Lecture 6: Constraint satisfaction: algorithms

- Basic algorithmic framework for solving CSPs
- DPLL algorithm for Boolean constraints in CNF

Constraint satisfaction: algorithms

- For some classes of constraints there are efficient special purpose algorithms (domain specific methods/constraint solvers).
- But now we consider general methods consisting of
  - constraint propagation algorithms and
  - search methods.
- We discuss the basic structure of such a general method (procedure Solve below), the components used, and their interaction.
- The DPLL algorithm for solving Boolean constraint satisfaction problems given in CNF is presented as an example of such a general method.

Constraint Programming: Basic Framework

```
procedure Solve:
var continue: Boolean;
continue:= TRUE;
while continue and not Happy do
   Preprocess;
   Constraint Propagation;
   if not Happy then
      if Atomic then
         continue:= FALSE
      else
         Split;
         Proceed by Cases
      end
   end
end
```


Solve

- The procedure Solve takes as input a constraint satisfaction problem (CSP) and transforms it until it is solved.
- It employs a number of subprocedures (Happy, Preprocess, Constraint Propagation, Atomic, Split, Proceed by Cases).
- The subprocedures Happy and Atomic test the given CSP to check the termination condition for Solve.
- The subprocedures Preprocess and Constraint Propagation transforms the given CSP to another one that is equivalent to it.
- Split divides the given CSP into two or more CSPs whose union is equivalent to the CSP.
- Proceed by Cases specifies what search techniques are used to process the CSPs generated by Split.
- The subprocedures will be explained in more detail below.

Equivalence of CSPs

- To understand Solve we need the notion of equivalence.
- Basically, CSPs $\mathbf{P}_1$ and $\mathbf{P}_2$ are equivalent if they have the same set of solutions.
- However, transformations can add new variables to a CSP and then equivalence is understood w.r.t. the original variables. This can be formalized using a notion of projection.
- For variables $X := x_1, \ldots, x_n$ with the domains $D_1, \ldots, D_n$, $d := (d_1, \ldots, d_n)$ and a subsequence $Y := x_{i_1}, \ldots, x_{i_l}$ of X, the projection $d[Y]$ of d on Y is the tuple $(d_{i_1}, \ldots, d_{i_l})$.
- CSPs $\mathbf{P}_1$ and $\mathbf{P}_2$ are equivalent w.r.t. a set of variables X iff

$$\{d[X] \mid d \text{ is a solution to } \mathbf{P}_1\} = \{d[X] \mid d \text{ is a solution to } \mathbf{P}_2\}$$

- Union of CSPs $\mathbf{P}_1, \ldots, \mathbf{P}_m$ is equivalent w.r.t. X to a CSP $\mathbf{P}_0$ iff

$$\{d[X] \mid d \text{ is a solution to } \mathbf{P}_0\} = \bigcup_{i=1}^{m} \{d[X] \mid d \text{ is a solution to } \mathbf{P}_i\}$$

Solved and Failed CSPs

- For termination we need to defined when a CSP has been solved and when it is failed, i.e., has no solution.
- Let C be a constraint on variables $y_1, \ldots, y_k$ with domains $D_1, \ldots, D_k$ ($C \subseteq D_1 \times \cdots \times D_k$).
- C is solved if $C = D_1 \times \cdots \times D_k$ and $C \neq \emptyset$.
- A CSP is solved if
  — all its constraints are solved
  — no domain of it is empty.
- A CSP is failed if
  — it contains the false constraint $\perp$, or
  — one of its domains or constraints is empty.

Transformations

- In the following we represent transformations of CSPs by means of proof rules.
- A rule

$$\frac{\mathbf{P}_0}{\mathbf{P}_1}$$

  transforms the CSP $\mathbf{P}_0$ to the CSP $\mathbf{P}_1$.
- A rule

$$\frac{\mathbf{P}_0}{\mathbf{P}_1 \mid \cdots \mid \mathbf{P}_1}$$

  transforms the CSP $\mathbf{P}_0$ to the set of CSPs $\mathbf{P}_1, \ldots, \mathbf{P}_n$.

Happy

This is a test applied to the current CSP to see whether the goal conditions set for the original CSP have been achieved. Typical conditions include:

- a solution has been found,
- all solutions have been found,
- a solved form has been reached from which one can generate all solutions,
- it is determined that no solution exists (the CSP is failed),
- an optimal solution w.r.t. some objective function has been found,
- all optimal solutions have been found.

Example. For a CSP

$\langle x_1 + x_2 = x_3, x_1 - x_2 = 0; x_i \in D_i \rangle$

the solved form could be, for example,

$\langle x_1 = x_2, x_3 = 2x_2; x_i \in D_i \rangle$

Preprocess

- Bring constraints to desired syntactic form.
- Example: Constraints on reals.
  Desired syntactic form: no repeated occurrences of a variable.

$$\frac{ax^7 + bx^5y + cy^{10} = 0}{ax^7 + z + cy^{10} = 0, bx^5y = z}$$

  (Notice a new variable is introduced.)

Atomic

- This is a test applied to the current CSP to see whether the CSP is amenable for splitting.
- Typically a CSP is considered atomic if the domains of the variables are either singletons or empty.
- But a CSP can be viewed atomic also if it is clear that search 'under' this CSP is not needed.
  For example, this could be the case when the CSP is "solved" or an optimal solution can be computed directly from the CSP.

Split

- After Constraint Propagation, Split is called when the test Happy fails but the CSP is not yet Atomic.
- A call to Split replaces the current CSP $\mathbf{P}_0$ by CSPs $\mathbf{P}_1, \ldots, \mathbf{P}_n$ such that the union of $\mathbf{P}_1, \ldots, \mathbf{P}_n$ is equivalent to $\mathbf{P}_0$, i.e., the rule

$$\frac{\mathbf{P}_0}{\mathbf{P}_1 \mid \cdots \mid \mathbf{P}_n}$$

  is applied
- A split can be implemented by splitting domains or constraints.
- For efficiency an important issue is the splitting heuristics, i.e. which split to apply and in which order to consider the resulting CSPs.

Split — a domain

- D finite (Enumeration) : $\dfrac{x \in D}{x \in \{a\} \mid x \in D - \{a\}}$
- D finite (Labeling) : $\dfrac{x \in \{a_1, \ldots, a_k\}}{x \in \{a_1\} \mid \ldots \mid x \in \{a_k\}}$
- D interval of reals (Bisection) : $\dfrac{x \in [a..b]}{x \in [a..\frac{a+b}{2}] \mid x \in [\frac{a+b}{2}..b]}$

Split — a constraint

- Disjunctive constraints like
  Start[task1] + Duration[task1] $\leq$ Start[task2] $\vee$
  Start[task2] + Duration[task2] $\leq$ Start[task1]

  can be split using the rule: $\dfrac{C1 \vee C2}{C1 \mid C2}$
- Constraints in "compound" form:

$$\frac{|p(\bar{x})| = a}{p(\bar{x}) = a \mid p(\bar{x}) = -a}$$

Heuristics

Which

- variable to choose,
- value to choose,
- constraint to split.

Examples:

(i) Select a variable that appears in the largest number of constraints (most constrained variable).

(ii) For a domain being an integer interval: select the middle value.

Proceed by Cases

- Various search techniques (covered in Lectures 3 and 4) can be applied.
- A typical solution is to use
  - backtracking,
  - branch and bound
- and combine these with
  - efficient constraint propagation and
  - intelligent backtracking (e.g., conflict directed backjumping)
- As the search trees are typically very big, you tend to avoid techniques where much more than the current branch of the search tree needs to be stored.

Constraint Propagation

- Intuition: Replace a CSP by an equivalent one that is "simpler".
- Reduction rules can reduces domains of variables and/or constraints.
- By constraint propagation we mean applying repeatedly reduction steps.
- Efficient constraint propagation enabling substantial reductions is a key issue for overall performance.

Reduce Domains

- Linear inequalities on integers:
  $$\frac{\langle x < y; x \in [l_x..h_x], y \in [l_y..h_y]\rangle}{\langle x < y; x \in [l_x..h'_x], y \in [l'_y..h_y]\rangle}$$
  where $h'_x = min(h_x, h_y - 1), l'_y = max(l_y, l_x + 1)$

  Example:
  $$\frac{\langle x < y; x \in [50..200], y \in [0..100]\rangle}{\langle x < y; x \in [50..99], y \in [51..100]\rangle}$$

Reduce Constraints

Usually by introducing new constraints.

- Transitivity of $<$: $\dfrac{\langle x < y, y < z; DE\rangle}{\langle x < y, y < z, x < z; DE\rangle}$
  This rule introduces new constraint, $x < z$.

Repeated Domain Reduction: Example

- Consider $\langle x < y, y < z; x \in [50..200], y \in [0..100], z \in [0..100]\rangle$
- Apply above rule to $x < y$:
  $\langle x < y, y < z; x \in [50..99], y \in [51..100], z \in [0..100]\rangle$.
- Apply it now to $y < z$:
  $\langle x < y, y < z; x \in [50..99], y \in [51..99], z \in [52..100]\rangle$
- Apply it again to $x < y$:
  $\langle x < y, y < z; x \in [50..98], y \in [51..99], z \in [52..100]\rangle$

Constraint Propagation Algorithms

- Deal with efficient scheduling of atomic reduction steps
- Try to avoid useless applications of atomic reduction steps
- Termination when local consistency is achieved.
- Depending on the class of constraints different notions of local consistency are achieved.
- The projection rule is a widely applicable and efficient general reduction rule.

Projection rule:

> *Take a constraint C on variables $x_1,\ldots,x_k$. Choose a variable x_i with domain D_i. Remove from D_i each value d for which there is no $(d_1,\ldots,d_i,\ldots,d_k) \in D_1 \times \cdots \times D_k$ such that $(d_1,\ldots,d_i,\ldots,d_k) \in C$ and $d_i = d$.*

Hyper-Arc Consistency

If the projection tule is the only atomic reduction step and it is applied as long as new reductions can be made, then the constraint propagation algorithm achieves a local consistency notion called hyper-arc consistency:

> *A CSP is hyper-arc consistent if for every constraint C on variables $x_1,\ldots,x_k$ and every variable x_i with domain D_i, for each value $d \in D_i$, there is $(d_1,\ldots,d_i,\ldots,d_k) \in D_1 \times \cdots \times D_k$ such that $(d_1,\ldots,d_i,\ldots,d_k) \in C$ and $d_i = d$.*

Example. Consider a CSP
$\langle C_1(x,y,z), C_2(x,z); x \in \{1,2,3\}, y \in \{1,2,3\}, z \in \{1,2,3\}\rangle$
where $C_1 = \{(1,1,2),(1,2,1),(2,3,3)\}$ and
$C_2 = \{(1,1),(2,2),(3,3)\}$.
This is not hyper-arc consistent because for the constraint $C_1(x,y,z)$ 3 belongs to the domain of x but there is no tuple $(3,d_2,d_3) \in C_1$.

Example

- Consider a CSP
  $\langle C_1(x,y,z), C_2(x,z); x \in \{1,2,3\}, y \in \{1,2,3\}, z \in \{1,2,3\}\rangle$
  where $C_1 = \{(1,1,2),(1,2,1),(2,3,3)\}$ and
  $C_2 = \{(1,1),(2,2),(3,3)\}$.
- Applying Projection rule to C_1 and all the variables x,y,z yields
  $\langle C_1(x,y,z), C_2(x,z); x \in \{1,2\}, y \in \{1,2,3\}, z \in \{1,2,3\}\rangle$
- Applying Projection rule to C_2 yields
  $\langle C_1(x,y,z), C_2(x,z); x \in \{1,2\}, y \in \{1,2,3\}, z \in \{1,2\}\rangle$
- Applying Projection rule to C_1 yields
  $\langle C_1(x,y,z), C_2(x,z); x \in \{1\}, y \in \{1,2\}, z \in \{1,2\}\rangle$
- Applying Projection rule to C_2 yields
  $\langle C_1(x,y,z), C_2(x,y,z); x \in \{1\}, y \in \{1,2\}, z \in \{1\}\rangle$
- Applying Projection rule to C_1 yields
  $\langle C_1(x,y,z), C_2(x,y,z); x \in \{1\}, y \in \{2\}, z \in \{1\}\rangle$
  (This CSP is hyper-arc consistent and happens to be solved).

Example: Boolean Constraints

- Happy: one solution has been found.
- Desired syntactic form (for preprocessing): Conjunctive normal form (CNF).
- Preprocessing: Transform a Boolean circuit with constraints to a set of clauses using Tseitin's translation (see Lecture 5).
- This problem of finding a solution to a set of Boolean constraints given in CNF is usually called the propositional satisfiability problem (SAT) and systems for solving the problem SAT solvers.

Boolean Constraints: Propagation

- Basic reduction step is given by the unit clause rule: $\dfrac{S \cup \{l\}}{S'}$

  where S is a set of clauses, l is a unit clause (a literal) and S' is obtained from S by removing
  (i) every clause that contains l and
  (ii) the complement of l from every remaining clause.
  (The complement of a literal: $\overline{v} = \neg v$ and $\overline{\neg v} = v$.)
  Example.
  $\{\neg v_1, v_1 \vee \neg v_2, v_2 \vee v_3, \neg v_3 \vee v_1, \neg v_1 \vee v_4\} \rightsquigarrow \{\neg v_2, v_2 \vee v_3, \neg v_3\}$

- Unit propagation (UP) (aka Boolean Constraint Propagation (BCP)):
  apply the unit clause rule until a conflict (empty clause) is obtained or no new unit clauses are available.
  Example.
  $\{\neg v_2, v_2 \vee v_3, \neg v_3\} \rightsquigarrow \{v_3, \neg v_3\} \rightsquigarrow \{\bot\}$ (conflict)

Boolean Constraints—cont'd

- Split:
  Apply the enumeration rule: $\dfrac{x \in \{0,1\}}{x \in \{0\} \mid x \in \{1\}}$
  Heuristics: a wide variety of heuristics used
- Proceed by cases: backtrack with propagation (and conflict driven backjumping)
- This gives the DPLL-algorithm (Davis-Putnam-Loveland-Logemann) which is the basis of most of the state-of-the-art SAT solvers.

Basic DPLL

Input: S is a set of clauses and M a set of literals
Output: If S has a model satisfying M, a set of literals giving such a model otherwise 'UNSAT'

$\mathrm{DPLL}(S, M)$
$\langle S', M' \rangle := \mathrm{simplify}(S, M)$;
if $S' = \emptyset$ **then** return M'
else if $\bot \in S'$ **then** return 'UNSAT'
else
  $L := \mathrm{choose}(S', M')$;
  $M'' := \mathrm{DPLL}(S' \cup \{L\}, M' \cup \{L\})$;
  if $M'' =$ 'UNSAT' **then** return $\mathrm{DPLL}(S' \cup \{\overline{L}\}, M' \cup \{\overline{L}\})$
  else return M''
  end if
end if

- $\mathrm{DPLL}(S, \{\})$ returns a model satisfying S iff S is satisfiable.

Basic DPLL—cont'd

- DPLL uses two subprocedures.
- $\mathrm{simplify}(S, M)$ implements constraint propagation
  Typically based on UP in which case $\mathrm{simplify}(S, M)$ returns $\langle S', M' \rangle$ where S' is the set of clauses obtained by applying unit propagation to S and M' is M extended with unit literals found when applying UP.
- $\mathrm{choose}(S', M')$ implements the search heuristics, i.e., decides for which variable the splitting rule is applied and which of the branches is considered first.
- The performance of the procedure depends crucially on the constraint propagation techniques and search heuristics.

Enhanced DPLL

- Conflict Driven Clause Learning (CDCL): used in many successful SAT solvers (`minisat, zchaff, BerkMin, ...`)
- Builds on learning new clauses from conflicts and doing non-chronological backjumping based on the learned clauses
- Learned clauses can prune the search space very efficiently (early detection of conflicts)
- Learned clauses provide an interesting basis for heuristics
- Restarting (but keeping learned clauses) seems to improve the robustness of the solver
- Solvers spend most of the time (80–95%) doing unit propagation
- New data structures have been introduced for coping with large instances (100k+ clauses)
  - Watched literal technique
  - Cache-aware implementation (small memory footprint, array based techniques)
  - Handling short clauses

`bczchaff`

- The second home assignment is solved using the Boolean CSP solver `bczchaff`.
- `bczchaff` solves Boolean CSPs given as Boolean circuits.
- It supports a wide range of gate functions (AND, OR, NOT, EQUIV, IMPLY, ODD, EVEN, ITE, [l,u])
- `bczchaff` solver is based on an efficient state-of-the-art SAT checker `zchaff`, which is a highly optimized implementation of the DPLL algorithm with conflict driven clause learning.
- Given a Boolean circuit `bczchaff` simplifies the circuit and then transforms it to CNF using an optimized version of the Tseitin's translation. The CNF form is solved using the `zchaff` engine and results are translated back to refer to the gates of the original circuit.

Example

Find a truth assignment for Boolean variable $b1, \ldots, bn$ such that if 3 to 5 of them are true, then 6 to 8 are false.

example.bc:

```
BC1.0
a0 := IMPLY(a1,a2);
a1 := [3,5](b1,b2,b3,b4,b5,b6,b7,b8,b9,b10);
a2 := [6,8](~b1, ~b2, ~b3, ~b4, ~b5, ~b6, ~b7,
~b8, ~b9, ~b10);
ASSIGN a0;
```

Running `bczchaff`:

```
> bczchaff example.bc
Parsing from example.bc
...
Number of Implication 163
a2 a1 ~b10 ~b9 ~b8 ~b7 ~b6 b5 ~b4 b3 ~b2 b1 a0
Satisfiable
```