CONSTRAINT PROGRAMMING

Introduction

- Disadvantages of SAT solvers:
  - The range of problems that can be solved is limited
    - integer variables can not be represented easily and efficiently
    - not every constraint can easily and efficiently be rewritten in CNF:
      - numerical constraints $x_1 + x_2 + \cdots + x_n \geq 4$
      - graph constraints
        ("from node *x* node *y* can be reached", "the shortest path from node *x* to node *y* may not be longer than *a*")
    - dealing with optimization problems is hard
  - The specification language is not very simple to use

Constraint Programming

- **Constraint programming:** a programming paradigm in which a problem is specified declaratively in terms of high-level constraints, and solvers find solutions

  “Constraint programming =
  Model (by user)
  +
  Search (by solver)”

Non-boolean Variables & High-level Constraints

- variables

  $E_{11} \ldots E_{99}$

- variables have domains

  $E_{xy} = \{1 \ldots 9\}$

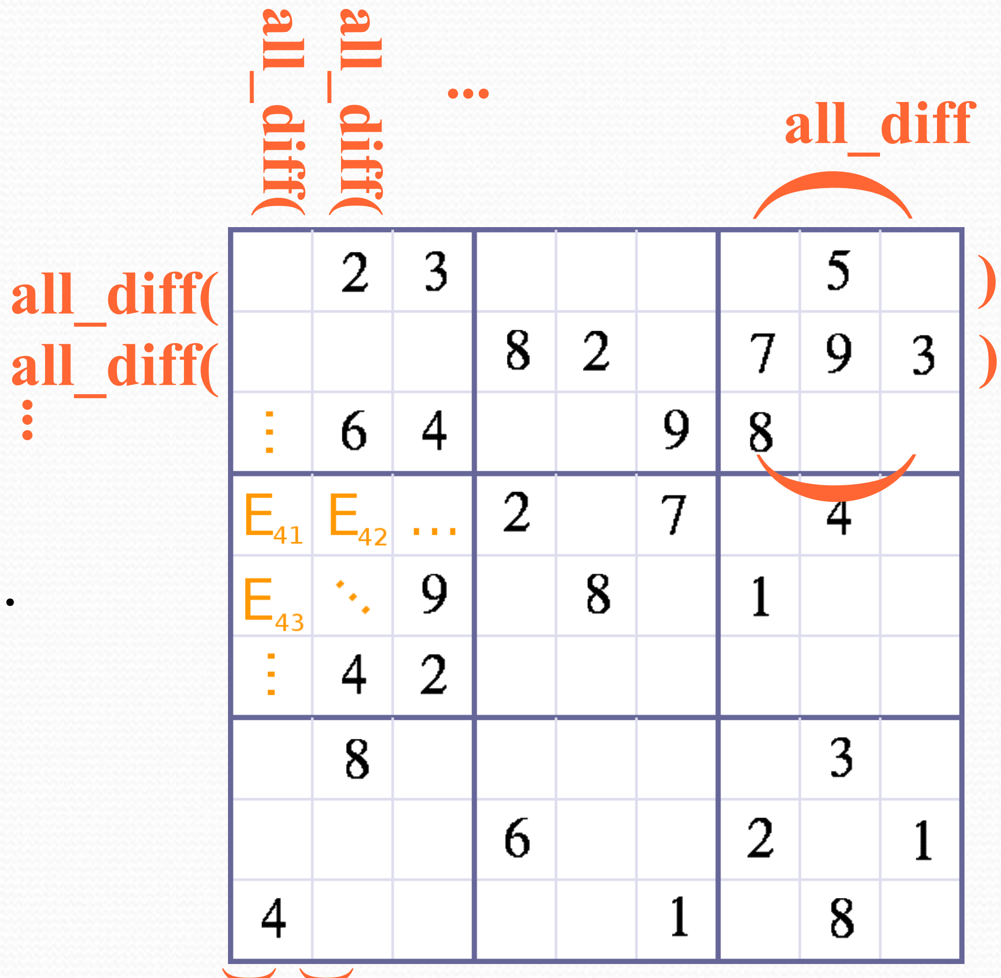
- Constraints

  all_different($[E_{1x}]$), …

  all_different($[E_{x1}]$),

  all_different($[E_{11} \ldots E_{33}]$), …

High-level all difference constraint



Solving

- Two approaches:
  - automatically translate high-level constraints into a low-level representation (like a CNF formula)
    - MiniZinc (specialized language) + G12 (solvers)
    - NumberJack (Python library)
  - run a solver which directly supports high-level constraints

Domains must be finite

Common in constraint programming are finite domain solvers based on exhaustive search & propagation

Propagation

- Each (high-level) constraint is implemented in a **propagator**, which **only** operates on the variables listed in the constraint

- For each variable we store the **domain** of values the variable can still take, which may be
  - the complete domain (i.e., all values – clearly only works for problems with finite domains)

$D(x) = \{ 2 \}, D(y) = \{ 2, 3 \}$

  - lower and upper bounds, i.e. the minimum and maximal value the variable can still take

Propagation

- The task of the propagator is to maintain **domain consistency**, i.e. to **shrink** the domains of variables to values that they can still take

if domain $D(x) = \{ 2 \}$, $D(y) = \{ 2, 3 \}$ and constraint $x \neq y$ apply, then we can deduce that $D(y) = \{3\}$.

Bounds

if domain $D(x) = \{ 1, \ldots, 5 \}$, $D(y) = \{ 1, 2 \}$ and constraint $x + y < 5$ apply then we deduce that $D(x) = \{ 1, \ldots, 3 \}$

CP Search

no domain to change any more

```
Search ( Variables ):
    propagate all constraints till fix point
    if contradiction found then return
    if at least one variable is not fixed yet then
        pick one variable V not fixed
        for each possible value of V do
            let V=value in this iteration
            Search ( Variables )
        od
    else
        print solution in Variables
```

CP Search

all rows: all_different(row)
all columns: all_different(col)
all squares: all_different(square)

CP: Branch & Propagate

- propagate 2 (row)
- branch 4
- propagate 6 (square)

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

Propagation

- Propagators may implement special algorithms and data structures

<u>all-different constraint:</u>

all variables in a list must have a different value

<u>algorithm 1:</u> use inequality constraints independently

$D(x_1) = \{ 1, 2 \}$

$D(x_2) = \{ 1, 3 \}$

$D(x_3) = \{ 1, 3 \}$

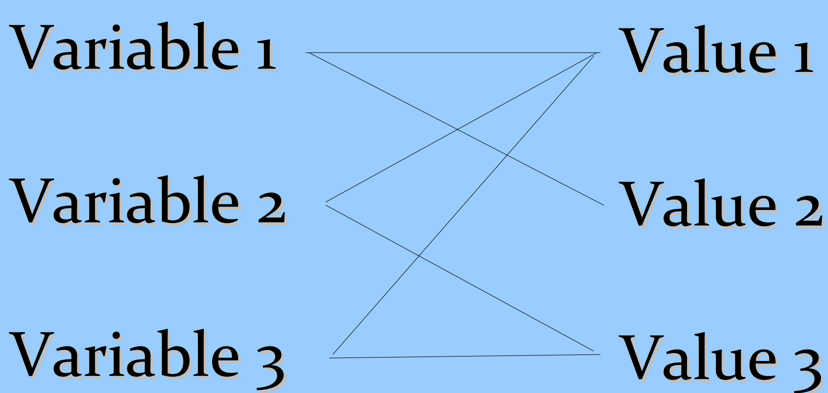
$x_1 \neq x_2, x_1 \neq x_3, x_2 \neq x_3$

Propagation for inequality:
if one variable is fixed,
remove the corresponding
value from the domain
of the other variable
→ nothing happens in example

Propagation

- Propagators may implement special algorithms and data structures

<u>all-different constraint:</u>
all variables in a list must have a different value
<u>algorithm 2:</u> graph-based; bipartite matching



Variable 1 is fixed to value 2

Comparison to SAT solvers

- CP solvers support larger numbers of constraints & optimization
- When applied to CNF formulas, they search less efficiently as:
  - there is no clause learning
  - there is no propagation for pure symbols

These weaknesses led to the development of SMT SAT solvers (SAT-Modulo-Theories), which combine ideas of constraint programming and SAT solvers

Robert Nieuwenhuis, 2006.

Implementation issues

- When to run a propagator?
  - when a variable changes? (In any way)
  - when one particular bound changes?

for domain $D(x) = \{ 1, \ldots, 3 \}$, $D(y) = \{ 1, 2, 3 \}$ and constraint $x + y < 5$; should we propagate when we remove value 1 from $D(y)$? When we remove value 3?

In the CP literature, many different such strategies have been explored, called AC1, AC2, <u>AC3</u>, … AC5

Implementation issues

- Should we store simplified constraints during the search?

$D(x)=\{1,2,3\}$, $D(y) = \{ 4 \}$, $D(z) = \{ 1, 2\}$,
$x + y + z < 10 \rightarrow x + z < 6$

- Which order to select variables?
- Which order to select values?

Implementation issues

- How to branch over variables?

D(x)={1,…,10}, D(z) = { 1,…,10}, x + y < 20

Branch with *D(x)={c} for all c in 1…10?*

Branch with *D(x)={1…,5}* and *D(x)={6,..10}*?

INTEGER LINEAR PROGRAMMING

Linear programming

- One special type of constraint is the linear constraint:

$$a_1 x_1 + a_2 x_2 + \cdots + a_n x_n \leq b$$

Constant **Real valued variable**

- A *linear program* is a constraint optimization problem on real-valued variables with a linear optimization criterion and linear constraints, and **no** other constraints:

maximize $c_1 x_1 + c_2 x_2 + \cdots + c_n x_n$

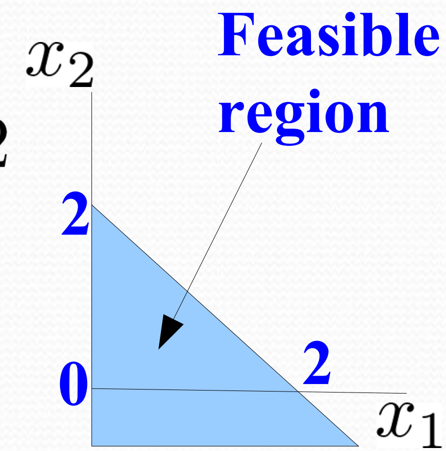
where

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n &\leq b_1 \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n &\leq b_2 \\ &\vdots \\ a_{m1}x_1 + a_{m2}x_2 + \cdots + a_{mn}x_n &\leq b_m \end{aligned}$$

Linear Programming Examples

$$\begin{array}{ll} \textbf{maximize} & x_1 + 2x_2 \\ \textbf{where} & x_1 + x_2 \le 2 \\ & -x_1 \le 0 \end{array}$$

$$x_1 = 0, x_2 = 2$$



$$\begin{array}{ll} \textbf{maximize} & -2x_1 + 3x_2 \\ \textbf{where} & x_1 - x_2 \le 2 \\ & -x_1 \le 0 \\ & x_2 \le 3.5 \end{array}$$

$$x_1 = 0, x_2 = 3.5$$

Integer Linear Programming

- **Integer** linear programming differs from linear programming in that we constrain some variables to integer values; if some variables are not integer, this is also referred to as **mixed** integer linear programming

$$\begin{array}{ll} \textbf{maximize} & -2x_1 + 3x_2 \\ \textbf{where} & x_1 - x_2 \leq 2 \\ & -x_1 \leq 0 \\ & x_2 \leq 3.5 \\ & x_1, x_2 \in \mathbb{Z} \end{array}$$

$$x_1 = 0, x_2 = 3$$

Solvers Linear Programming

- **Linear programs** can be solved in polynomial time by using *interior point* algorithms, which "walk through the interior of the feasible region"

- In practice, **linear programs** are often solved using *simplex* algorithms, which "walk over the outer rim of the feasible region" (the edges of the convex polytope)

Solvers for Integer Linear Programming

- There are no polynomial solvers for **integer linear programming**
- Most solvers are based on *cut-and-branch*
  - solve the program without integer constraints
  - if solution is not integer, try to add a “clever” linear constraint that “cuts” the non-integer solution from the feasible solution space, without changing the feasible integer solutions
  - branch if no such linear constraint can be found, for the two closest integer values for one of the variables that does not have an integer value

Graph Coloring using ILP

Example: we could use ILP to solve graph coloring with *k* colors

(left: constraint in SAT form) **(right: constaint in ILP form)**

- for each node *i*, create a formula

$$\phi_i = p_{i1} \vee p_{i2} \vee \cdots \vee p_{ik} \qquad x_{i1} + x_{i2} + \cdots + x_{ik} \geq 1$$

indicating that each node *i* must have a color

- for each node *i* and different pair of colors c_1 and c_2, create a formula

$$\phi_{ic_1c_2} = \neg p_{ic_1} \vee \neg p_{ic_2} \qquad (1 - x_{ic_1}) + (1 - x_{ic_2}) \geq 1$$

indicating a node may not have more than 1 color

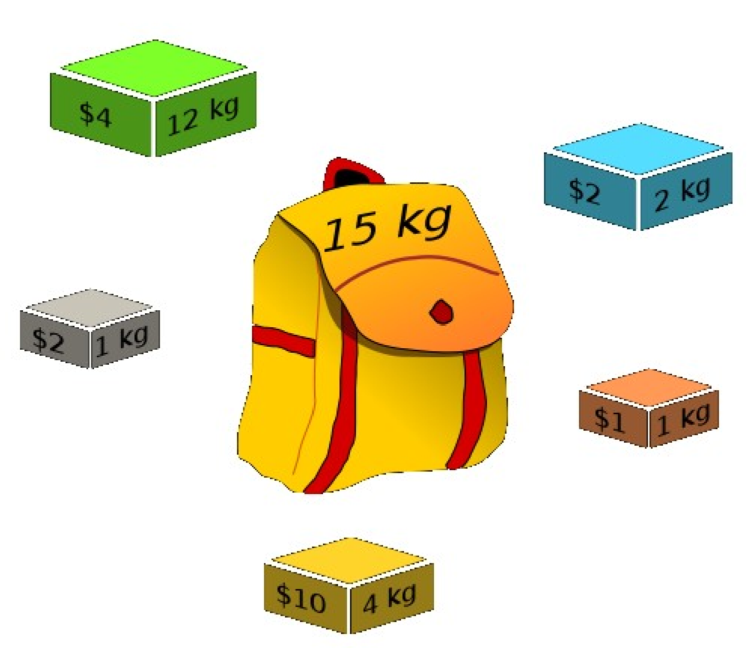
- for each edge, create *k* formulas

$$\phi_{ijc} = \neg p_{ic} \vee \neg p_{jc} \qquad (1 - x_{ic}) + (1 - x_{jc}) \geq 1$$

indicating that a pair of connected nodes *i* and *j* may not both have color *c* at the same time

- for each variable the requirement that its value can only be zero or one

Knapsack using ILP



- **Given:**
  - N items with sizes $a_1,\ldots,a_N$, prices $p_1,\ldots,p_N$
  - A maximum weight W
- **Find:**
  - a subset of items I → *variables* x_i, *each with domain {0,1}*
- **Such that:**
  - $\sum_{i=1}^{n} p_i x_i$ is **maximal** (very valuable knapsack)
  - $\sum_{i=1}^{n} a_i x_i \leq W$ (knapsack with low weight)

Also try: Set Cover

Comparison

	Variables	Constraints	Optimization	Special Technology
SAT Solver	Boolean (0/1)	Clauses	Not supported directly	Clause learning, unit propagation, pure literals
CP Finite Domain Solver	Finite domain	Many	Many	Propagation
LP Solver	Real	Linear	Linear	Interior points, simplex
ILP Solver	Integer	Linear	Linear	Interior points, simplex, cut-and-branch