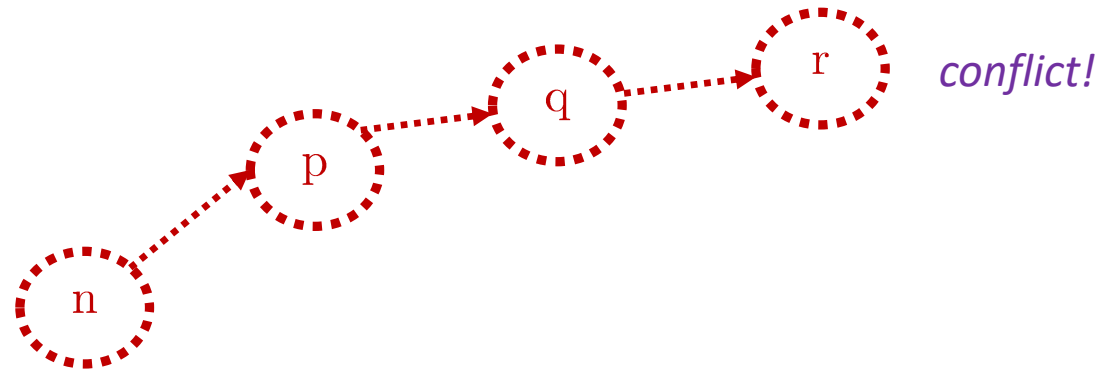
Conjunctive Normal Form

$$\begin{aligned}
 & (n \vee p) \wedge \\
 & (\neg n \vee p \vee s) \wedge \\
 & (\neg p \vee \neg r) \wedge \\
 & (\neg u \vee t) \wedge \\
 & ((p \Rightarrow \neg r \wedge \neg(t \Leftrightarrow u)) \wedge \\
 & ((\neg p \Rightarrow \neg s) \wedge (s \wedge n)) \wedge \\
 & (((\neg q \wedge (r \vee t)) \vee (q \wedge s)) \wedge \\
 & ((u \Rightarrow t) \wedge t \vee \neg u) \wedge \\
 & (t \Rightarrow u \vee r) \vee u)
 \end{aligned}$$

- Want to avoid evaluating full formula *at every search step*
- We convert the formula to *Conjunctive Normal Form*
- A conjunction of disjunctions
 - the disjunctions are *clauses*
- Each disjunct in a clause is a literal: *constant or its negation*
- The formula is true exactly when: for each clause, *at least one disjunct is made true*

(Idea 1) Early Conflict Detection

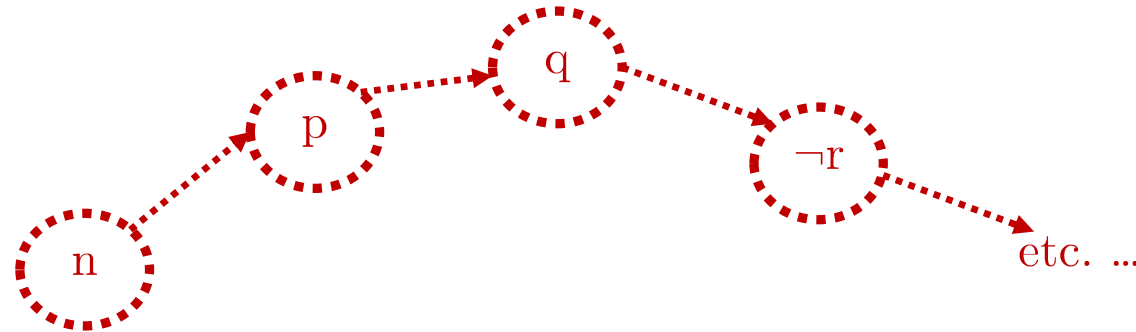
- ✓ $(n \vee p) \wedge$
- ✓ $(\cancel{n} \vee p \vee s) \wedge$
- $(\cancel{p} \vee \cancel{r}) \wedge$
- $(\neg u \vee t) \wedge$
- ✓ $(r \vee q \vee t) \wedge$
- $(\cancel{q} \vee s) \wedge$
- $(\cancel{p} \vee t \vee u) \wedge$
- $(\cancel{p} \vee \neg t \vee \neg u) \wedge$
- ✓ $(r \vee \neg t \vee u)$



- During search, mark the clauses which are *already satisfied* ✓
- Dually, cross out the disjuncts *made false* by our decisions so far
- We can backtrack as soon as *any clause has all disjuncts crossed out*

(Idea 1) Early Conflict Detection

✓ $(n \vee p) \wedge$
✓ $(\cancel{n} \vee p \vee s) \wedge$
✓ $(\cancel{p} \vee \neg r) \wedge$
 $(\neg u \vee t) \wedge$
 $(\cancel{r} \vee q \vee t) \wedge$
 $(\cancel{\neg q} \vee s) \wedge$
 $(\cancel{\neg p} \vee t \vee u) \wedge$
 $(\cancel{\neg p} \vee \neg t \vee \neg u) \wedge$
 $(\cancel{r} \vee \neg t \vee u)$



- Early backtracking *prunes a large portion of the search space*
- The *sooner* we reach a conflict, the more of the search space is pruned
- Ideally, *order decisions* to reach conflicts fast (how? These orders are only *discovered during the search*)

Recall: Solving Puzzles as SAT Problems

Idea: we don't usually solve puzzles by random search (alone)...

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

State space
Physical constraints
Puzzle rules
Given inputs

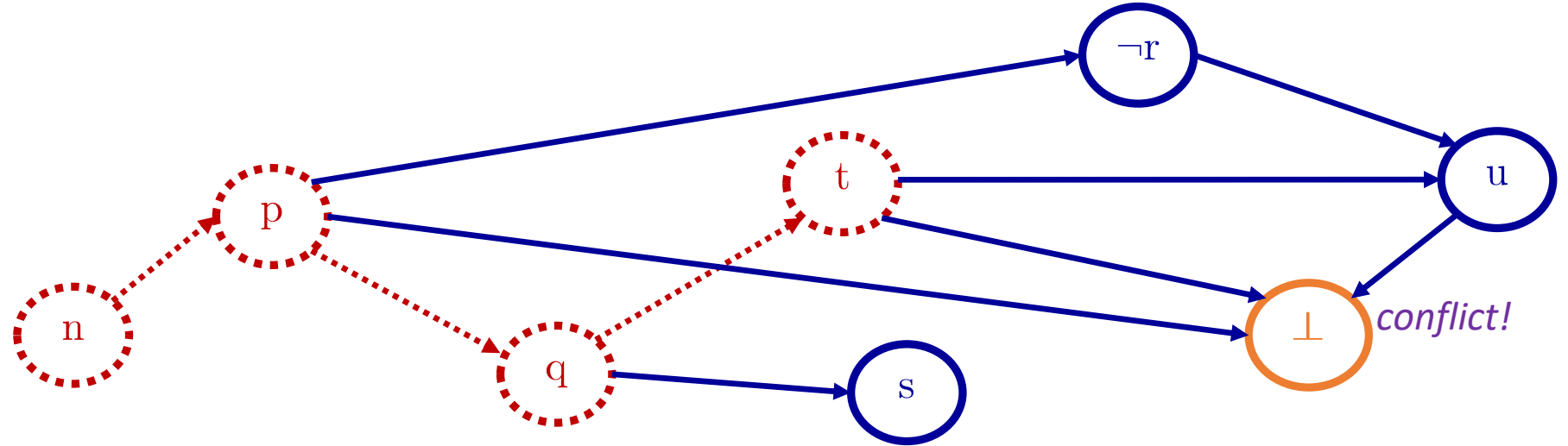


*Puzzle
solution*

How do you go about solving a Sudoku?

(Idea 2) Unit Propagation

- ✓ $(n \vee p) \wedge$
- ✓ $(\cancel{n} \vee p \vee s) \wedge$
- ✓ $(\cancel{p} \vee \neg r) \wedge$
- ✓ $(\cancel{u} \vee t) \wedge$
- ✓ $(\cancel{r} \vee q \vee t) \wedge$
- ✓ $(\cancel{q} \vee s) \wedge$
- ✓ $(\cancel{p} \vee t \vee u) \wedge$
- $(\cancel{p} \vee \cancel{t} \vee \cancel{u}) \wedge$
- ✓ $(\cancel{r} \vee \cancel{t} \vee u)$



- We add simple *deduction steps* to our search
- Detect when a clause has *only one disjunct remaining*
 - such a clause is called a *unit clause*
- As soon as this happens, add the *remaining* disjunct
 - represented with a **blue (solid) node**
 - add **blue (solid) edges** from the nodes *causing each other disjunct to be crossed-out*
- This deduction technique is called *unit propagation*

Davis-Putnam-Logemann-Loveland (DPLL) Algorithm

- The *DPLL* algorithm is one of the first SAT solving algorithms (1962)
 - Improved versions of DPLL are the basis for almost all *modern SAT solvers*
- We assume an input formula *A* in CNF, equivalently representable as:
 - a *set* of clauses, or a *set of sets* of literals
- Algorithm rewrites clauses until the *set* is empty, or *a clause* is empty
 - in the former case, *sat*, in the latter case, *unsat*
 - in *sat* case we also return a model; the algorithm builds one up incrementally
- We will present the algorithm as operating on a *formula A*, but will
 - consider clauses *up to reorderings* of their disjuncts (i.e. as sets of literals)
- We assume all clauses contain each constant *at most once*
 - this needs extra simplification steps for input clauses & new ones generated

DPLL Algorithm Basics

- DPLL explores the search space of potential models, and *backtracks*
 - e.g. guess that p should be true, if we find no model exists, try $\neg p$ instead
- We will define the algorithm as building up a *partial model* M
 - A *partial model* assigns truth values to only some constants; a partial function
 - We represent partial models using *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) with the following properties:
 - The partial model M is our *current attempt* at building a model to satisfy A
 - Initially our partial model is empty $\{\}$ and A is our input formula (in CNF)
 - The formulas A are always such that any model found for A by extending M is guaranteed to be a *model for the original input formula* (see the exercises)






DPLL Algorithm Rules (cf. Ideas 1 and 2 above)

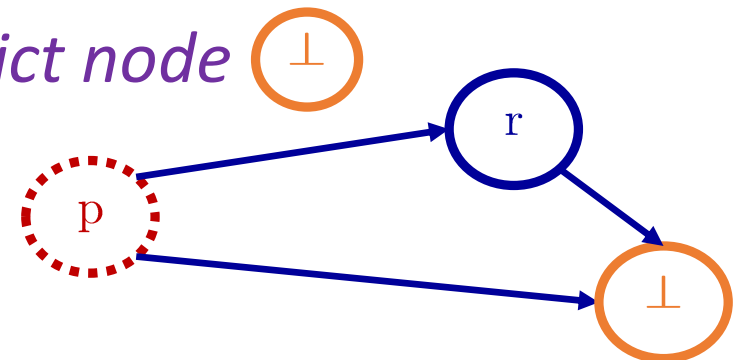
- If A is \top then return (**sat**, M)
- If A contains an empty clause \perp then return **unsat**
- *Pure literal rule*: If p occurs *only positively (negatively)* in A , delete clauses of A in which p occurs, update M to $M \cup \{p\}$ (to $M \cup \{\neg p\}$)
- *Unit propagation*: If l is a *unit 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
- *Decision*: Pick any literal l in A for which neither l nor $\sim l$ are in M
 - Apply the algorithm to $(M \cup \{l\}, A \wedge l)$: if we get (**sat**, M') then return this
 - otherwise, apply the algorithm to $(M \cup \{\sim l\}, A \wedge \sim l)$ and return the result
 - here, l is called the *decision literal* (the one we try out adding to the model)

DPLL Search Space





- The *Decision* rule is the only one which may need undoing
 - uses of this rule can be visualised with a *search tree of decision literals*
 - Unit propagation computes *consequences* of the decisions already made
 - backtracking will always reverse the choice of a single *decision literal*
 - the *consequences* of a backtracked *decision literal* must also be reverted
- The *choice* of decision literal is underspecified, in the algorithm
 - different implementations may pick different strategies (also true/false first?)
- e.g. $(\neg p \vee q) \wedge (\neg p \vee \neg r) \wedge (\neg p \vee s) \wedge (p \vee t \vee s) \wedge (p \vee \neg t \vee \neg s)$
 - choosing *p* first as decision literal is faster than choosing *q* or *s* etc. (why?)
- Ideally, we will choose the *most relevant* literals as decision literals:
 - those which lead us *quickly* to a model, or to unsatisfiability (& backtracking)
- Is there a way to tune the exploration of this search space *on the fly*?
 - somehow exploit problem-specific information uncovered *during* the search?

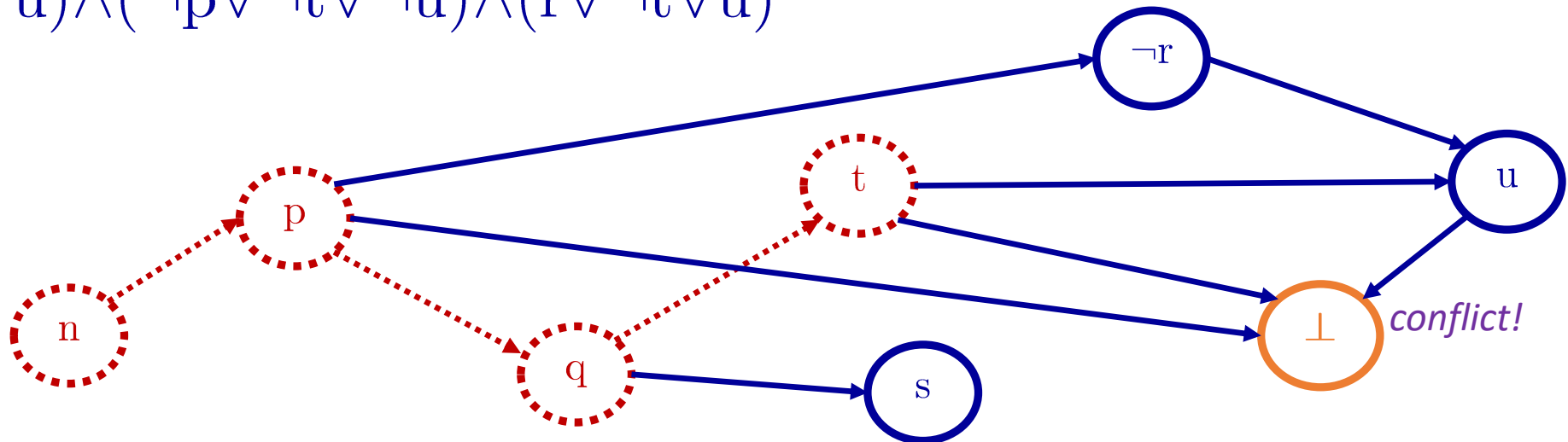
The Implication Graph

- We visualised 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 l was removed when adding its negation $\sim l$ to the model
 - the current unit propagation step is *possible due to* these previous choices
 - we record such dependencies via blue  edges (from each $\sim l$ node)
- when a clause becomes empty, add a *conflict node* 
 - we add blue edges to the node analogously
 - e.g. with input $(\neg p \vee r) \wedge (\neg p \vee \neg r)$ we can get:



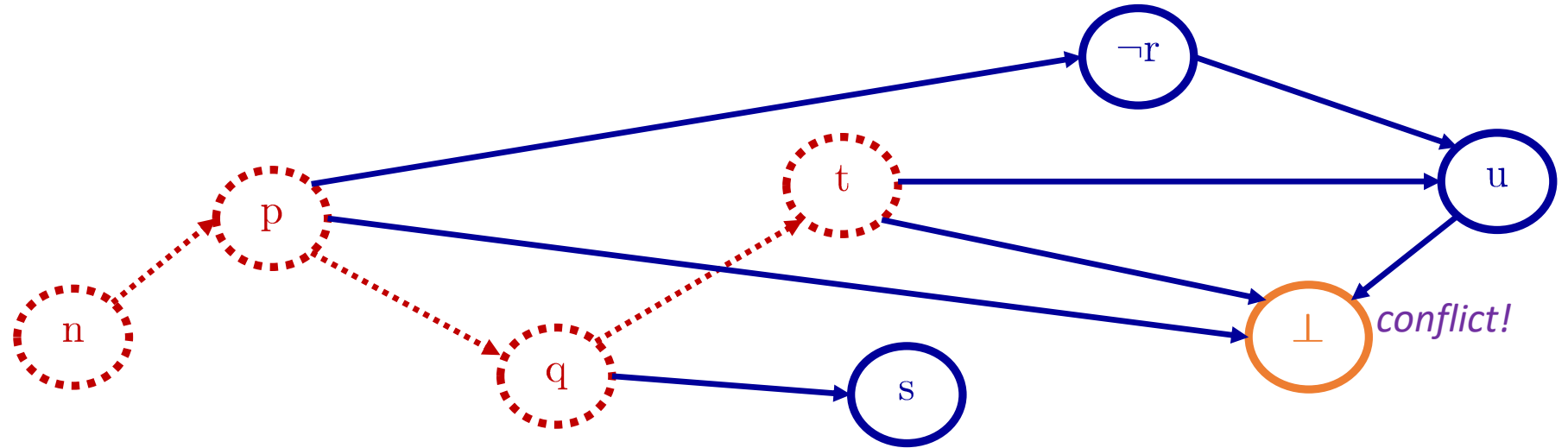
Implication Graph

- We visualised 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*
 - for each disjunct *previously removed* (if any) from the unit clause, add an edge  from the node for the negation of the disjunct to the new node
- e.g. $(n \vee p) \wedge (\neg n \vee p \vee s) \wedge (\neg p \vee \neg r) \wedge (\neg u \vee t) \wedge (r \vee q \vee t) \wedge (\neg q \vee s) \wedge (\neg p \vee t \vee u) \wedge (\neg p \vee \neg t \vee \neg u) \wedge (r \vee \neg t \vee u)$



Unit Propagation – Extended Backtracking

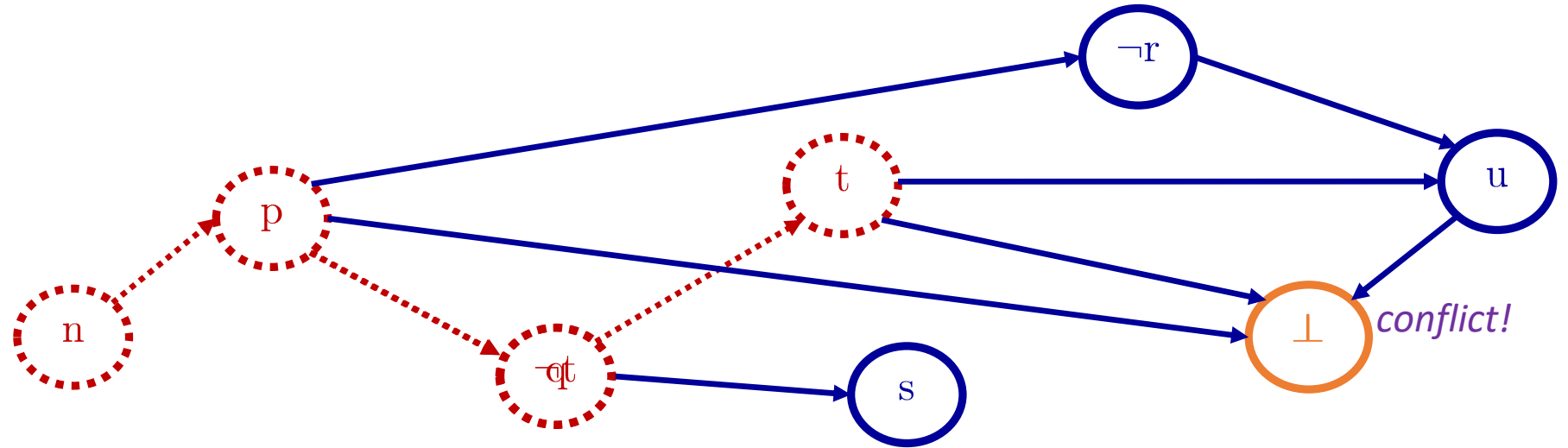
- ✓ $(n \vee p) \wedge$
- ✓ $(\cancel{n} \vee p \vee s) \wedge$
- ✓ $(\cancel{p} \vee \neg r) \wedge$
- ✓ $(\cancel{u} \vee t) \wedge$
- ✓ $(\cancel{r} \vee q \vee t) \wedge$
- ✓ $(\cancel{q} \vee s) \wedge$
- ✓ $(\cancel{p} \vee t \vee u) \wedge$
- $(\cancel{p} \vee \cancel{t} \vee \cancel{u}) \wedge$
- ✓ $(\cancel{r} \vee \cancel{t} \vee u)$



- When we find a conflict, we *backtrack a (red) decision*
- Nodes *reachable from it by blue edges* also get removed
 - their *causes are no-longer in the graph*
- Blue edges define *relevant* decisions for node/conflict
- Here, only **p** and **t** were relevant for the conflict
 - Ideally, we would have tried **t** directly after **p**
 - Flipping **q** later we can rebuild the *same conflict* ☹

(Idea 3) Back-jumping

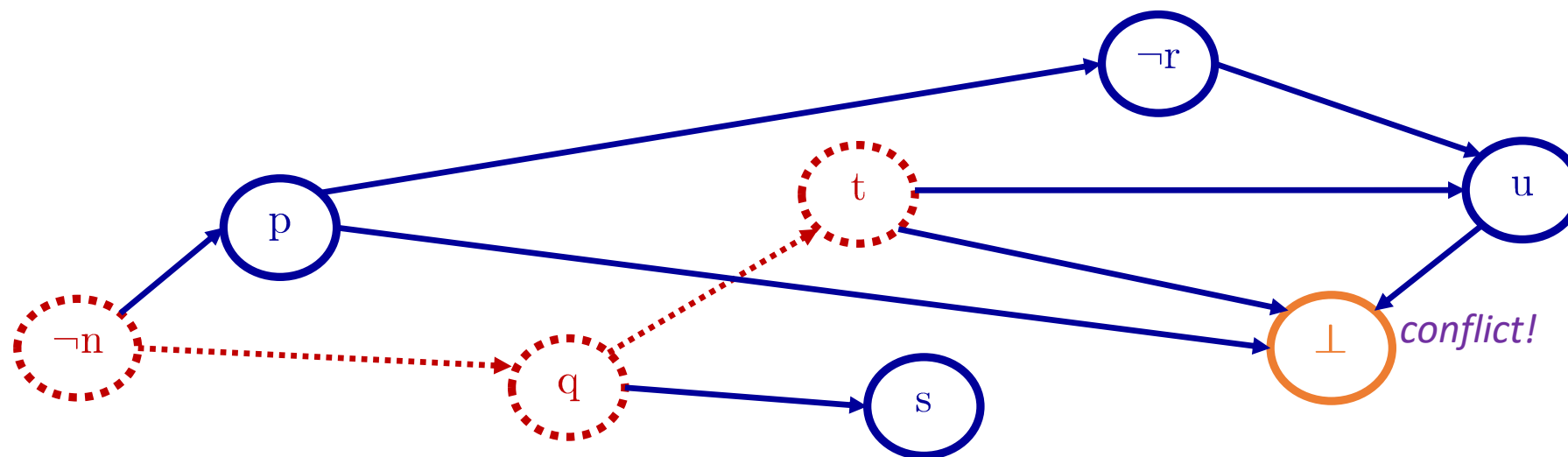
- ✓ $(n \vee p) \wedge$
- ✓ $(\cancel{n} \vee p \vee s) \wedge$
- ✓ $(\cancel{p} \vee \neg r) \wedge$
- ✓ $(\cancel{u} \vee t) \wedge$
- ✓ $(\cancel{r} \vee q \vee t) \wedge$
- ✓ $(\cancel{q} \vee s) \wedge$
- ✓ $(\cancel{p} \vee t \vee u) \wedge$
- $(\cancel{p} \vee \cancel{t} \vee \cancel{u}) \wedge$
- ✓ $(\cancel{r} \vee \cancel{t} \vee u)$



- New idea: *rearrange the stack when we backtrack*
- Pop as far as possible only removing *1 relevant decision*
 - Here, p and t are relevant; q is not
- Flip only the *relevant literal popped*, and continue
- End effect: as if we had *originally* chosen p and then t
 - We prune a larger search space (independent of q)

More Repeated Work...

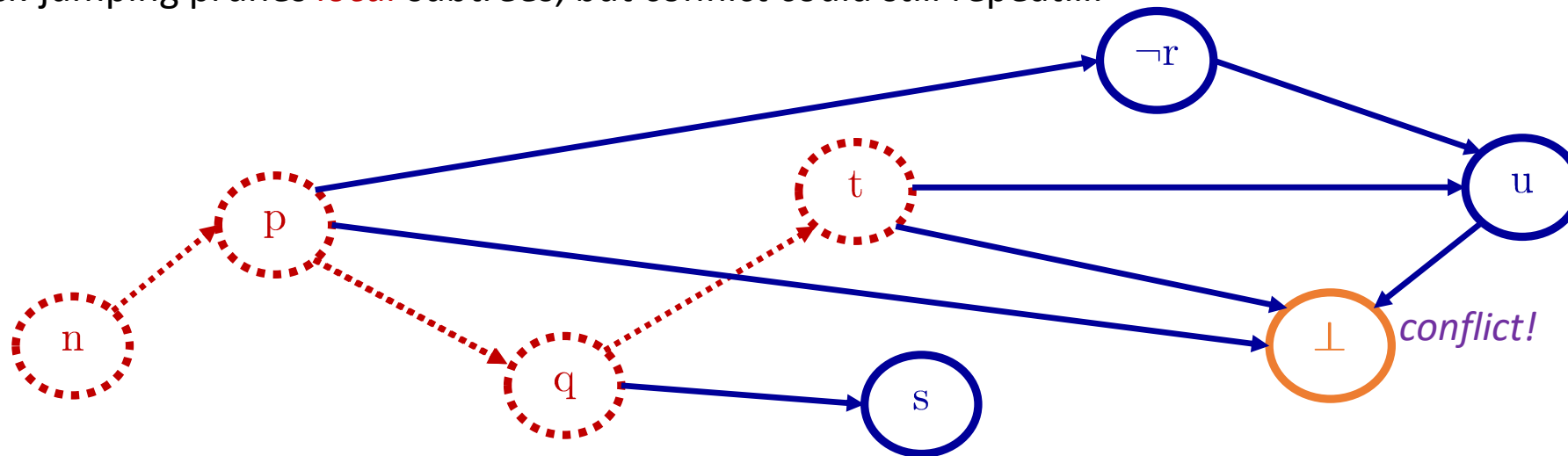
- ✓ (~~n~~ ∨ p) ∧
- ✓ (¬n ∨ p ∨ s) ∧
- ✓ (~~¬p~~ ∨ ¬r) ∧
- ✓ (~~¬u~~ ∨ t) ∧
- ✓ (~~r~~ ∨ q ∨ t) ∧
- ✓ (~~¬q~~ ∨ s) ∧
- ✓ (~~¬p~~ ∨ t ∨ u) ∧
- (~~¬p~~ ∨ ~~¬t~~ ∨ ~~¬u~~) ∧
- ✓ (~~r~~ ∨ ~~¬t~~ ∨ u)



- Suppose we backtrack right back to flipping n to $\neg n$
- Now, we can rebuild the “same” conflict over again...
- How can we avoid *ever* repeating this same conflict?

(Idea 4) Clause Learning

Back-jumping prunes *local* subtrees, but conflict could still repeat....



- New idea: *change the formula to rule out this conflict (ever)*
- (basic version) Take all nodes *with direct edges* to the conflict, negate each and form a *new clause* from these disjuncts
 - e.g. here **p** and **t** and **u**; we form the clause $\neg p \vee \neg t \vee \neg u$
- Conjoin this new clause to the *formula, permanently*
 - Idea: force *at least one decision* to be made differently
- We cut out *multiple symmetric areas* of the search space
- But $\neg u$ seems redundant here, as **p** and **t** force **u** above... 34

$$\checkmark (n \vee p) \wedge$$

$$\checkmark (\cancel{n} \vee p \vee s) \wedge$$

$$\checkmark (\cancel{p} \vee \neg r) \wedge$$

$$\checkmark (\cancel{u} \vee t) \wedge$$

$$\checkmark (\cancel{r} \vee q \vee t) \wedge$$

$$\checkmark (\cancel{q} \vee s) \wedge$$

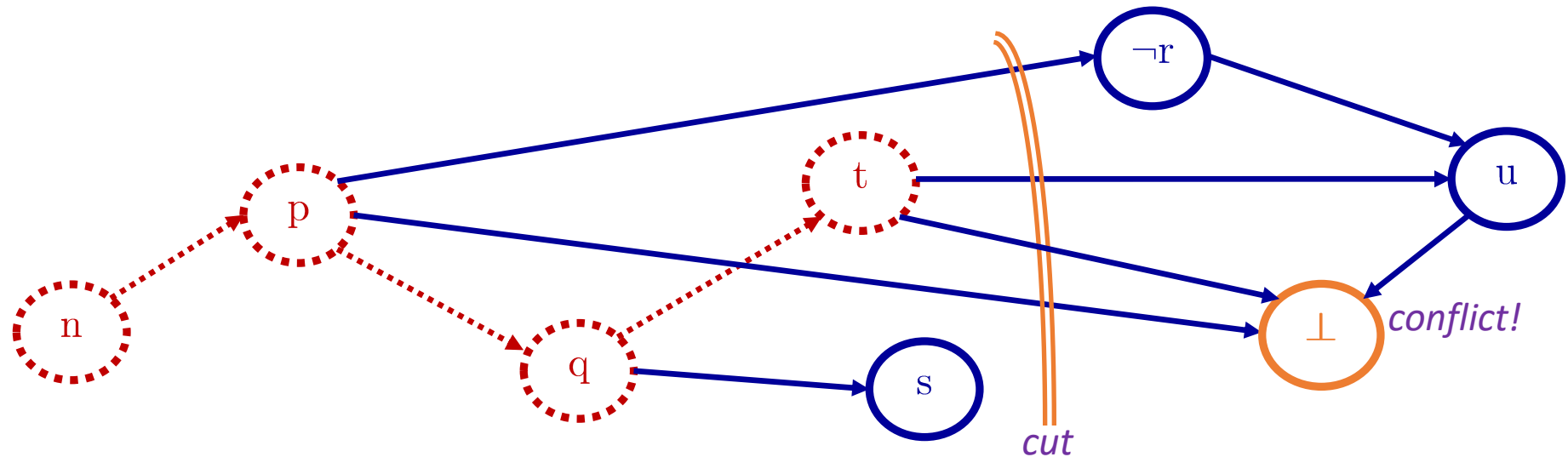
$$\checkmark (\cancel{p} \vee t \vee u) \wedge$$

$$(\cancel{p} \vee \cancel{t} \vee \cancel{u}) \wedge$$

$$\checkmark (\cancel{r} \vee \cancel{t} \vee u) \wedge$$

$$(\neg p \vee \neg t \vee \neg u)$$

Idea 4 Improved: Generalised Clause Learning



- We *generalise* the approach to clause learning:
 - draw *any boundary* (cut) separating conflict from relevant decision literals
 - It should only cross edges that (transitively) reach the conflict node (re-draw if needed!)
 - for nodes with outgoing edges crossing the boundary, take the *disjunction* of the *negations* of their literals (e.g. $\neg p \vee \neg t$): this clause *rules out* the conflict
 - when back-jumping, *conjoin this new clause* to the current formula
 - different cuts will yield different clauses and performance boosts
 - e.g. where would you “cut” to get the basic version from the previous slide?

Conflict-Driven Clause Learning (CDCL) Algorithm

- *Conflict-Driven Clause Learning (CDCL)* is the extension of DPLL with:
 - *back-jumping* and *clause learning* (Ideas 3 and 4 above)
 - complementary; in principle, each extension could be used without the other
- Learned clauses are always implied by the original clause set
 - but extra clauses can *trigger unit propagation earlier*, pruning branches
- Different ways to cut the implication graph lead to different clauses
 - e.g. “*1-UIP cut*”: just after the latest node on all paths between *last relevant decision literal* and *conflict node* (this latest node may be the decision literal)
- CDCL algorithms are used in almost all *modern SAT solvers*
 - better than DPLL in practice (less so for *random SAT* examples – why?)
 - on the other hand, *too many* clauses may slow down algorithm operations
 - In practice, clauses are also *periodically removed* during SAT solver runs

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 two different (but related) algorithms: *DPLL* and *CDCL*
- For both algorithms, several design decisions have not been specified
 - e.g. order of rule applications, selection of decision literals, cut strategy
 - these aspects don't affect final results but influence performance significantly
- All algorithms work on *CNF representations* of propositional formulas
- CDCL beats DPLL by *pruning & reordering* 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 significantly different performance in practice
 - Knowing the underlying algorithms is important for selecting a representation
- In the next slide deck: *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 (earlier SAT algorithm): *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
 - *CPSC 513: Introduction to Formal Verification and Analysis*. Alan Hu / Mark Greenstreet
- Latest SAT-solving Competition: *SAT Competition 2023*. <https://satcompetition.github.io/2023/>