If \( y \) has never been used to derive a conflict, then skip \( \bar{x} \) case.

Immediately \textit{jump back} to the \( x \) case – assuming \( x \) was used.

Split on \(-3\) first (bad decision).

Split on \(-1\) and get first conflict.

Regularly backtrack and assign \( 1 \) to get second conflict.
Backjumping helps to recover from bad decisions

- bad decisions are those that do not contribute to conflicts
- without backjumping same conflicts are generated in second branch
- with backjumping the second branch of bad decisions is just skipped

- particularly useful for unsatisfiable instances
  - in satisfiable instances good decisions will guide us to the solution

- with backjumping many bad decisions increase search space roughly quadratically instead of exponentially with the number of bad decisions
**Implication Graph**

- The implication graph maps inputs to the result of resolutions.
- Backward from the empty clause all contributing clauses can be found.
- The variables in the contributing clauses are contributing to the conflict.
- Important optimization, since we only use unit resolution:
  - Generate graph only for resolutions that result in unit clauses.
  - The assignment of a variable is the result of a decision or a unit resolution.
  - Therefore, the graph can be represented by saving the reasons for assignments with each assigned variable.

**General Implication Graph as Hyper-Graph**

(edges of directed hyper graphs may have multiple source and target nodes)

**Optimized Implication Graph for Unit Resolution in DP**

- Graph becomes an ordinary (non hyper) directed graph.
- Simplifies implementation:
  - Store a pointer to the reason clause with each assigned variable.
  - Decision variables just have a null pointer as reason.
  - Decisions are the roots of the graph.

**Learning**

- Can we learn more from a conflict?
  - Backjumping does not fully avoid the occurrence of the same conflict.
  - The same (partial) assignments may generate the same conflict.
- Generate conflict clauses and add them to CNF:
  - The literals contributing to a conflict form a partial assignment.
  - This partial assignment is just a conjunction of literals.
  - Its negation is a clause (implied by the original CNF).
  - Adding this clause avoids this partial assignment to happen again.
Conflict Driven Backtracking/Backjumping

- observation: current decision always contributes to conflict
  - otherwise BCP would have generated conflict one decision level lower
  - conflict clause has (exactly one) literal assigned on current decision level

- instead of backtracking
  - generate and add conflict clause
  - undo assignments as long conflict clause is empty or unit clause
    (in case conflict clause is the empty clause conclude unsatisfiability)
  - resulting assignment from unit clause is called conflict driven assignment

---

CNF for following Examples

```
-3 1 2 0
3 -1 0
3 -2 0
-4 -1 0
-4 -2 0
-3 4 0
3 -4 0
-3 5 6 0
3 -5 0
3 -6 0
4 5 6 0
```

We use a version of the DIMACS format.

Variables are represented as positive integers.

Integers represent literals.

Subtraction means negation.

A clause is a zero terminated list of integers.

CNF has a good cut made of variables 3 and 4 (cf Exercise 4 + 5).

(but we are going to apply DP with learning to it)

---

DP with Learning Run 1 (3 as 1st decision)

```
l = 0  (no unit clause originally, so no implications)
   -----------------------------
   decision 3
   l = 1
   3 4 5 6
   -1
   -2
   empty clause (conflict)
   l = 0
   unit
   -3
   -4
   -5
   -6
   empty clause (conflict)
```

Unit clause $-3$ is generated as learned clause and we backtrack to $l = 0$

since $-3$ has a real unit clause as reason, an empty conflict clause is learned

---

DP with Learning Run 2 Fig. 1 (-1, 3 as decision order)

```
l = 0  (no unit clause originally, so no implications)
   -----------------------------
   decision -1
   l = 1
   -1 3
   (no implications on this decision level either)
   l = 2
   (using the FIRST clause)
   decision 2
   3
   4
   empty clause (conflict)
   l = 0
   unit
   -4 -2
```

since FIRST clause was used to derive 2, conflict clause is $(1 -3)$

backtrack to $l = 1$ (smallest level for which conflict clause is a unit clause)
DP with Learning Run 2 Fig. 2 (-1, 3 as decision order)

\[
\begin{align*}
  l &= 0 & & \text{(no unit clause originally, so no implications)} \\
  l &= 1 & & \\
  & & \text{decision} \\
  & & 4 \rightarrow 3 \rightarrow -5 \rightarrow -6 \rightarrow 4 \ 5 \ 6 \\
  & & \text{1st conflict clause} \\
  & & \text{empty clause (conflict)}
\end{align*}
\]

learned conflict clause is the unit clause 1

backtrack to decision level \( l = 0 \)

DP with Learning Run 2 Fig. 3 (-1, 3 as decision order)

\[
\begin{align*}
  l &= 0 & & \\
  l &= 1 & & \text{unit} \\
  & & 4 \rightarrow 3 \rightarrow -5 \rightarrow 4 \ 5 \ 6 \\
  & & \text{2nd conflict clause} \\
  & & \text{empty clause (conflict)}
\end{align*}
\]

since the learned clause is the empty clause, conclude unsatisfiability

DP with Learning Run 3 Fig. 1 (-6, 3 as decision order)

\[
\begin{align*}
  l &= 0 & & \text{(no unit clause originally, so no implications)} \\
  l &= 1 & & \\
  & & \text{decision} \\
  & & -6 \rightarrow 3 \rightarrow -2 \rightarrow -1 \rightarrow 3 \ 4 \ 5 \ 6 \\
  & & \text{(no implications on this decision level either)} \\
  & & \text{decision} \\
  & & 3 \rightarrow 4 \rightarrow -1 \rightarrow -3 \rightarrow -2 \rightarrow -3 \ 1 \ 2 \\
  & & \text{empty clause (conflict)}
\end{align*}
\]

learn the unit clause \(-3\) and BACKJUMP to decision level \( l = 0 \)

DP with Learning Run 3 Fig. 1 (-6, 3 as decision order)

\[
\begin{align*}
  l &= 0 & & \text{(no unit clause originally, so no implications)} \\
  l &= 1 & & \\
  & & \text{decision} \\
  & & -6 \rightarrow 3 \rightarrow -5 \rightarrow 4 \ 5 \ 6 \\
  & & \text{1st conflict clause} \\
  & & \text{empty clause (conflict)}
\end{align*}
\]

finally the empty clause is derived which proves unsatisfiability
### Toplevel Loop in DP with Learning

```c
int sat (Solver solver)
{
    Clause conflict;
    for (;;)
    {
        if (bcp_queue_is_empty (solver) && !decide (solver))
            return SATISFIABLE;
        conflict = deduce (solver);
        if (conflict && !backtrack (solver, conflict))
            return UNSATISFIABLE;
    }
}
```

### Backtracking in DP with Learning

```c
int backtrack (Solver solver, Clause conflict)
{
    Clause learned_clause; Assignment assignment; int new_level;
    if (decision_level(solver) == 0)
        return 0;
    analyze (solver, conflict);
    learned_clause = add (solver);
    assignment = drive (solver, learned_clause);
    enqueue_bcp_queue (solver, assignment);
    new_level = jump (solver, learned_clause);
    undo (solver, new_level);
    return 1;
}
```

### Learning as Resolution

- **conflict clause**: obtained by backward resolving empty clause with reasons
  - start at clause which has all its literals assigned to false
  - resolve one of the false literals with its reason
  - invariant: result still has all its literals assigned to false
  - continue until user defined size is reached
- Gives a nice correspondence between resolution and learning in DP
  - allows to generate a resolution proof from a DP run
  - implemented in RELSAT solver

### Conflict Clauses as Cuts in the Implication Graph

A simple cut always exists: set of roots (decisions) contributing to the conflict.
Unique Implication Points (UIP)

Detection of UIPs

- can be found by graph traversal in the order of made assignments
- \textit{trail} respects this order
- traverse reasons of variables on trail starting with conflict
- count "open paths" (initially size of clause with only false literals)
- if all paths converged at one node, then UIP is found
- decision of current decision level is a UIP and thus a \textit{sentinel}

Further Options in Using UIPs

- assume a non decision UIP is found
- this UIP is part of a potential cut
- graph traversal may stop (everything \textit{behind} the UIP is ignored)
- negation of the UIP literal constitutes the conflict driven assignment
- may start new clause generation (UIP replaces conflict)
  - each conflict may generate multiple learned clauses
  - however, using only the first UIP encountered seems to work best

More Heuristics for Conflict Clauses Generation

- intuitively is is important to localize the search (cf cutwidth heuristics)
- cuts for learned clauses may only include UIPs of current decision level
- on lower decision levels an arbitrary cut can be chosen
- multiple alternatives
  - include all the roots contributing to the conflict
  - find minimal cut (heuristically)
  - \textbf{cut off at first literal of lower decision level} (works best)
Second Order Dynamic Decision Heuristics: VSIDS

- “second order” because it involves statistics about the search
- Variable State Independent Decaying Sum (VSIDS) decision heuristic (implemented in CHAFF and LIMMAT solver)
- VSIDS just counts the occurrences of a literal in conflict clauses
- literal with maximal count (score) is chosen
- score is multiplied by a factor $f < 1$ after a certain number of conflicts occurred (this is the “decaying” part and also called rescoring)
- emphasizes (negation of) literals contributing recently to conflicts (localization)

BERKMIN’s Dynamic Second Order Heuristics

- observation:
  - recently added conflict clauses contain all the good variables of VSIDS
  - the order of those clauses is not used in VSIDS
- basic idea:
  - simply try to satisfy recently learned clauses first
  - use VSIDS to choose the decision variable for one clause
  - if all learned clauses are satisfied use other heuristics
  - intuitively obtains another order of localization (no proofs yet)
- results are mixed, but in general slightly more robust than just VSIDS

Restarts

- for satisfiable instances the solver may get stuck in the unsatisfiable part
  - even if the search space contains a large satisfiable part
- often it is a good strategy to abandon the current search and restart
  - restart after the number of decisions reached a restart limit
- avoid to run into the same dead end
  - by randomization (either on the decision variable or its phase)
  - and/or just keep all the learned clauses
- for completeness dynamically increase restart limit

Other Types of Learning

- similar to look-ahead heuristics: polynomially bounded search
  - may be recursively applied (however, is often too expensive)
- Stålmarck’s Method
  - works on triplets (intermediate form of the Tseitin transformation):
    \[ x = (a \land b), y = (c \lor d), z = (e \oplus f) \] etc.
  - generalization of BCP to (in)equalities between variables
  - test rule splits on the two values of a variable
- Recursive Learning (Kunz & Pradhan)
  - works on circuit structure (derives implications)
  - splits on different ways to justify a certain variable value
Stålmarck’s Method

1. BCP over (in)equalities: \[ x = y \quad z = (x \oplus y) \quad z = 0 \]
   etc.

2. Structural rules: \[ x = (a \lor b) \quad y = (a \lor b) \quad x = y \]
   etc.

3. Test rule: \[ \{x = 0\} \cup E \quad \{x = 1\} \cup E \]
   \[ \downarrow \quad \downarrow \]
   \[ E_0 \cup E \quad E_1 \cup E \]
   \[ (E_0 \cap E_1) \cup E \]

Assume \( x = 0 \), BCP and derive (in)equality \( E_0 \), then assume \( x = 1 \), BCP and derive (in)equality \( E_1 \). The intersection of \( E_0 \) and \( E_1 \) contains the (in)equality valid in any case.

Stålmarck’s Method Recursively

\[ x = 0 \quad x = 1 \]
\[ \downarrow \quad \downarrow \]
\[ y = 0 \quad y = 1 \quad y = 0 \quad y = 1 \]
\[ \downarrow \quad \downarrow \quad \downarrow \quad \downarrow \]
\[ E_{00} \quad E_{01} \quad E_{10} \quad E_{11} \]
\[ \downarrow \quad \downarrow \quad \downarrow \quad \downarrow \]
\[ E_0 \quad E_1 \]
\[ \downarrow \quad \downarrow \]
\[ E \]

(we do not show the (in)equality that do not change)

Stålmarck’s Method Summary

- Recursive application
  - Depth of recursion bounded by number of variables
  - Complete procedures (determines satisfiability or unsatisfiability)
  - For a fixed (constant) recursion depth \( k \) polynomial!

- \( k \)-saturation:
  - Apply split rule recursively up to depth \( k \) on all variables
  - 0-saturation: apply all rules accept test rule (just BCP: linear)
  - 1-saturation: apply test rule (not recursively) for all variables
    (until no new (in)equality can be derived)