

Introduction to Algorithms and Data Structures

Lecture 27: Dealing with NP-completeness (exhaustive search)

Mary Cryan

School of Informatics
University of Edinburgh

Implications of NP-complete status

When prove a problem is NP-complete, we no longer expect to be able to design polynomial-time algorithms to generate exact solutions (to the Decision problem or to an Optimization version)

What are our options?

- ▶ Heuristic methods (“rules of thumb”) that might not guarantee good results, but behave well in practice.
- ▶ Might there be a polynomial-time algorithm to search for an *approximate* solution rather than an exact one? (L25)
- ▶ Brute-force methods that run in exponential-time (today)
- ▶ Recursive backtracking (today)

Today we mostly dealing with the **search** version of the NP-complete problem - though we will resolve the **decision** question too.

We will demonstrate our methods wrt SAT.

“Brute force”

*Might not have studied it formally but probably we know what it means:
“try all possibilities”*

If our propositional formula is in CNF over n logical variables, then there are 2^n different assignments to iterate through.

We have $\Phi = C_1 \wedge \dots \wedge C_m$

Consider a specific test assignment $x = x_1 \dots x_n \in \{0, 1\}^n$

- ▶ Each clause C_j can be checked for satisfiability wrt x in time $O(|C_j|)$.
- ▶ The formula Φ can be checked for satisfiability in time $\sum_{j=1}^m O(|C_j|) = O(|\Phi|)$.

So the full “brute force” algorithm could be carried out in $O(2^n \cdot |\Phi|)$ time.

For a 3-CNF formula this would be $O(2^n \cdot m)$ time

Recursive backtracking (basic algorithm)

We set-up the exploration of the search space to exploit shared properties of the collection of the assignments $\mathbf{x} \in \{0, 1\}^n$.

Work with respect to a “current partial assignment” $\mathbf{b} = b_i \mid i \in \mathcal{I}$.

Algorithm SAT-backtrack($\Phi = C_1 \wedge \dots \wedge C_m, \mathcal{I}, \mathbf{b}$)

1. **if** ($m = 0$) **then return** T
2. **else if** Φ contains an empty clause **then return** F
3. **else** choose an unassigned variable x_i ($i \in [n] \setminus \mathcal{I}$) how?
4. $\Phi' \leftarrow \Phi(x_i \leftarrow 0)$
(simplifying Φ' based on this new assignment)
5. **if** SAT-backtrack($\Phi', \mathcal{I} \cup \{i\}, \mathbf{b} \cdot 0$)
6. **then return** T
7. **else** $\Phi' \leftarrow \Phi(x_i \leftarrow 1)$
(simplifying Φ' based on this new assignment)
8. **return** SAT-backtrack($\Phi', \mathcal{I} \cup \{i\}, \mathbf{b} \cdot 1$)

Example: recursive backtracking

We take Φ over 4 variables $\{x_1, x_2, x_3, x_4\}$ with the 10 clauses

$$\begin{aligned}\Phi = & (x_1 \vee x_2) \wedge (x_1 \vee \bar{x}_3) \wedge (\bar{x}_1 \vee \bar{x}_2 \vee x_3) \wedge (\bar{x}_2 \vee \bar{x}_3) \wedge (x_2 \vee x_3) \\ & \wedge (x_1 \vee x_2 \vee x_3 \vee x_4) \wedge (\bar{x}_2 \vee x_3 \vee x_4) \wedge (\bar{x}_1 \vee \bar{x}_4) \wedge (\bar{x}_3 \vee x_4) \wedge (x_1 \vee \bar{x}_4)\end{aligned}$$

We work through this example in the following slides.

Example: recursive backtracking

$x_1 \leftarrow 0$

($x_1 \vee x_2$)

($x_1 \vee \bar{x}_3$)

($\bar{x}_1 \vee \bar{x}_2 \vee \bar{x}_3$)

($\bar{x}_2 \vee \bar{x}_3$)

($x_2 \vee x_3$)

($x_1 \vee x_2 \vee x_3 \vee x_4$)

($\bar{x}_2 \vee \bar{x}_3 \vee x_4$)

($\bar{x}_1 \vee \bar{x}_4$)

($\bar{x}_3 \vee \bar{x}_4$)

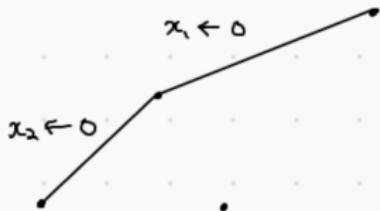
($x_1 \vee \bar{x}_4$)

Example: recursive backtracking

$x_1 \leftarrow 0$

- ~~$(x_1 \vee x_2)$~~
- ~~$(x_1 \vee \bar{x}_3)$~~
- $\checkmark (\bar{x}_1 \vee \bar{x}_2 \vee \bar{x}_3)$
- $(\bar{x}_2 \vee \bar{x}_3)$
- $(x_2 \vee x_3)$
- ~~$(x_1 \vee x_2 \vee x_3 \vee x_4)$~~
- $(\bar{x}_2 \vee x_3 \vee x_4)$
- $\checkmark (\bar{x}_1 \vee \bar{x}_4)$
- $(\bar{x}_3 \vee \bar{x}_4)$
- ~~$(x_1 \vee \bar{x}_4)$~~

Example: recursive backtracking



C_1 empty $(x_1 \vee x_2)$ \circ

~~$(x_1 \vee \bar{x}_3)$~~

✓ $(\bar{x}_1 \vee \bar{x}_2 \vee \bar{x}_3)$

✓ $(\bar{x}_2 \vee \bar{x}_3)$

~~$(\bar{x}_2 \vee x_3)$~~

~~$(x_1 \vee \bar{x}_2 \vee x_3 \vee x_4)$~~

✓ $(\bar{x}_2 \vee x_3 \vee x_4)$

✓ $(\bar{x}_1 \vee \bar{x}_4)$

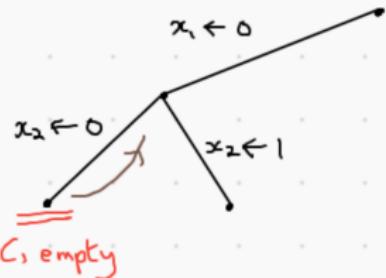
$(\bar{x}_3 \vee \bar{x}_4)$

~~$(x_1 \vee \bar{x}_4)$~~

Example: recursive backtracking

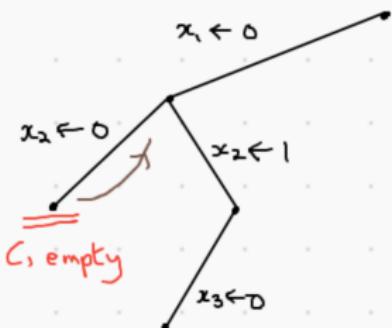


Example: recursive backtracking



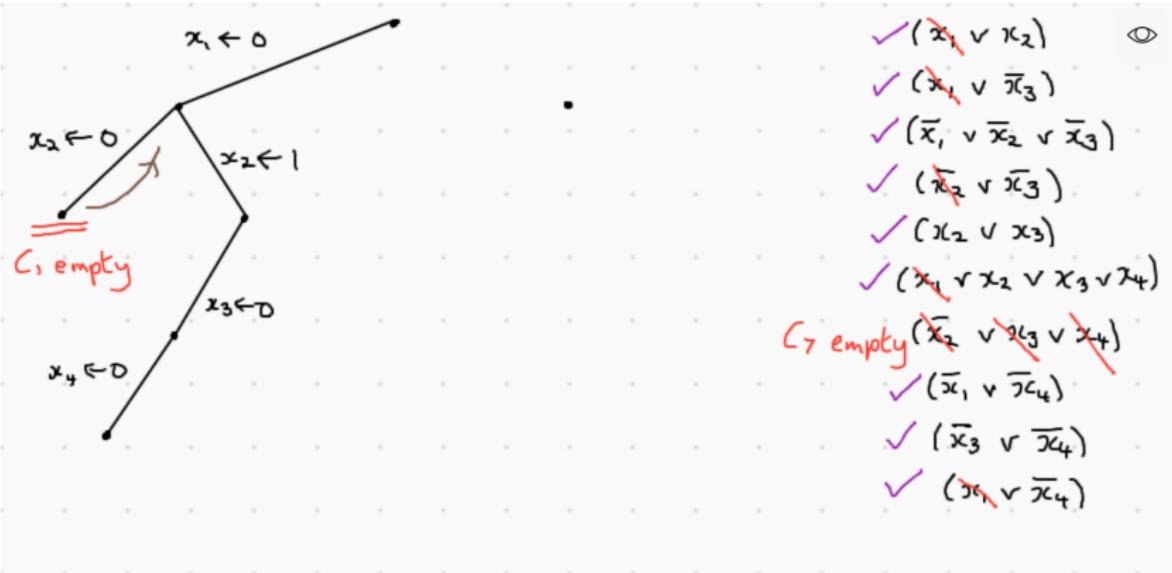
- ✓ ~~($x_1 \vee x_2$)~~ ◊
- ✓ ~~($x_1 \vee \bar{x}_3$)~~
- ✓ ~~($\bar{x}_1 \vee \bar{x}_2 \vee \bar{x}_3$)~~
- ✓ ~~($\bar{x}_2 \vee \bar{x}_3$)~~
- ✓ ~~($x_2 \vee x_3$)~~
- ✓ ~~($x_1 \vee x_2 \vee x_3 \vee x_4$)~~
- ✓ ~~($\bar{x}_2 \vee x_3 \vee x_4$)~~
- ✓ ~~($\bar{x}_1 \vee \bar{x}_4$)~~
- ✓ ~~($\bar{x}_3 \vee \bar{x}_4$)~~
- ✓ ~~($x_1 \vee \bar{x}_4$)~~

Example: recursive backtracking

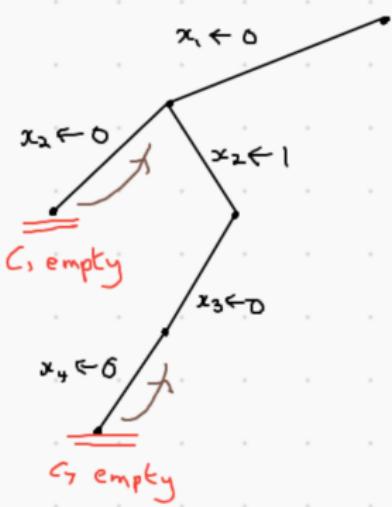


- ✓ ~~($x_1 \vee x_2$)~~
- ✓ ~~($x_1 \vee \bar{x}_3$)~~
- ✓ ~~($\bar{x}_1 \vee \bar{x}_2 \vee \bar{x}_3$)~~
- ✓ ~~($\bar{x}_2 \vee \bar{x}_3$)~~
- ✓ ~~($x_2 \vee x_3$)~~
- ✓ ~~($x_1 \vee x_2 \vee x_3 \vee x_4$)~~
- ✓ ~~($\bar{x}_2 \vee \bar{x}_3 \vee x_4$)~~
- ✓ ~~($\bar{x}_1 \vee \bar{x}_4$)~~
- ✓ ~~($\bar{x}_3 \vee \bar{x}_4$)~~
- ✓ ~~($x_4 \vee \bar{x}_4$)~~

Example: recursive backtracking

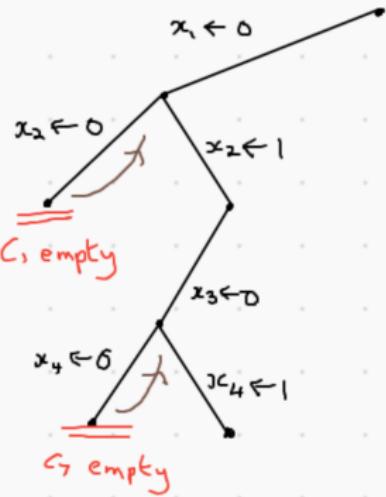


Example: recursive backtracking



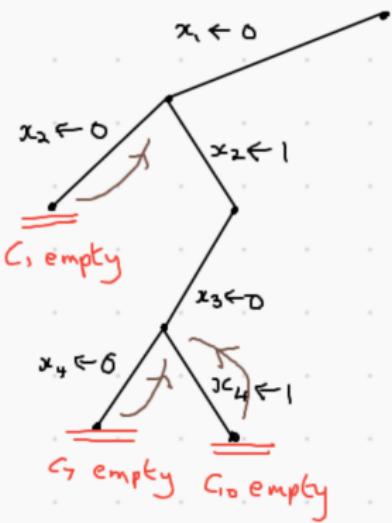
- ✓ ~~($x_1 \vee x_2$)~~
- ✓ ~~($x_1 \vee \bar{x}_3$)~~
- ✓ ~~($\bar{x}_1 \vee \bar{x}_2 \vee \bar{x}_3$)~~
- ✓ ~~($\bar{x}_2 \vee \bar{x}_3$)~~
- ✓ ~~($x_2 \vee x_3$)~~
- ✓ ~~($x_1 \vee x_2 \vee x_3 \vee x_4$)~~
- ✓ ~~($\bar{x}_2 \vee \bar{x}_3 \vee x_4$)~~
- ✓ ~~($\bar{x}_1 \vee \bar{x}_4$)~~
- ✓ ~~($\bar{x}_3 \vee \bar{x}_4$)~~
- ✓ ~~($x_1 \vee \bar{x}_4$)~~

Example: recursive backtracking



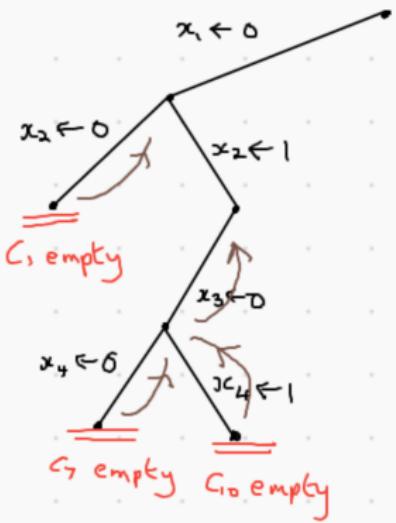
- ✓ $(x_1 \vee x_2)$ \diamond
- ✓ $(x_1 \vee \bar{x}_3)$
- ✓ $(\bar{x}_1 \vee \bar{x}_2 \vee \bar{x}_3)$
- ✓ $(\bar{x}_2 \vee \bar{x}_3)$
- ✓ $(x_2 \vee x_3)$
- ✓ $(x_1 \vee x_2 \vee x_3 \vee x_4)$
- ✓ $(\bar{x}_2 \vee \bar{x}_3 \vee x_4)$
- ✓ $(\bar{x}_1 \vee \bar{x}_4)$
- ✓ $(\bar{x}_3 \vee \bar{x}_4)$
- C₁₀ empty $(x_4 \vee \bar{x}_4)$

Example: recursive backtracking



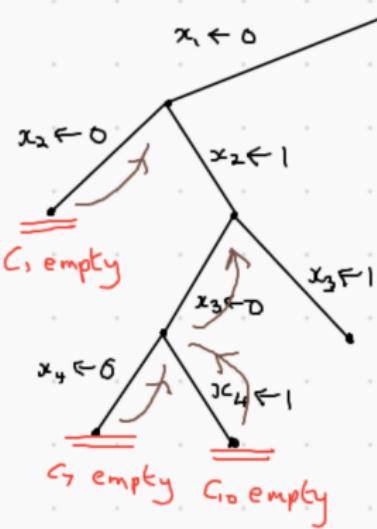
- ✓ $(x_1 \vee x_2)$ \diamond
- ✓ $(x_1 \vee \bar{x}_3)$
- ✓ $(\bar{x}_1 \vee \bar{x}_2 \vee \bar{x}_3)$
- ✓ $(\bar{x}_2 \vee \bar{x}_3)$
- ✓ $(x_2 \vee x_3)$
- ✓ $(x_1 \vee x_2 \vee x_3 \vee x_4)$
- ~~✓ $(\bar{x}_2 \vee \bar{x}_3 \vee x_4)$~~
- ✓ $(\bar{x}_1 \vee \bar{x}_4)$
- ✓ $(\bar{x}_3 \vee \bar{x}_4)$
- ~~✓ $(x_0 \vee \bar{x}_4)$~~

Example: recursive backtracking



- ✓ $(x_1 \vee x_2)$ \diamond
 ~~$(x_1 \vee \bar{x}_3)$~~
- ✓ $(\bar{x}_1 \vee \bar{x}_2 \vee \bar{x}_3)$
 ~~$(\bar{x}_2 \vee \bar{x}_3)$~~
- ✓ $(x_2 \vee x_3)$
- ✓ $(x_1 \vee x_2 \vee x_3 \vee x_4)$
 ~~$(\bar{x}_2 \vee x_3 \vee x_4)$~~
- ✓ $(\bar{x}_1 \vee \bar{x}_4)$
 ~~$(\bar{x}_3 \vee \bar{x}_4)$~~
- $(\bar{x}_1 \vee \bar{x}_4)$

Example: recursive backtracking



✓ $(x_1 \vee x_2)$

~~$(x_2 \vee \bar{x}_3)$~~

✓ $(\bar{x}_1 \vee \bar{x}_2 \vee \bar{x}_3)$

~~$(\bar{x}_2 \vee \bar{x}_3)$~~

~~$(x_2 \vee x_3)$~~

✓ $(x_1 \vee x_2 \vee x_3 \vee x_4)$

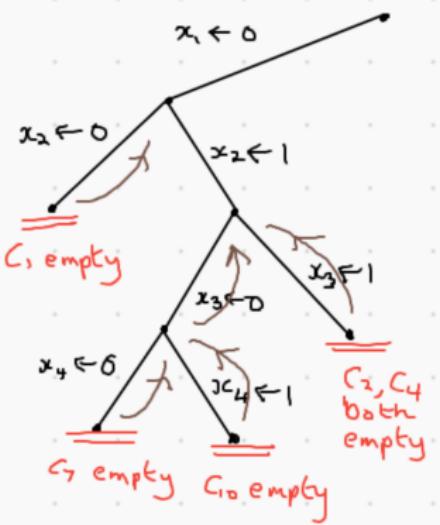
✓ $(\bar{x}_2 \vee x_3 \vee x_4)$

✓ $(\bar{x}_1 \vee \bar{x}_4)$

~~$(\bar{x}_3 \vee \bar{x}_4)$~~

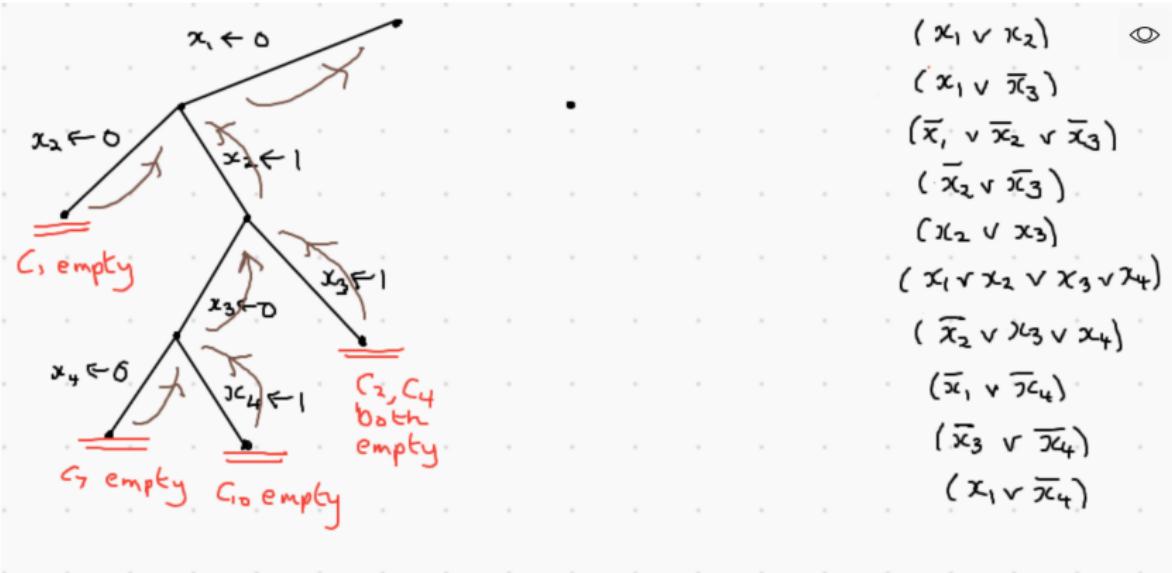
~~$(x_1 \vee \bar{x}_4)$~~

Example: recursive backtracking

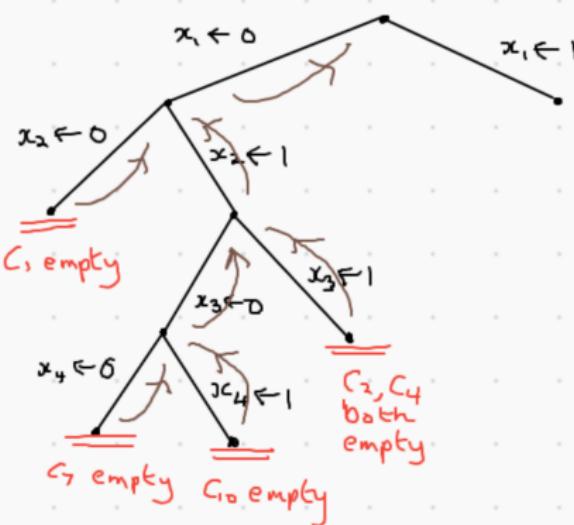


- ✓ ~~($x_1 \vee x_2$)~~
- ✓ ~~($x_1 \vee \bar{x}_3$)~~
- ✓ ~~($\bar{x}_1 \vee \bar{x}_2 \vee \bar{x}_3$)~~
- ✓ ~~($\bar{x}_2 \vee \bar{x}_3$)~~
- ✓ ~~($x_2 \vee x_3$)~~
- ✓ ~~($x_1 \vee x_2 \vee x_3 \vee x_4$)~~
- ✓ ~~($\bar{x}_2 \vee x_3 \vee x_4$)~~
- ✓ ~~($\bar{x}_1 \vee \bar{x}_4$)~~
- ✓ ~~($\bar{x}_3 \vee \bar{x}_4$)~~
- ✓ ~~($x_1 \vee \bar{x}_4$)~~

Example: recursive backtracking

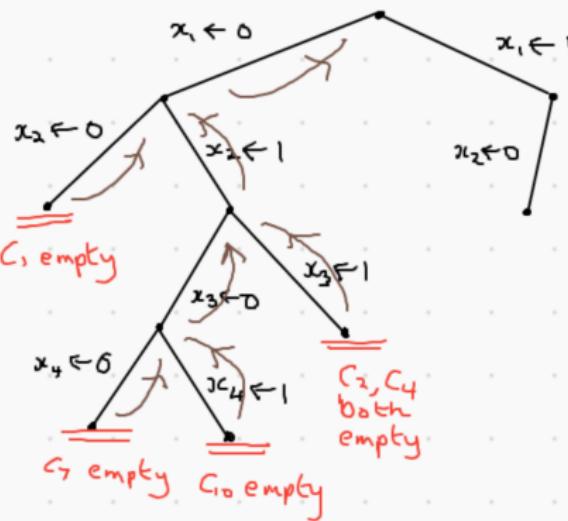


Example: recursive backtracking



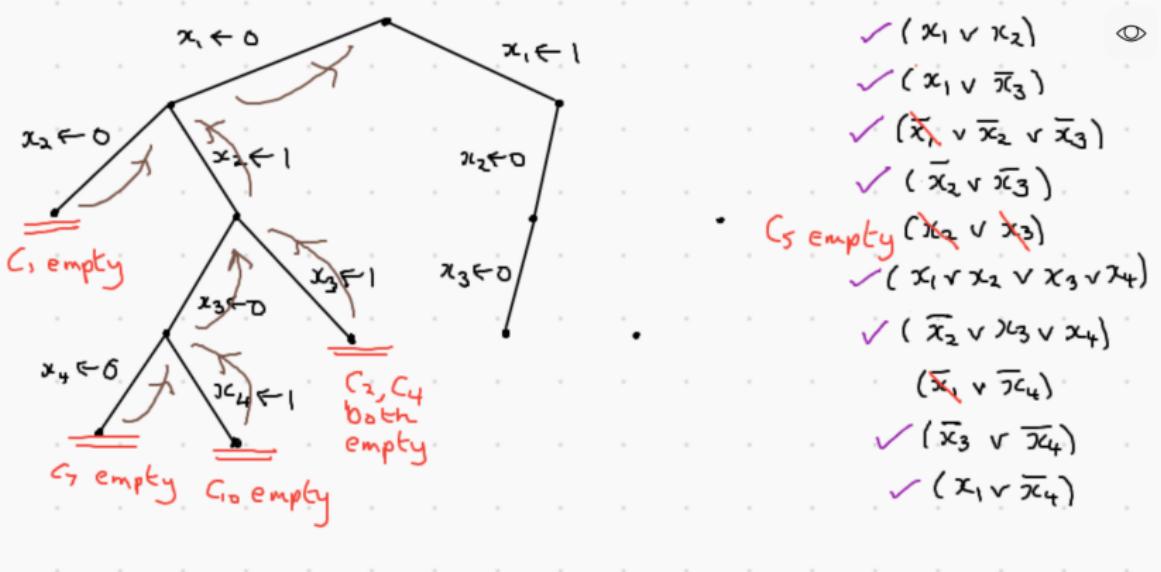
- ✓ $(x_1 \vee x_2)$ \diamond
- ✓ $(x_1 \vee \bar{x}_3)$
- ~~$(\bar{x}_1 \vee \bar{x}_2 \vee \bar{x}_3)$~~
- ~~$(\bar{x}_2 \vee \bar{x}_3)$~~
- ~~$(x_2 \vee x_3)$~~
- ✓ $(x_1 \vee x_2 \vee x_3 \vee x_4)$
- ~~$(\bar{x}_2 \vee x_3 \vee x_4)$~~
- ~~$(\bar{x}_1 \vee \bar{x}_4)$~~
- ~~$(\bar{x}_3 \vee \bar{x}_4)$~~
- ✓ $(x_1 \vee \bar{x}_4)$

Example: recursive backtracking

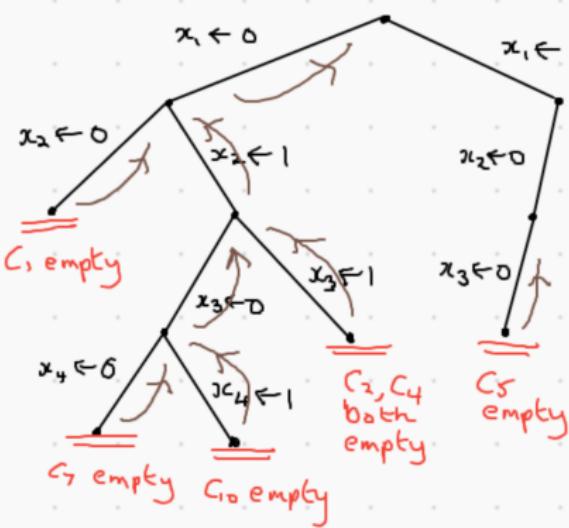


- ✓ $(x_1 \vee x_2)$ ◊
- ✓ $(x_1 \vee \bar{x}_3)$
- ✓ $(\bar{x}_1 \vee \bar{x}_2 \vee \bar{x}_3)$
- ✓ $(\bar{x}_2 \vee \bar{x}_3)$
- ~~$(\bar{x}_2 \vee x_3)$~~
- ✓ $(x_1 \vee x_2 \vee x_3 \vee x_4)$
- ✓ $(\bar{x}_2 \vee x_3 \vee x_4)$
- ~~$(\bar{x}_1 \vee \bar{x}_4)$~~
- ~~$(\bar{x}_3 \vee \bar{x}_4)$~~
- ✓ $(x_1 \vee \bar{x}_4)$

Example: recursive backtracking

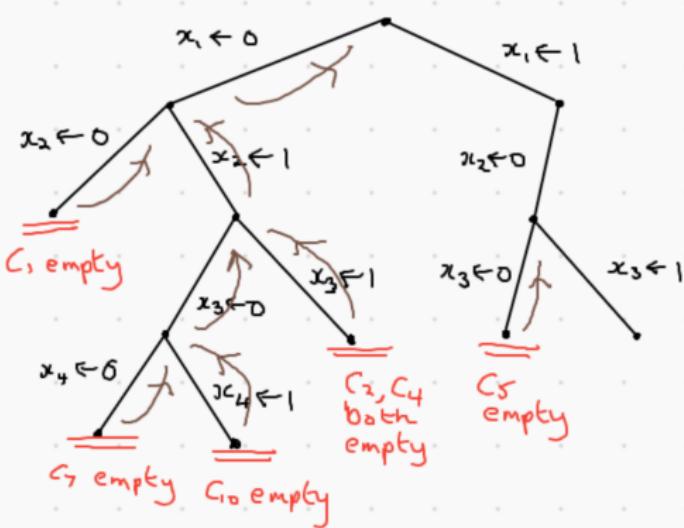


Example: recursive backtracking



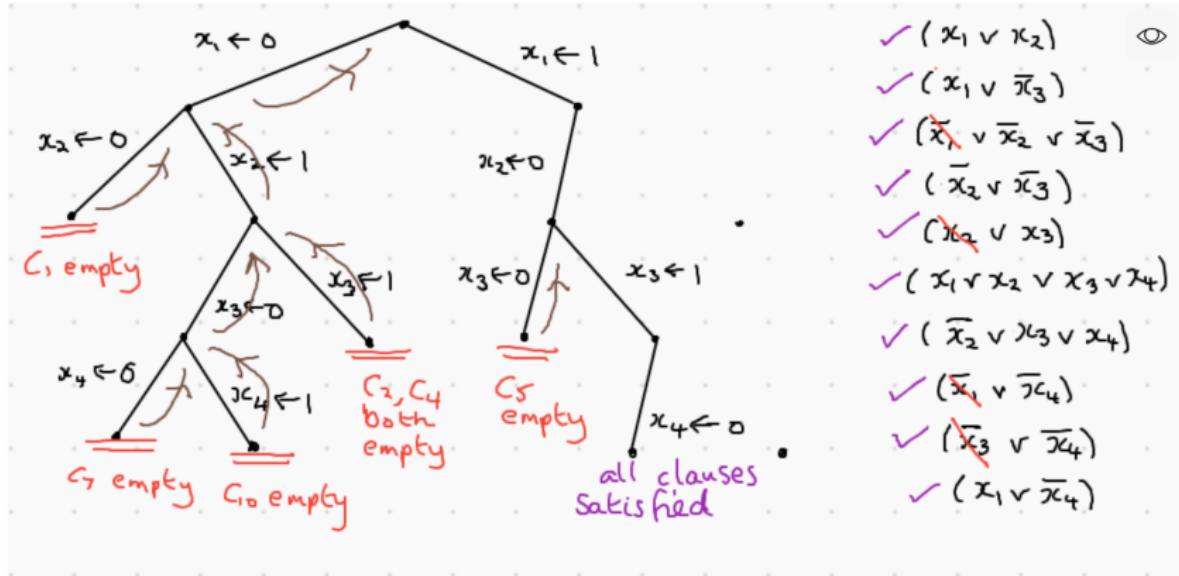
- ✓ $(x_1 \vee x_2)$ \diamond
- ✓ $(x_1 \vee \bar{x}_3)$
- ✓ $(\bar{x}_1 \vee \bar{x}_2 \vee \bar{x}_3)$
- ✓ $(\bar{x}_2 \vee \bar{x}_3)$
- ~~($x_2 \vee x_3$)~~
- ✓ $(x_1 \vee x_2 \vee x_3 \vee x_4)$
- ✓ $(\bar{x}_2 \vee x_3 \vee x_4)$
- ~~($\bar{x}_1 \vee \bar{x}_4$)~~
- ~~($\bar{x}_3 \vee \bar{x}_4$)~~
- ✓ $(x_1 \vee \bar{x}_4)$

Example: recursive backtracking



- ✓ $(x_1 \vee x_2)$ ⊗
- ✓ $(x_1 \vee \bar{x}_3)$
- ✓ $(\cancel{x}_1 \vee \bar{x}_2 \vee \bar{x}_3)$
- ✓ $(\bar{x}_2 \vee \bar{x}_3)$
- ✓ $(\cancel{x}_2 \vee x_3)$
- ✓ $(x_1 \vee x_2 \vee x_3 \vee x_4)$
- ✓ $(\bar{x}_2 \vee x_3 \vee x_4)$
- ~~($\cancel{x}_1 \vee \bar{x}_4$)~~
- ~~($\cancel{x}_3 \vee \bar{x}_4$)~~
- ✓ $(x_1 \vee \bar{x}_4)$

Example: recursive backtracking



Definitions

We work with respect to a general SAT formula $\Phi = C_1 \wedge \dots \wedge C_m$ where each of the C_j contains between 1 and n literals over the variables $\{x_1, \dots, x_n\}$.

- ▶ A **unit clause** is a clause which contains *exactly one literal* (for example, $C_j = (x_2)$ or $C_j = (\bar{x}_7)$)
- ▶ A **pure literal** is defined with respect to the whole collection of clauses: the situation when a logical variable x_i appears with only one **polarity**, either
 - ▶ *only appearing* as the positive literal x_i ...
 - ▶ or alternatively *only appearing* as the negative literal \bar{x}_i (but not with both signs)
- ▶ Both definitions apply with respect to “active” clauses/literals in a partial assignment (some variables set), as well as the original CNF.

Some observations

We can make some observations to improve recursive backtracking:

unit clause If Φ contains a unit clause (ℓ), then there is only *one possible option* for setting the underlying variable in a fully Satisfying assignment:

If ℓ is x_i , then set $x_i \leftarrow 1$, else if ℓ is \bar{x}_i , set $x_i \leftarrow 0$.

pure literal If we have some variable x_i that always appears with the same polarity (always as x_i , or alternatively always as \bar{x}_i) ... then **we may assume** we set x_i is set to its polarity in a satisfying assignment.

There might also be a satisfying assignment setting x_i in conflict with its polarity, but it cannot *hurt* to match the polarity.

Applying these rules will alter the Φ , removing some clauses (unit clause) or reducing the number of literals in other clauses.

We should iterate

Davis, Putnam, Logemann, Loveland (DPLL)

Algorithm DPLL($\Phi = C_1 \wedge \dots \wedge C_m$)

1. **if** every literal in Φ is “pure” **then return** T
2. **else if** Φ contains an empty clause **then return** F
3. **else**
4. **while** we have some “unit clause” (ℓ) in Φ
5. **if** $(\ell \text{ is } x_i)$ **then** $\Phi \leftarrow \Phi(x_i \leftarrow 1)$
6. **else if** $(\ell \text{ is } \bar{x}_i)$ **then** $\Phi \leftarrow \Phi(x_i \leftarrow 0)$
7. **while** we have some “pure literal” ℓ in Φ
8. **if** $(\ell \text{ is } x_i)$ **then** $\Phi \leftarrow \Phi(x_i \leftarrow 1)$
9. **else if** $(\ell \text{ is } \bar{x}_i)$ **then** $\Phi \leftarrow \Phi(x_i \leftarrow 0)$
10. Choose a undetermined variable x_i of Φ how?
11. **return** (DPLL($\Phi(x_i \leftarrow 0)$) **or** DPLL($\Phi(x_i \leftarrow 1)$))

$\Phi(x_i \leftarrow 0)$ eliminates the clauses that contain \bar{x}_i (now satisfied), and deletes any x_i literals inside individual clauses of Φ (clauses become harder to satisfy).
 $\Phi(x_i \leftarrow 1)$ is symmetric.

Choosing a variable to “split” on

The process of forking-off DPLL($\Phi(x_i \leftarrow 0)$) and DPLL($\Phi(x_i \leftarrow 1)$) is called **splitting** on x_i .

There are different ways used to choose the next x_i :

- ▶ Any variable still active in the “as-yet-unsatisfied” clauses (our approach).
- ▶ Variable that appears in the most clauses.
- ▶ Variable x_1 that a lot, *mostly* with one polarity (match that polarity)
- ▶ Any literal in the shortest clause.
- ▶ The variable x_i with the highest weighted sum of clause sizes
$$\sum_{k=2}^n 2^{-k} |\{C_j : x_i \in C_j, |C_j| = k\}|.$$

These are all *heuristics* - will work well on some instances of SAT, but may be poor choices for other examples.

Impact

Running time of DPLL:

- ▶ The bound of $O(2^n \cdot |\phi|)$ still applies.
- ▶ Running-time in practice is *much much better* than this.
- ▶ Proving improved upper bounds:
Can't really hope for significant improvements (SAT being NP-complete).

DPLL said to be “a collection of algorithms” (the different heuristics for splitting).

DPLL forms the basis of many practical “SAT solvers” like GRASP and Chaff.

Further study

Viewing:

The “Sudoku problem” can also be addressed via recursive backtracking.

Check out Juan Casanova’s lectures (part of this set).

Doing:

Run the DPLL backtracking algorithm on the example from these slides.

You will notice it is much quicker than the naïve backtracking algorithm (especially on the $x_1 \leftarrow 0$ branch).