

Constraints in the Subproblem

When troubles spring up like mushrooms

User Habits

What if we know something about the habits of our users?

- E.g. we may know that they don't tend to spend a long time on a single page
- We could use this information to further regularize the problem

Specifically, we could add a constraint in the subproblem

I.e. by putting a limit on consecutive visits to the same node

- It seems simple enough, but in practice it's serious issue

Why is that the case?

User Habits

What if we know something about the habits of our users?

- E.g. we may know that they don't tend to spend a long time on a single page
- We could use this information to further regularize the problem

Specifically, we could add a constraint in the subproblem

I.e. by putting a limit on consecutive visits to the same node

- It seems simple enough, but in practice it's serious issue

Why is that the case?

- Such a constraint violates a basic assumption in Dijkstra's method
- I.e. that all path information can be condensed into its length

With the new constraint, our shortest path method no longer works

Walking the Line

With our shortest path approach, we were walking a fine line

- The problem could be solved in polynomial time
- ...But even a small addition could make it NP-hard instead

With the new constraint, pricing becomes indeed NP-hard

There is nothing we can do about that

- ...But perhaps we can use a better suited technique
- Something designed specifically for NP-hard, combinatorial problems
- ...With lots of messy constraints

Walking the Line

With our shortest path approach, we were walking a fine line

- The problem could be solved in polynomial time
- ...But even a small addition could make it NP-hard instead

With the new constraint, pricing becomes indeed NP-hard

There is nothing we can do about that

- ...But perhaps we can use a better suited technique
- Something designed specifically for NP-hard, combinatorial problems
- ...With lots of messy constraints

For example, we could use Constraint Programming

...In its more modern incarnation, **Lazy Clause Generation** (a.k.a. CP-SAT)

Constraint Programming and Lazy Clause Generation

Very little is lazy about that

Constraint Satisfaction Problems

CP is techniques designed to address Constraint Satisfaction Problems (CSPs):

$$\langle X, D, C \rangle$$

Where:

- X is a set of decision variables
- D is the set of their domains
- C is a set of constraints
- f is a cost function

Almost any decision problem fits those definitions...

...But in practice, a given CP solver provides

- A library of supported variables types
- A library of supported constraints

...And Constraint Optimization Problems

CP can handle Constraint Optimization Problems (COP):

$$\langle f, X, D, C \rangle$$

- Where f is a cost function

COPs are tackled as a **sequence of CSPs** via this scheme:

- best solution $x^* = \perp$
- while true find a solution for $\langle X, D, C \rangle$
 - If a solution x' is found:
 - $x^* = x'$
 - $C = C \cup \{f(x) < f(x')\}$ # We as for an improving solution
 - otherwise, break the loop

The solver state is **maintained between solutions** so as not to waste effort

Variables and Constraints

In terms of supported variables types

- All CP solver provide **integer** variables
- Some also provide **numeric**, **interval**, **set**, or **graph** variables

In terms of supported constraints

- All CP solvers provide **equalities**, **inequalities**, _ over **linear expressions**
 - $y = a^T \mathbf{x}, y \leq a^T \mathbf{x}$
- All CP solvers provide \neq constraints
 - $y \neq x$
- Most CP solvers provide **max** and **min** constraints
 - $y = \max(\mathbf{x}), y = \min(\mathbf{x})$
- Some CP solvers provide products and modulo constraints (over scalars)
 - $z = xy, y = x \mod a$

Variables and Constraints

CP solver provide also **constraints with non-mathematical nature**

- E.g. logical constraints:

$$x \vee y, x \wedge y, x \Rightarrow y \dots$$

- E.g. a set of variables should take all different values:

$$\text{ALLDIFFERENT}(x)$$

- E.g. a set of variables should take/not take values from a table T :

$$\text{ALLOWED}(x, T) \quad \text{and} \quad \text{FORBIDDEN}(x, T)$$

- E.g. a set of activities with start times x and durations d should not overlap:

$$\text{NOOVERLAP}(x, d)$$

Propagators

CP solvers are **search based**

They maintain information about the variable domains in a **Domain Store**:

- The solver may store the domain bounds, i.e. $x_i \in \{lb_i, \dots, ub_i\}$
- ...Or the individual allowed values, i.e. $x_i \in \{v_0, v_1, \dots\}$
- Other representations are also possible

Constraints are associated to algorithms called **propagators**

- A propagator takes as input the current variable domains
- ...And can **prune** (some) provably infeasible values

By doing so, we can dramatically reduce the size of the search space

Propagators often rely on **structural patterns** to improve pruning

- E.g. ALLDIFFERENT(x) can prune more than $x_i \neq x_j, \forall i \neq j$
- ...Since it can reason on multiple variables at the same time

Propagators

Let's see an example for $\text{ALLOWED}([x_0, x_1], T)$, with T given by:

x_0	x_1
0	0
0	1
1	1

Let's that initially $D_0 = \{0, 1\}$, and $D_1 = \{0, 1\}$

- If x_0 loses the value **0**
 - ...Then the **ALLOWED** propagator prunes **0** from D_1
 - ...Because it no longer has a feasible **support** in D_0
- If x_1 loses the value **1**
 - ...Then the **ALLOWED** propagator prunes **1** from D_0
 - ...Because it no longer has a feasible **support** in D_1

Propagators and Lazy Clause Generation

In Lazy Clause Generation solvers, propagators have two additional tasks:

1) Whenever they prune a domain, they also **generate boolean literals**

- These correspond to the pruning operations
 - E.g. in our two example we would generate literals $[x_0 \neq 0]$ and $[x_1 \neq 1]$
- These literals represent **variables** associated to the state of a constraint
 - E.g. $[x_0 \neq 0] = 1$ if $0 \notin D_0$ and $[x_0 \neq 0] = 0$ otherwise

2) Whenever they prune a domain, they also generate an **explanation**

- This is a logical **clause** representing the reasoning that led to pruning
 - E.g. in our first example we would generate $[x_0 \neq 0] \Rightarrow [x_1 \neq 0]$
- These clauses are constraints on the literal variables
 - They function like normal constraints (except they are specifically tracked)

Constraint Propagation

Pruning can trigger the activation of other propagators

...In a process called **Constraint Propagation**

- After some activations, the process reaches a **fix point**
- If propagation causes a domain to become empty, we have a **conflict**
- ...In which case we need to backtrack

As a consequence, propagators are called many times per search node

We can have millions of propagator calls in a solution process

- For this reason, they often run in **constant or low-degree polynomial** time
- ...And propagator are heavily optimized
 - E.g. the **ALLOWED** constraint is so efficient
 - ...That tables with $> 100,000$ entry can be handled almost instantly
 - This is achieved by relying on incremental computation

Constraint Propagation and Implication Graphs

In LCG solvers, constraint propagation generates an **implication graph**

This consists of the **literals**, connected by the generated **explanations**

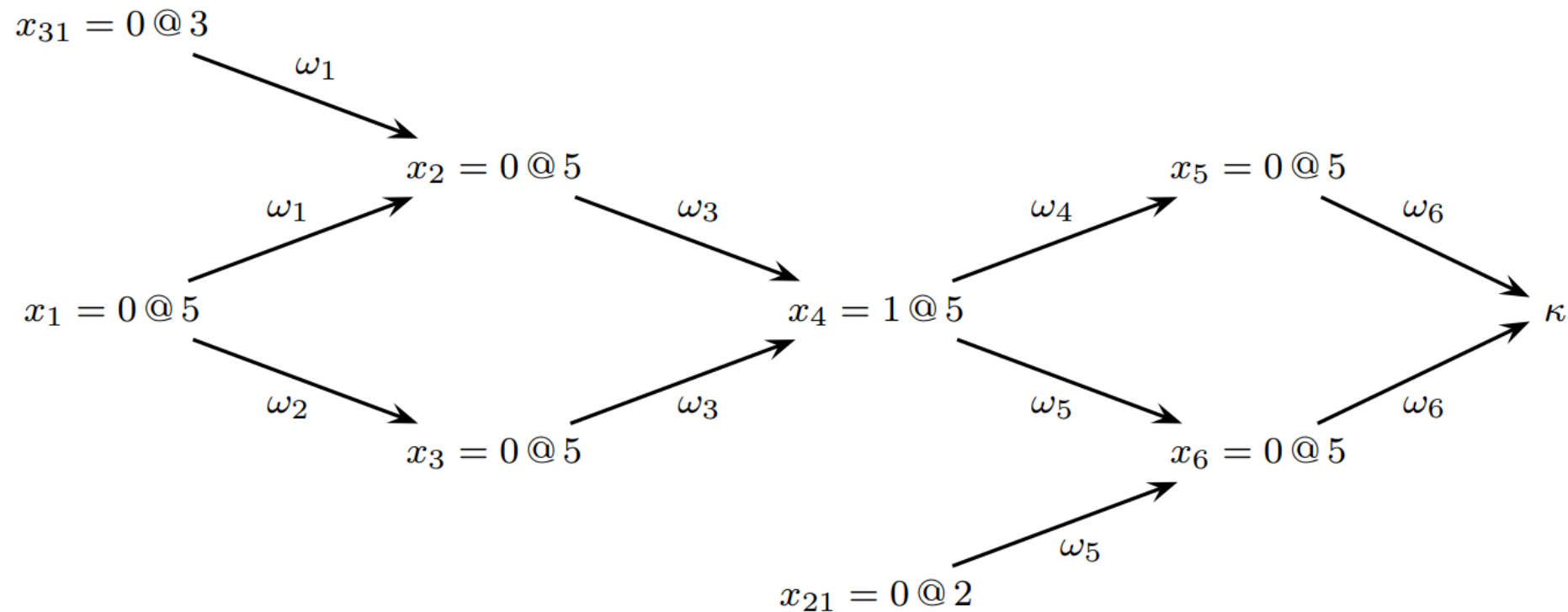


Image from [this book chapter](#)

- In the pictures, **x** variables correspond to boolean literals (i.e. constraint states)
- ...And each **ω** is an explanation

Conflict Driven Clause Learning

In LCG solvers, each search decision also generates a literal

E.g. if we assign 1 to x_0 , we generate $[x_0 = 1]$

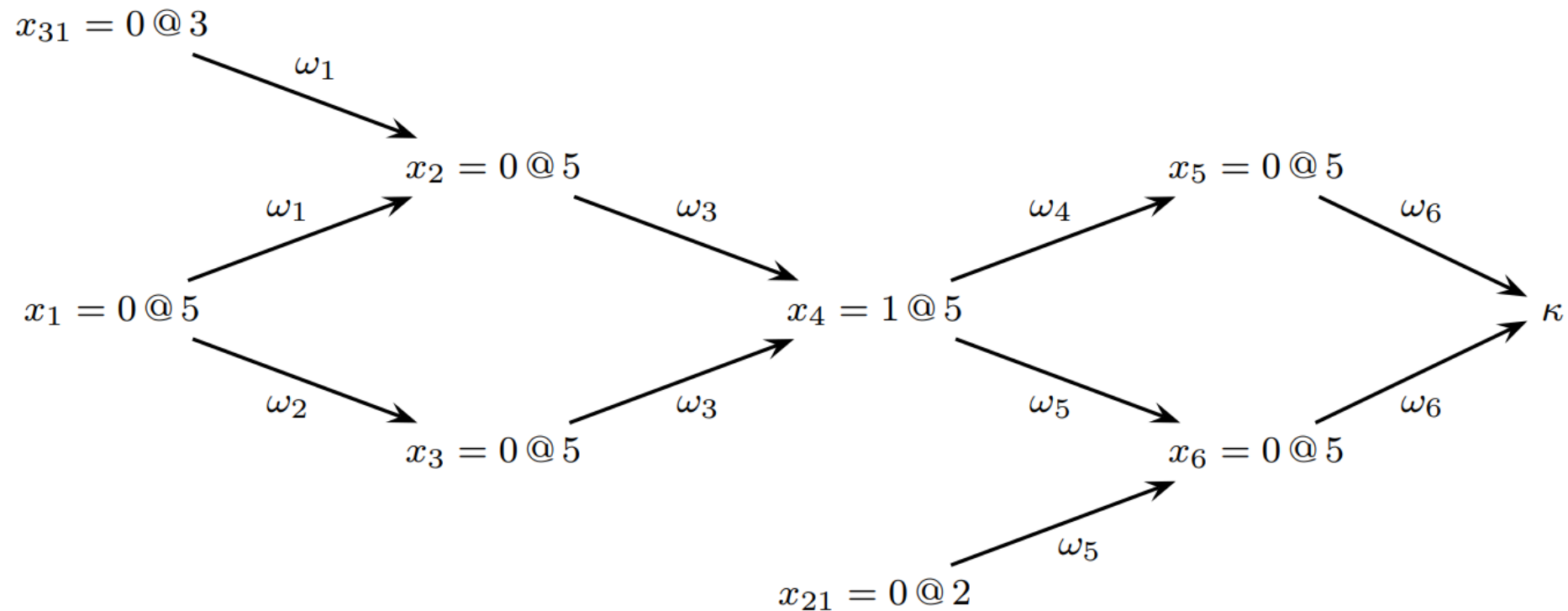


Image from [this book chapter](#)

Each decisions is associated to a **decision value**

- In the picture, they are the number after the @ symbol
- When we make a new decision, we increment the current decision value

Conflict Driven Clause Learning

In LCG solvers, each search decision also generates a literal

E.g. if we assign 1 to x_0 , we generate $[x_0 = 1]$

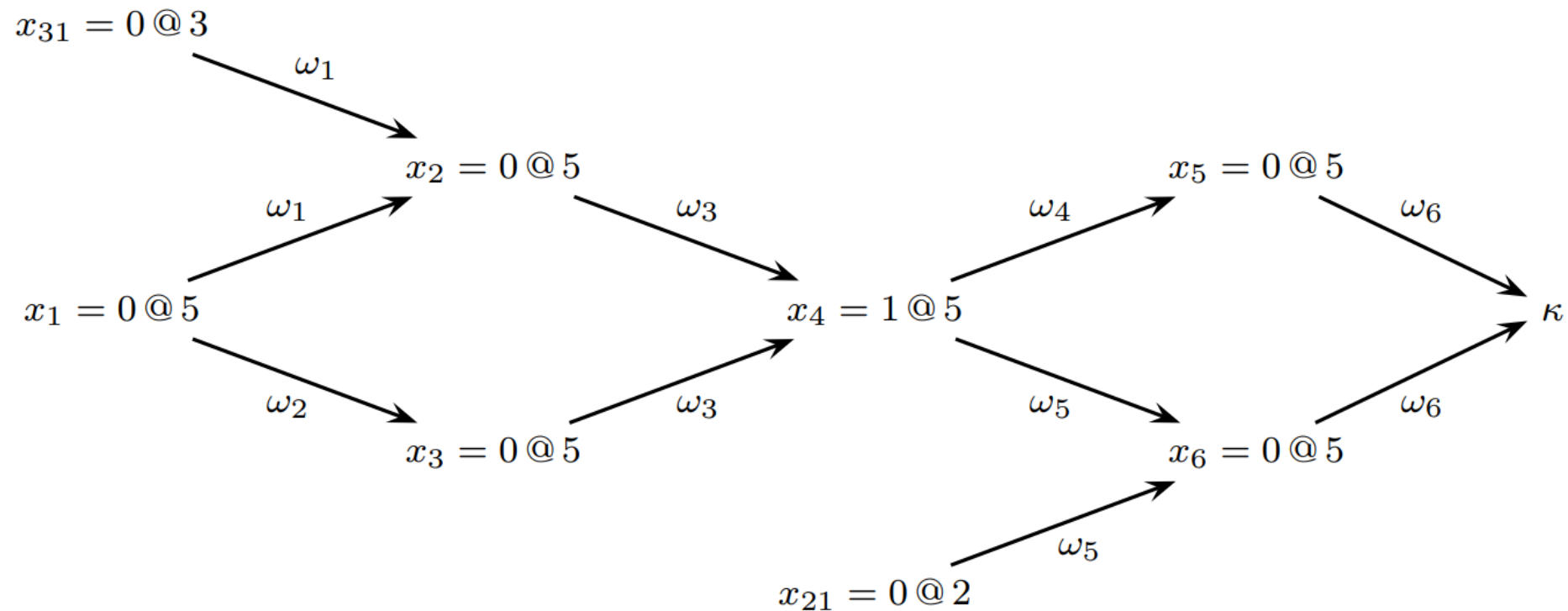


Image from [this book chapter](#)

Literals generated by propagation are also labeled with a decision value

- ...But in this case there is **no increment**
- In the picture, many literals are associated to decision level 5

Conflict Driven Clause Learning

In case of a conflict (κ in the figure), an LCG solver can **learn a constraint**

This technique is referred to as **Conflict Driven Clause Learning**

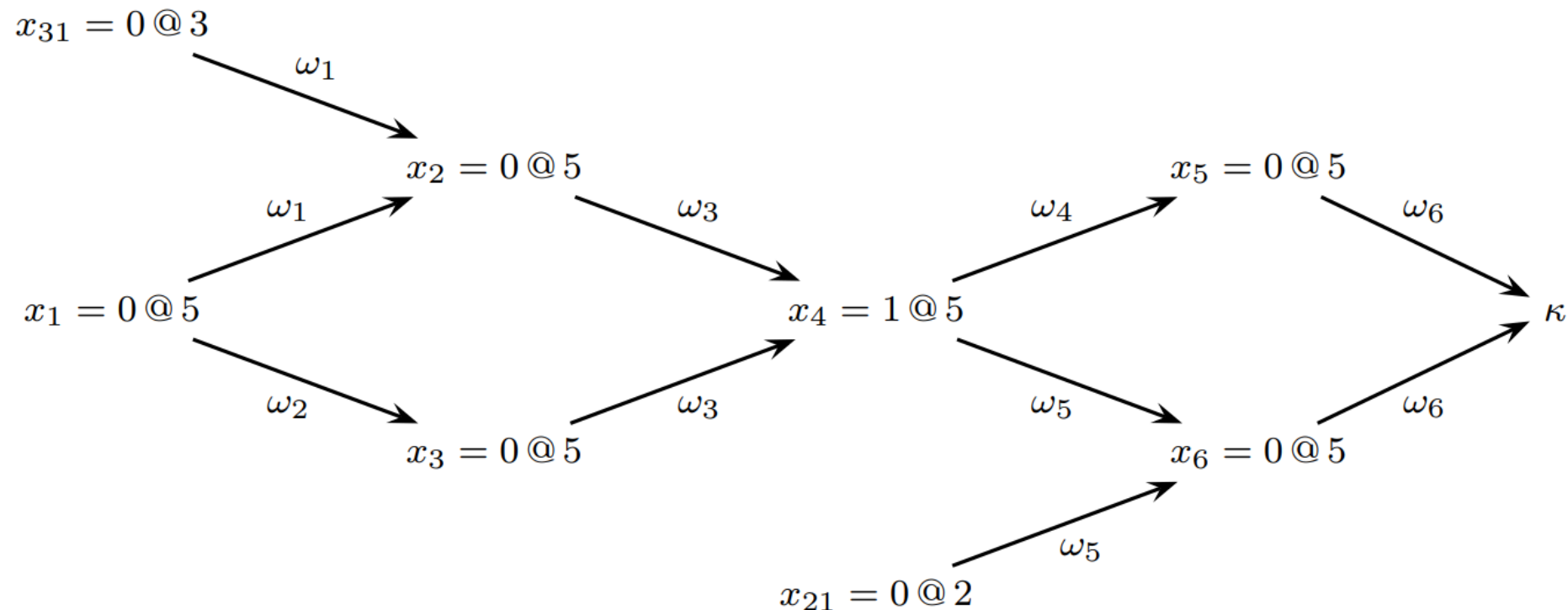


Image from [this book chapter](#)

The idea is to identify **which decisions (literals) are to blame** for the conflict

- First, we identify all literal with the **same decision value** as the conflict
- The earliest one ($x_1 = 0@5$ in the figure) always corresponds to a decision

Conflict Driven Clause Learning

In case of a conflict (κ in the figure), an LCG solver can **learn a constraint**

This technique is referred to as **Conflict Driven Clause Learning**

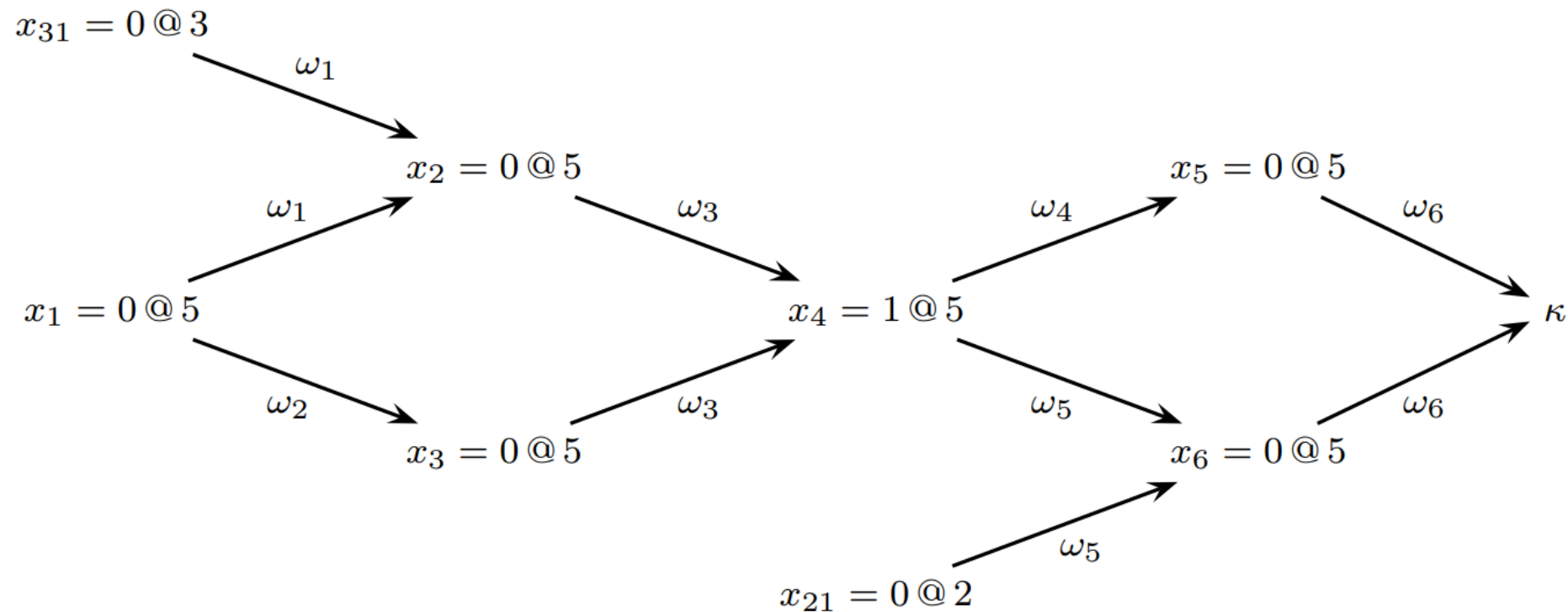


Image from [this book chapter](#)

Without this literal, the conflict would not arise

- ...Therefore it will appear in the learned constraint

Conflict Driven Clause Learning

In case of a conflict (κ in the figure), an LCG solver can **learn a constraint**

This technique is referred to as **Conflict Driven Clause Learning**

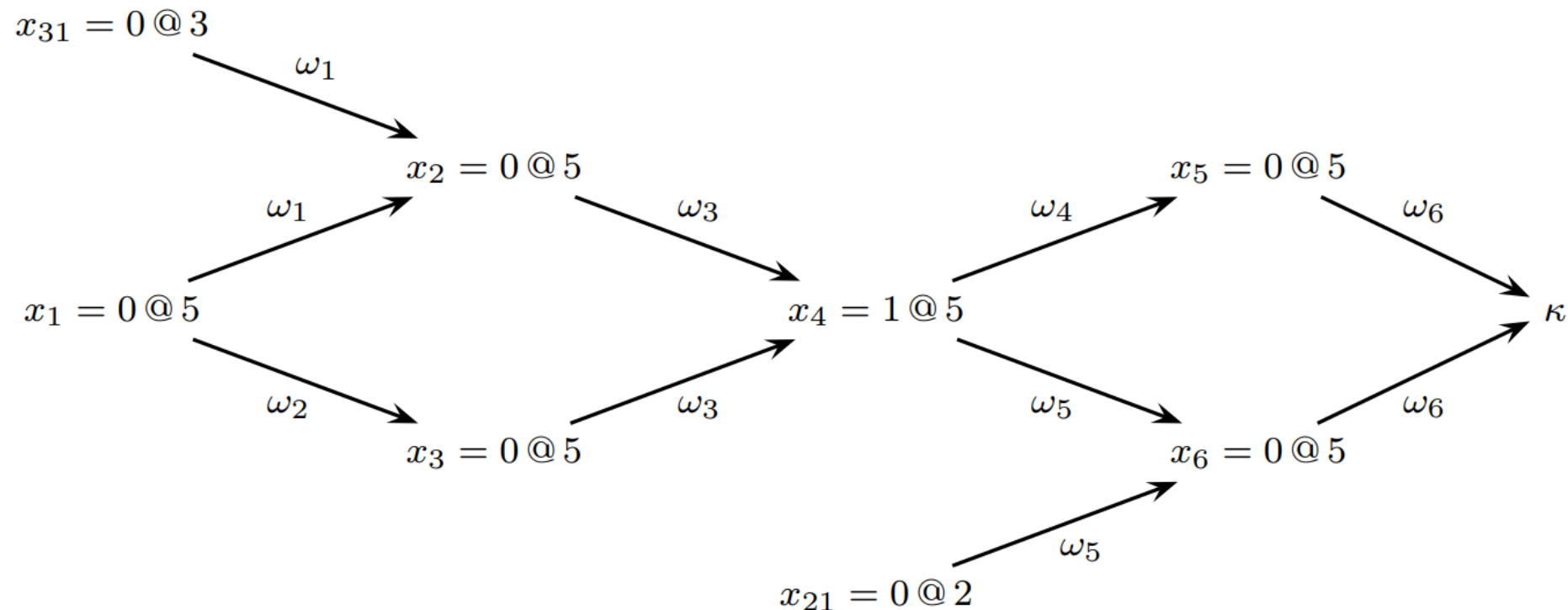


Image from [this book chapter](#)

To this, we add all literals with decision level **lower** than the current one

- ...That are connected via explanation to literals in the current decision level
- In the figure, those would be $x_{31} = 0 @ 3$ and $x_{21} = 0 @ 2$

Conflict Driven Clause Learning

In case of a conflict (κ in the figure), an LCG solver can **learn a constraint**

This technique is referred to as **Conflict Driven Clause Learning**

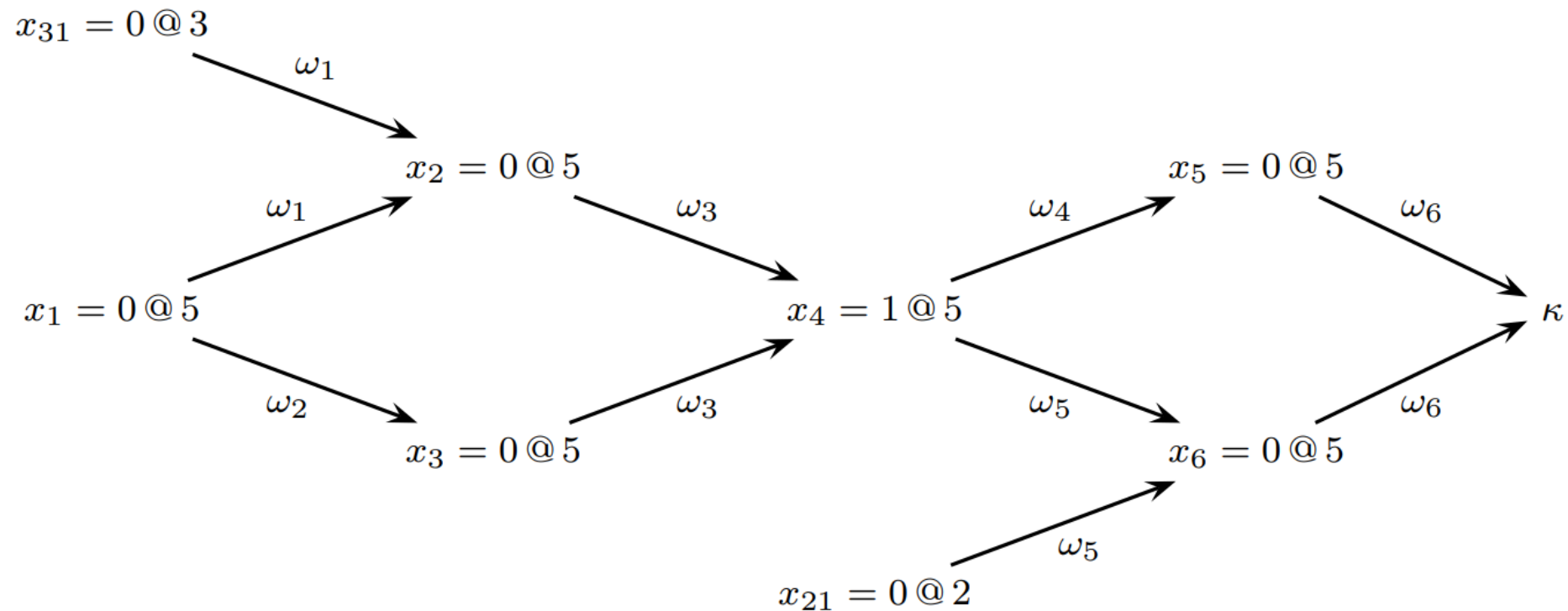


Image from [this book chapter](#)

If we want to avoid the conflict, **at least one** of these literals should be **false**

Therefore the clause we learn is: $\neg[x_1 = 0] \vee \neg[x_{31} = 0] \vee \neg[x_{21} = 0]$

Conflict Driven Clause Learning

The clause we learn is **globally valid**

...So that we can restart search and we will not make the same mistake again

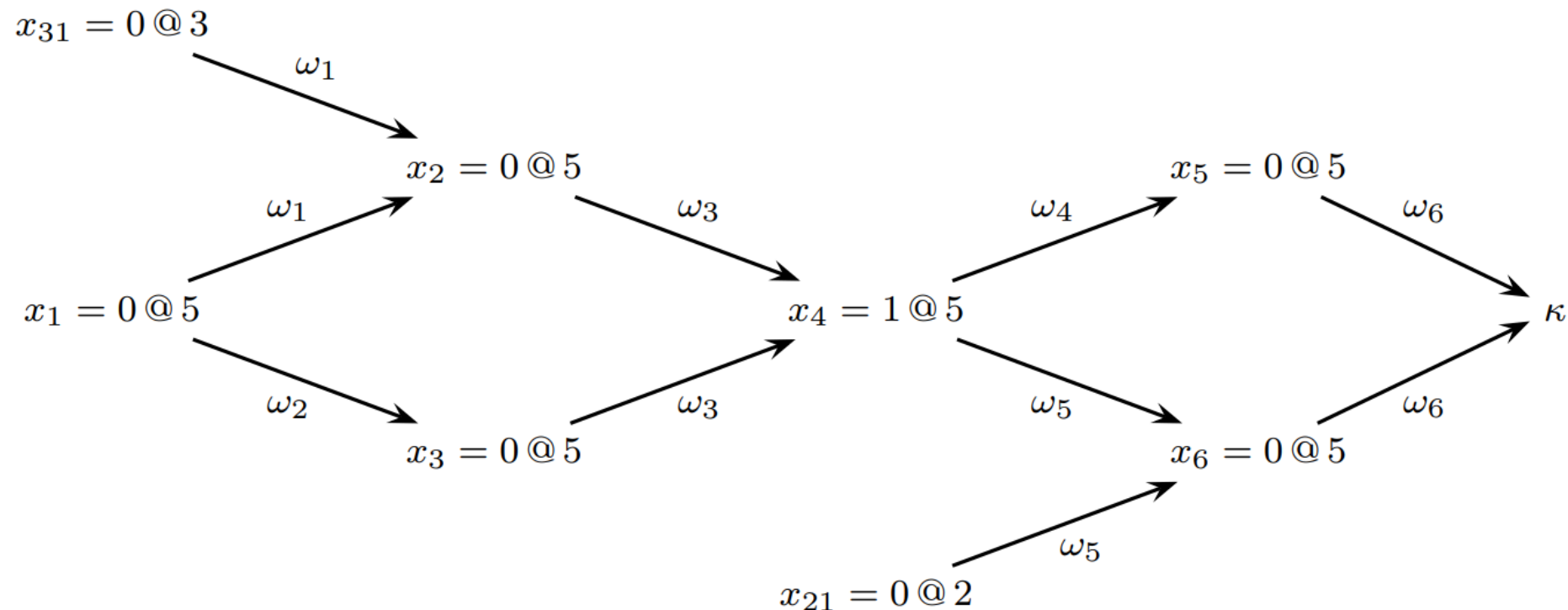


Image from [this book chapter](#)

In other words, we have a **complete** method that does **not** rely on tree search

- CDCL was invented for pure SAT solvers, and it is key to their efficiency
- In LCG we used it to obtain similarly strong benefits

Some Considerations

As usual, we have just scratched the surface for CP/LCG

- You can find more information about classical CP [in this handbook](#)
- ...And for LCG the best starting point are the papers by [Peter Stuckey](#).

Unlike MILP, CP does not rely on numerical optimization

- Combinatorial constraints are first-class citizens
 - E.g. we have no big-Ms here!
- It tends to work best for problems with many combinatorial elements

Unlike MILP, CP lack a global bounding method

- There is no LP relaxation, and propagation works at a **local** level
- CDCL goes a long way towards countering this issue
- ...But sometimes the lack of a global bound leads to weaker performance