Foundations of Computing

Module Introduction

Emmanuel Hebrard (adapted from) João Marques Silva

Outline

1. Constraint Programming
2. Clause Learning in CP
1 Constraint Programming

2 Clause Learning in CP

- Constraint Satisfaction Problems are generalization of Boolean satisfiability to non-Boolean domains
- Standard constraint programming solvers are similar to DPLL
  - No clause learning (Clause-learning CSP solvers existed before CDCL but were not that successful)
  - But stronger propagation

**Constraint Propagation**

Given a constraint $c = (R(c), S(c))$, a propagator is an algorithm that reduce the domains so that the constraint is arc consistent.
Propagators

A constraint solver is a library of constraints, each with its dedicated propagator

Arc Consistency

A constraint $c$ is Arc Consistent on domain $D$ if and only if for every $x \in S(c)$ and for every $j \in D(x)$, there exists a tuple $\sigma \in R(c) \cap \prod_{x \in X} D(x)$ such that $\sigma(x) = j$.

- The constraint can be a clause: arc consistency corresponds to unit propagation
- The constraint can be a primitive relation (e.g., ‘$\leq$’) and arc consistency is easy and efficient
  - Propagation of $x \leq y$:
    - Event lower bound of $x$ ($\min(x)$) has changed: update $\min(y)$ to $\min(x)$
    - Event upper bound of $y$ has changed: update $\max(x)$ to $\max(y)$
    - Do not wake up on other events

- Can be a much larger and more complex relation, even an NP-hard relation
  - E.g., “the graph given by the incidence matrix $x$ is a clique of size greater than or equal to $y$”
  - Arc consistency is not required for correctness (and is NP-hard when the constraint relation is NP-hard)

AllDifferent

- $\text{AllDifferent}(x_1, \ldots, x_n) \iff \forall 1 \leq i < j \leq n, \ x_i \neq x_j$
- For instance: $\text{AllDifferent}(x_1, x_2, x_3, x_4)$
  - $D(x_1) = \{1\}$
  - $D(x_2) = \{1, 2, 3\}, D(x_2) = \{2, 3\}$
  - $D(x_3) = \{1, 2, 3\}, D(x_3) = \{2, 3\}$
  - $D(x_4) = \{1, 2, 3, 4\}, D(x_4) = \{4\}$

- Only two solutions: $(1, 2, 3, 4)$ and $(1, 3, 2, 4)$, therefore:
  - $x_2 = 1, x_3 = 1, x_4 = 1, x_4 = 2, x_4 = 3$ are not viable

- How can we compute that efficiently?
  - Generating and testing the validity all permutations would take exponential time
**AllDifferent**

- \( \text{AllDifferent}(x_1, \ldots, x_n) \iff \forall 1 \leq i < j \leq n, \ x_i \neq x_j \)

- For instance: \( \text{AllDifferent}(x_1, x_2, x_3, x_4, x_5, x_6) \)
  - \( \mathcal{D}(x_1) = \{1, 2, 3, 5\} \)
  - \( \mathcal{D}(x_2) = \{2, 3, 4\} \)
  - \( \mathcal{D}(x_3) = \{3, 5\} \)
  - \( \mathcal{D}(x_4) = \{1, 2, 3, 4, 5\} \)
  - \( \mathcal{D}(x_5) = \{3, 5\} \)
  - \( \mathcal{D}(x_6) = \{4, 5, 6, 7\} \)

A solution of the **AllDifferent** constraint is a *maximal matching* of the graph.

- We can compute a maximal matching in \( O(n^3 m) \) (Hopcroft Karp).
- Cycle: alternative matching. *Strongly Connected Components* are set of vertices all pairwise connected by a cycle. Tarjan’s Algorithm finds them all in \( O(nm) \).
- An edge \((x, v)\) belongs to a strongly connected component iff the value \( v \) is viable for \( x \Rightarrow \) pruning!
**AllDifferent**

- AllDifferent \((x_1, \ldots, x_n) \Leftrightarrow \forall 1 \leq i < j \leq n, \ x_i \neq x_j\)

- For instance: AllDifferent \((x_1, x_2, x_3, x_4, x_5, x_6)\)
  
  - \(D(x_1) = \{1, 2, 3, 5\}\)
  - \(D(x_2) = \{2, 3, 4\}\)
  - \(D(x_3) = \{3, 5\}\)
  - \(D(x_4) = \{1, 2, 3, 4, 5\}\)
  - \(D(x_5) = \{3, 5\}\)
  - \(D(x_6) = \{4, 5, 6, 7\}\)
  - \(D(x_1) = \{1, 2\}\)
  - \(D(x_2) = \{2, 4\}\)
  - \(D(x_3) = \{3, 5\}\)
  - \(D(x_4) = \{1, 2, 4\}\)
  - \(D(x_5) = \{3, 5\}\)
  - \(D(x_6) = \{6, 7\}\)

---

**Constraint Propagation**

- When and how propagators are called?
  
- Typically via a **Constraint Queue** and an **Event Stack**

- The event stack contains events corresponding to domain **reduction**
  
  - Variable \(x\) is assigned a value \(v\)
  - The lower (resp. upper) bound of variable \(x\) has increased (resp. decreased)
  - The domain of variable \(x\) has lost at least one value
  - The domain of variable \(x\) has lost at value \(v\)

- Every propagator **watches** some events

---

**Algorithm 0: Constraint Propagation**

```
repeat
  while Event-Stack \(\neq \emptyset\) do
    e \(\leftarrow\) Event-Stack.pop-back();
    foreach c \(\in\) Watchers(e) do
      Constraint-Queue.add(c);
    if Constraint-Queue \(\neq \emptyset\) then
      c \(\leftarrow\) Constraint-Queue.pop-priority();
      c.propagate(e);
    /* might push events on the event stack */
  until Event-Stack = \emptyset;
```
Sudoku

\[
\begin{array}{ccccccc}
7 & 8 & 9 & 4 & 5 & 6 & 1 \\
8 & 7 & 9 & 4 & 5 & 6 & 2 \\
7 & 8 & 9 & 5 & 6 & 4 & 1 \\
1 & 2 & 3 & 7 & 8 & 9 & 6 \\
2 & 7 & 8 & 9 & 4 & 5 & 6 \\
3 & 6 & 7 & 8 & 4 & 5 & 9 \\
4 & 5 & 6 & 1 & 2 & 3 & 7 \\
5 & 6 & 1 & 2 & 3 & 4 & 7 \\
6 & 1 & 2 & 3 & 4 & 5 & 6 \\
\end{array}
\]
### Sudoku \( BC(\text{AllDifferent}) \)

\[
\begin{array}{cccccccc}
7 & 4 & 6 & 2 & 7 & 4 & 6 & 5 \\
8 & 7 & 5 & 7 & 9 & 2 & 7 & 9 \\
1 & 3 & 5 & 9 & 4 & 6 & 1 & 4 \\
7 & 9 & 3 & 7 & 8 & 9 & 5 & 6 \\
5 & 4 & 2 & 7 & 9 & 3 & 6 & 1 \\
9 & 6 & 3 & 8 & 6 & 2 & 1 & 9 \\
7 & 4 & 6 & 9 & 4 & 5 & 6 & 8 \\
2 & 5 & 6 & 5 & 6 & 8 & 4 & 9 \\
4 & 1 & 8 & 9 & 5 & 2 & 7 & 6 \\
\end{array}
\]

### Sudoku \( AC(\text{AllDifferent}) \)

\[
\begin{array}{cccccccc}
7 & 4 & 6 & 2 & 7 & 4 & 6 & 5 \\
8 & 7 & 5 & 7 & 9 & 2 & 7 & 9 \\
1 & 3 & 5 & 9 & 4 & 6 & 1 & 4 \\
7 & 9 & 3 & 7 & 8 & 9 & 5 & 6 \\
5 & 4 & 2 & 7 & 9 & 3 & 6 & 1 \\
9 & 6 & 3 & 8 & 6 & 2 & 1 & 9 \\
7 & 4 & 6 & 9 & 4 & 5 & 6 & 8 \\
2 & 5 & 6 & 5 & 6 & 8 & 4 & 9 \\
4 & 1 & 8 & 9 & 5 & 2 & 7 & 6 \\
\end{array}
\]
\[
\sum_{i=1}^{n} a_i x_i = K
\]

- **Subset Sum**: given a set of integers and an integer \( K \), does there exist a subset whose sum is equal to \( K \)
  - A variable with domain \( \{0, 1\} \) for each integer, coefficients are the integers
  - Finding a support is NP-hard
  - Therefore, achieving \( AC \) is NP-hard
  - Achieving \( BC \) is NP-hard too, since on \( \{0, 1\} \) domains, a bounds support is a support
- However, one can enforce \( BC \) on each conjunct of:
  \[
  \sum_{i=1}^{n} a_i x_i \leq K \quad \text{and} \quad \sum_{i=1}^{n} a_i x_i \geq K
  \]
\[ \sum_{i=1}^{n} a_i x_i \leq K \]

- Assume that all coefficients are positive
- \( \max(x_i) + \sum_{j=1}^{n} a_j \min(x_j) - \min(x_i) \leq K \)
  - \( x_i \leq K - \sum_{j=1}^{n} a_j \min(x_j) - \min(x_i) \)
- \( \min(x_i) + \sum_{j=1}^{n} a_j \max(x_j) - \min(x_i) \geq K \)
  - \( x_i \geq K - \sum_{j=1}^{n} a_j \max(x_j) - \min(x_i) \)
Example: Kakuro

\[ \sum_{i=1}^{6} x_i = 39 \]

\[ \text{AllDifferent}(\{x_1, \ldots, x_6\}, \{1, \ldots, 9\}) \]

\[
\begin{align*}
x_1 : & \quad \{8, 9\} \\
x_2 : & \quad \{1, 2, 6, 7, 8, 9\} \\
x_3 : & \quad \{8, 9\} \\
x_4 : & \quad \{1, 5, 6, 8, 9\} \\
x_5 : & \quad \{1, 2, 6, 7, 8, 9\} \\
x_6 : & \quad \{4, 5, 8, 9\}
\end{align*}
\]

Propagation

\[ \text{AllDifferent}(\{x_1, x_3\}, \{8, 9\}) \]
Example: Kakuro

\[ \sum_{i=1}^{6} x_i = 39 \]

\text{ALLDIFFERENT}(\{x_1, \ldots, x_6\}, \{1, \ldots, 9\})

\begin{align*}
x_1 : & \{ 8, 9 \} \\
x_2 : & \{ 6, 7 \} \\
x_3 : & \{ 8, 9 \} \\
x_4 : & \{ 5, 6 \} \\
x_5 : & \{ 6, 7 \} \\
x_6 : & \{ 4, 5 \}
\end{align*}

**Propagation**

\[ \sum_{i=1}^{6} x_i = 39 \]

\[ \Rightarrow \min(x_2) \geq 39 - \sum_{i \neq 2} \max(x_i) \]

\[ \Rightarrow \min(x_2) \geq 3, (\& \ \min(x_3) \geq 3 \ & \min(x_4) \geq 2) \]
### Motivation

- Constraint programming has powerful propagation algorithm

- Example, Kakuro:
  - **Constraint Programming**
    - One variable $x_{i,j} \in \{1, \ldots, 9\}$ for every cell
    - For every clue:
      - One AllDifferent constraint and two Cardinality constraints
  - **SAT Encoding**
    - One variable $x_{i,j,v}$ for every cell and every $v \in \{1, \ldots, 9\}$ plus a linear number of clauses (somewhat equivalent)
    - For every clue of size $n$:
      - 9$(n-1)n/2$ binary clauses to encode AllDifferent: unit propagation is not as strong as constraint propagation on AllDifferent
      - SAT encoding of cardinality: unit propagation is not as efficient as constraint propagation on Cardinality

- But no clause learning!
  - Clause learning was developed in CP (even before zChaff and GRASP) but was not as successful
• There are efficient encoding of domains, e.g., *sequential counters*
  
  ▶ $x_v$: variable $x$ takes value $v$, $s_v$: variable $x$ lower than or equal to $v$

• Same space complexity ($O(|D|)$)

• Domain change slightly less efficient
  
  ▶ Assignment, value removal and bound change take $O(|D|)$ time in the SAT encoding
  ▶ They are in constant time in CP
  ▶ However, amortized to the same worst-case down a branch (removing all values one at a time takes $O(|D|)$ time in both cases)
  ▶ There are many more *read* operations than *write* operations

• Domain events correspond to *domain literals*:
  
  ▶ Upper bound of $x$ has changed to $v$: $s_v$
  ▶ Lower bound of $x$ has changed to $v$: $s_{v-1}$
  ▶ Value $v$ was removed from the domain of $x$: $x_v$
  ▶ Value $v$ has been assigned to variable $x$: $x_v$

---

**Lazy Clause Generation**

• Initially only *domain clauses*, constraints are propagated as in CP [Katsirelos & Bacchus]

• For every domain reduction $l$ made by propagating a constraint generate an asserting explanation clause ($p_1 \lor p_2 \lor \ldots \lor l$)
  
  ▶ Used during conflict analysis, but *not* for unit propagation (the propagator already does this pruning)
  ▶ Learn first UIP clauses exactly as CDCL (and unit propagate them)

• Every constraint has a dedicated propagation algorithm *and an explanation algorithm*

  ▶ Explanation clauses can be generated *a posteriori* (during conflict analysis) to avoid unnecessary calls to the explanation algorithm
Example: $x \leq y$

- Propagation of $x \leq y$:
  - Event $\bar{x}_v$ (lower bound of $x$ has changed to $v + 1$): triggers $\bar{y}_v$
    - Explanation clause ($\bar{x}_v \lor \bar{y}_v$)
  - Event $y_v$ (upper bound of $y$ has changed): triggers $x_v$
    - Explanation clause ($x_v \lor \bar{y}_v$)

- Suited for lazy explanation: the context is irrelevant
Explaining `AllDifferent`: Hall sets

- Strongly connected components that do not include \( t \) have as many variables as values (Hall sets)
  - The only way to a free value is via \( t \)
- Consider any edge \((v \rightarrow x)\) connecting a Hall set to a distinct SCC
  - There cannot be a edge between \( x \) and the Hall set of \( v \) otherwise the SCCs would not be distinct
- A Hall set is a set of variables \( X \) such that \( |\bigcup_{x \in X} D(x)| = |X|\)
  - An edge \((v \rightarrow x)\) is arc inconsistent if and only if \( v \) is in a Hall set and \( x \) is not in the same SCC

For instance: `AllDifferent(x_1, x_2, x_3, x_4)`

- \( D(x_1) = \{1, 2, 3\} \)
- \( D(x_2) = \{1, 2, 3\} \)
- \( D(x_3) = \{1, 2, 3\} \)
- \( D(x_4) = \{1, 2, 3, 4\} \)

- \( \{1, 2, 3\} \) is a Hall set, therefore \( \{1, 2, 3\} \) are not viable for \( x_4 \)

- We can use the Hall set as explanation clause:

\[
(s_{1,3} \land s_{2,3} \land s_{3,3}) \implies \neg s_{4,3}
\]

\[
\iff
\neg s_{1,3} \lor \neg s_{2,3} \lor \neg s_{3,3} \lor \neg s_{4,3}
\]

(i.e., if \( x_1 \leq 3 \) and \( x_2 \leq 3 \) and \( x_3 \leq 3 \), then \( x_4 > 3 \))
- Mapping between CSP variables and Boolean variables (can be implicit)

- Propagation of the original constraints is done via propagators (dedicated algorithms)

- Propagators generate explanation clauses, used to encode the conflict graph

- Learn First-UIP clauses with this conflict graph

- Propagate the learnt clauses via unit-propagation