Distributed Constraint Reasoning 1

Michal Jakob
Agent Technology Center,
Dept. of Computer Science and Engineering,
FEE, Czech Technical University

AE4M36MAS Autumn 2014 - Lecture 8
Where are We?

Agent architectures (inc. BDI architecture)
Logics for MAS
Non-cooperative game theory
Cooperative game theory
Auctions
Social choice

Distributed constraint reasoning
Motivating Example: Meeting Scheduling

Formalization? Algorithms?
Lecture Objectives

At the end of 2-lecture series you will

- will be able to express problems as distributed constraint reasoning problems
- will be able to use basic solution algorithms for constraint satisfaction and optimization
Lecture Outline

1. Introduction
2. Definitions and Examples
3. Solution Methods
4. Asynchronous Backtracking Algorithm
5. Bottom-up Algorithms
6. Summary and Outlook
Introduction

Distributed Constraint Reasoning 1
Constraint Reasoning

Constraints pervade our lives (time, money, energy, ...) and usually perceived as elements that **limit solutions** to the problems we face.

From a **computational point of view**, they:
- **reduce the space** of possible solutions
- **encode knowledge** about the problem at hand
- are **key components** for efficiently solving hard problems

Hard computational problems can often be **made tractable** by carefully **considering the constraints** that define the **structure of the problem**.

**General framework**: applies to planning and scheduling, operational research, automated reasoning and decision theory, computer vision... and multiagent systems
Constraint Reasoning can be used to address coordination and optimization problems in MAS.

- Set of agents must come to some agreement, typically via some form of negotiation, about which action each agent should take in order to jointly obtain the best solution for the whole system.

We will consider Distributed Constraint Reasoning Problems where:

- Each agent negotiates locally with just a subset of other agents (usually called neighbours) that are those that can directly influence his/her behaviour.

Suitable for the cooperative setting

- use game theory otherwise
Definitions

Distributed Constraint Reasoning 1
Constraint Network

A constraint network $\mathcal{N}$ is formally defined as a triple $\langle X, D, C \rangle$ where:

- $X = \{x_1, \ldots, x_n\}$ is a set of variables;
- $D = \{D_1, \ldots, D_n\}$ is a set of variable domains, which enumerate all possible values of the corresponding variables; and
- $C = \{C_1, \ldots, C_m\}$ is a set of constraints; where a constraint $C_i$ is defined on a subset of variables $S_i \subseteq X$ which comprise the scope of the constraint
  - $r_i = |S_i|$ is the arity of constraint $i$
Hard vs. Soft Constraints

**Hard constraint** $C_i^h$ is a Boolean **predicate** $P_i$ that defines **valid joint assignments** of variables in the scope

$$P_i : D_{i_1} \times \cdots \times D_{i_r} \rightarrow \{F, T\}$$

**Soft constraint** $C_i^s$ is a **function** $F_i$ that maps every possible joint assignment of all variables in the scope to a real value

$$F_i : D_{i_1} \times \cdots \times D_{i_r} \rightarrow \mathbb{R}$$
Binary constraint networks are those where each constraint (soft or hard) is defined **over two variables**

Binary constraint networks can be represented by a constraint graph

Every constraint network can be mapped **to a binary** constraint network

- requires the addition of variables and constraints
- may increase the size of the model

*Algorithms* explained for binary constraints but can be extended to \( n \)-ary.
Types of Constraint Reasoning Problems

Constraint Satisfaction Problem (CSP)
- **Objective**: find an assignment for all the variables in the network that satisfies all constraints.
- Extension to MaxCSP/MinCSP: Maximize the number of satisfied constraints / minimize the number of violated constraints.

Constraint Optimization Problem (COP)
- **Objective**: find an assignment for all the variables in the network that satisfies all constraints and optimizes a global function.
- **Global function** = aggregation (typically sum) of constrain functions, i.e., \( F = \sum F_i \)

COP provides more **modelling power** on the expense of more complex solution algorithms.
When operating in a decentralized context:
- a set of agents control variables
- agents interact to find a solution to the constraint network
A distributed constraint reasoning problem consists of a constraint network \( \langle X, D, C \rangle \) and a set of agents \( A = \{A_1, \ldots, A_k\} \) where each agent:

- **controls a subset** of the variables \( X_i \subseteq X \)
- is only **aware of constraints** that involve variable it controls
- communicates only with its **neighbours**

1:1 agent-to-variable mapping assumed for algorithm explanation (can be generalized)/
Types of DCR Problems

1. Distributed CSP (DCSP)
2. Distributed COP (DCOP)
Examples / Applications

Distributed Constraint Reasoning
Examples

Many standard benchmark problems in computer science can be viewed as DCOPs

- e.g. graph colouring

As can many real-world applications

- human-agent organization (e.g. meeting scheduling)
- sensor networks and robotics (e.g. channel allocation)
Graph Colouring

- Popular benchmark
- Simple formulation
- Complexity controlled with few parameters:
  - Number of available colors
  - Number of nodes
  - Density (\#nodes/\#constraints)
- Many versions of the problem:
  - CSP, MaxCSP, COP
Graph Colouring: CSP

- **Nodes can take k colors**
- **Any two adjacent nodes should have different colors**
  - If it happens this is a conflict

![Graph Colouring Examples]

**Yes!**

**No!**
Graph Colouring: Min CSP

- Minimize the number of conflicts
Graph Colouring: COP

- Different weights to violated constraints
- Preferences for different colors
Graph Colouring: DCOP

- Each node:
  - controlled by one agent
- Each agent:
  - Preferences for different colors
  - Communicates with its direct neighbours in the graph

- A1 and A2 exchange preferences and conflicts
- A3 and A4 do not communicate
Channel Allocation in Sensor Networks

Find a non-conflicting assignment of communication channels assuming local communication only
DCSP Formalization of Channel Allocation

**Agents** $A = \{A_1, \ldots, A_n\}$ correspond to sensors.

**Variables** $X = \{X_1, \ldots X_n\}$ correspond to selected broadcast channels: $X_i$ is the channel on which the sensor $A_i$ broadcasts.

**Domains** $D = \{D_1, \ldots, D_n\}$ correspond to available channels.

For each pair of sensors $i, j$ that have overlapping broadcast ranges, there is a corresponding Boolean constraint $P_{i,j}$ so that

$$P_{i,j}(X_i, X_j) = T \text{ iff } X_i \neq X_j$$

i.e. sensors will overlapping ranges must use different channels.

**Objective**: Find a channel allocation where no overlapping sensors use the same channel.
Example: Meeting Scheduling

Window 13:00 – 20:00
Duration 1h

Better after 18:00

Window 15:00 – 18:00
Duration 2h

From http://cs.smu.ca/~pawan/wi07/petcu.pdf
Meeting Scheduling Formalization

[13 – 20] [15 – 18]

PL BL

BC

No overlap (Hard)

Equals (Hard)

Preference (Soft)
Solution Approach: DCSP

Distributed Constraint Reasoning 1
Requirements on a Good Algorithm

Soundness/Correctness: the solution returned is valid
Termination: in a finite number of steps
Completeness: finds an (optimal) solution if it exists
DCSP Solution approaches

Top-Down

- Prunning (e.g. Filtering, Hyper-resolution)

- Search (e.g. Asynchronous backtracking)

Bottom-Up (e.g. Distributed breakout)

approximate but scalable

- rarely produce a solution → used for preprocessing

complete but not scalable
Distributed Algorithms

**Synchronous:** agents take steps following some fixed order (or computing steps are done simultaneously, following some external clock).

**Asynchronous:** agents take steps in arbitrary order, at arbitrary relative speeds.

**Partially synchronous:** there are some restrictions in the relative timing of events.
## Synchronous vs Asynchronous

### Synchronous
- A few agents are active, most are **waiting**
- Active agents take decisions with **up-to-date** information
- Low degree of concurrency
- Poor robustness
- Algorithms: direct extensions of centralized ones

### Asynchronous
- All agents are active **simultaneously**
- Information is less updated, **obsolescence** appears
- High degree of **concurrency**
- High **robustness**
- Algorithms: new approaches
Asynchronous Backtracking Algorithm (ABT)

Distributed Constraint Reasoning 1
Asynchronous Backtracking: Assumptions

1. Agents communicate by **sending messages**

2. An agent can send messages to others, iff it knows their identifiers (**directed communication** / no broadcasting)

3. The **delay** transmitting a message is **finite** but random

4. For any pair of agents, messages are **delivered in the order** they were sent

5. Agents **know the constraints in which they are involved**, but not the other constraints

6. Each agent owns a **single variable** (agents = variables)

7. Constraints are **binary** (2 variables involved)

---

**not essential, can be lifted**
Synchronous Backtracking

Agents agree on an variable order and repeat:

1. send partial solution up to $X_{k-1}$ to $k$-th agent.
2. $k$-th agent generates the next extension to this partial solution.
3. if the solution cannot be extended consistently: $k \leftarrow k - 1$ (backtrack control to previous agent).
4. if solution can be extended consistently, $k \leftarrow k + 1$ (pass control to the next agent)
5. if $k < 1$: stop $\rightarrow$ unsolvable.
6. if $k > n$: stop $\rightarrow$ assignment = solution

Problem: Only one agent working at a time $=>$ very inefficient
Asynchronous Backtracking (ABT)

**Revolutionary** idea in 1998

**Fully asynchronous** algorithm
- all agents active, take a value and inform
- no agent has to wait for other agents

**Total order** among agents (to avoid cycles) → **priorities**

**Constraints are directed**: from higher-priority to lower-priority agents

ABT plays in asynchronous distributed context the same role as backtracking in centralized
ABT: Core Principles

**High-priority** agents **decide** on assignment, **lower-priority** have to **accommodate** or say they cannot.

1. Higher-priority agent (j) informs a lower-priority agent (k) of its assignment

2. Lower-priority agent (k) evaluates the shared $c_{jk}$ constraint with its own assignment
   - If permitted $\rightarrow$ no action
   - else $\rightarrow$ look for a value consistent with j
     - If it exists $\rightarrow$ k takes that value
     - else $\rightarrow$ the agent view of k is a **nogood** $\rightarrow$ distributed backtrack

*More communication needed in asynchronous case to compensate for the lack of shared execution state*
ABT: NoGoods

Nogood: conjunction of (variable, value) pairs of higher-priority agents which removes a value of the current (lower-priority) agent.

Example: \( x \neq y, D_x = D_y = \{a, b\} \), \( x \) higher-priority than \( y \):
- when \( x \) assumes \( a \) and a message \([x \leftarrow a]\) arrives to \( y \), the agent \( y \) generates the nogood \( x = a \implies y \neq a \) that removes value \( a \) of \( D_y \)
- if \( x \) changes value, when \([x \leftarrow b]\) arrives to \( y \), the no good \( x = a \implies y \neq a \) is eliminated, value \( a \) is available again and a new nogood removing \( b \) is generated

Nogoods are required to ensure **systematic traversal of search space in asynchronous, distributed context**
ABT: NoGood Resolution

When all values of variable $y$ are removed, the conjunction of the left-hand sides of its nogoods is also a nogood.

**Resolution**: the process of generating a new nogood that is a logical consequence of existing ones.

**Example**:

$x \neq y, \ z \neq y, D_x = D_y = D_z = \{a, b\}, \ x, z$ higher priority than $y$

assume: $x = a \Rightarrow y \neq a; \ z = b \Rightarrow y \neq b$; i.e., all values for $y$ ruled out

then: $x = a \land z = b$ is a nogood

i.e. in a directed form: $x = a \Rightarrow z \neq b$ (assuming $x$ higher-priority than $z$)

(escalating the problem from $y$)
How ABT Works

**Asynchronous** action; spontaneous assignment

Four operations:

- **Assignment**: $j$ takes value $a$: $j$ informs lower priority agents
- **Backtrack (no good)**: $k$ has no consistent values with higher-priority agents: $k$ resolves nogoods and sends $a$ a backtrack (nogood) message
- **New links**: $j$ receives a nogood mentioning $i$, unconnected with $j$: $j$ asks $i$ to set up a link
- **Stop**: “no solution” (empty nogood) detected by an agent: stop

**Solution**: when agents are silent for a while (*quiescence*), every constraint is satisfied => solution;

- detected by specialized algorithms outside ABT
**ABT: Messages**

**Ok?**(\(i \rightarrow k, a\)): higher-priority agent \(i\) informs lower-priority agent \(k\) that it takes value \(a\)

**NoGood**(\(k \rightarrow j, i = a \Rightarrow j \neq b\)): when all \(k\)'s values are forbidden:
- \(k\) requests \(j\) (the nearest higher-priority agent in the nogood) to backtrack
- then: \(k\) forgets \(j\)'s value, \(k\) takes some value
- \(j\) may detect obsolescence of the NoGood message

**AddLink**(\(j \rightarrow i\)): set a link from \(i\) to \(j\), to know \(i\)'s value

**Stop**: there is no solution
**Current context / agent view:** values of higher-priority constrained agents

**NoGood store:** each removed value has a *justifying* nogood

\[ x_i = a \land x_j = b \Rightarrow x_k \neq c \]

- Stored nogoods must be **active**: left-hand side of the nogood satisfied in the current context
- If a nogood is no longer active, it is removed (and the value is available again)
ABT: Graph Coloring Example

Variables $x_1$, $x_2$, $x_3$; $D_1 = \{b, a\}$, $D_2 = \{a\}$, $D_3 = \{a, b\}$

3 agents, lex ordered:

Agent 1

Agent 2

Agent 3

2 difference constraints: $c_{13}$ and $c_{23}$

Constraint graph:

Value-sending agents: $x_1$ and $x_2$

Constraint-evaluating agent: $x_3$

Each agent checks constraints of incoming links: $Agent_1$ and $Agent_2$ check nothing, $Agent_3$ checks $c_{13}$ and $c_{23}$
ABT Example

1. $D_1 = \{b, a\}$
   - $x_1 = b$
   - $D_2 = \{a\}$
   - $x_2 = a$
   - $D_3 = \{a, b\}$

2. $U_1 = \{b, a\}$
   - $D_1 = \{a\}$
   - $x_1 = b \implies x_2 \neq a$
   - $D_2 = \{a\}$
   - $D_3 = \{a, b\}$

3. $D_1 = \{b, a\}$
   - $x_1 = b$
   - $D_2 = \{a\}$
   - $x_2 = a$
   - $D_3 = \{a, b\}$

4. $D_1 = \{b, a\}$
   - $x_1 = b$
   - $D_2 = \{a\}$
   - $x_2 = a$
   - $D_3 = \{a, b\}$

5. $D_1 = \{b, a\}$
   - $x_1 \neq b$
   - $D_2 = \{a\}$
   - $x_2 = a$
   - $D_3 = \{a, b\}$

6. $D_1 = \{b, a\}$
   - $x_1 = a$
   - $D_2 = \{a\}$
   - $x_2 = a$
   - $D_3 = \{a, b\}$
ABT: Why AddLink?

Imagine ABT without AddLink message:

\[ x_2 \] rejects Nogood message as obsolete (because it does not know the value of \( x_1 \)), \( x_3 \) keeps on sending it => infinite loop!!

AddLink avoids it: **obsolete** info is **removed** in finite time
ABT Properties

Soundness/Correctness
- silent network $\iff$ all constraints are satisfied

Completeness
- ABT performs an **exhaustive traversal** of the search space
- Parts not searched: those eliminated by nogoods
- Nogoods are legal: **logical consequences** of constraints
- Therefore, either there is no solution $\Rightarrow$ ABT generates the empty nogood, or it finds a solution if it exists

Termination
- there is no infinite loop (by induction in the depth of the agent)
Asynchronous Weak-Commitment Search (AWC)

ABT problem: highly constraint variables can be assigned very late

Solution: Use dynamic priorities

Change ok? messages to include agent’s current priority

Use min-conflict heuristic: choose assignment minimizing the number of violations
Distributed Breakout Algorithm
ABT Issues

Uneven division of labor: lowest-priority agents do most of the work

Generating nogoods is complex and computationally expensive operation (also for AWC)

Cannot scale to large problems (100’s of variables at most)

➡️ What if we sacrifice completeness?
Hill Climbing
Hill Climbing

Agents asynchronously change their assignments so that they reduce the number of their violated constraints.

Can get stuck in local optima \(\rightarrow\) use techniques to escape local optima.

But: detection of local optima expensive in a distributed system.
Definition (Quasi-local minimum)
An agent is in a quasi-local minimum if it is violating some constraint and neither it nor any of its neighbors can make a change that results in lower cost for all.

Quasi-local minimum can be detected locally
Distributed Breakout Algorithm

**Key idea:** If in a quasi-local minimum, increase the weight of violated constraints

**Messages:**
- `HANDLE-OK?(i → j, x_i)` where `i` is the agent and `x_i` is its current value
- `HANDLE-IMPROVE(i, improve)` where `improve` is the maximum `i` could gain by changing to some other color
\textbf{HANDLE-OK?}(j, x_j)

1 \hspace{1em} \textit{received-ok}[j] \leftarrow \text{TRUE}

2 \hspace{1em} \textit{agent-view} \leftarrow \textit{agent-view} + (j, x_j)

3 \hspace{1em} \textbf{if} \ \forall_{k \in \text{neighbors}} \ \textit{received-ok}[k] = \text{TRUE}

4 \hspace{1em} \textbf{then} \ \text{SEND-IMPROVE}()

5 \hspace{1em} \forall_{k \in \text{neighbors}} \ \textit{received-ok}[k] \leftarrow \text{FALSE}
SEND-IMPROVE()

1. \(cost \leftarrow \text{evaluation of } x_i \text{ given current weights and values.}\)
2. \(my\text{-improve} \leftarrow \text{possible maximal improvement}\)
3. \(new\text{-value} \leftarrow \text{value that gives maximal improvement}\)
4. \(\forall_{k\in\text{neighbors}} k.\text{HANDLE-IMPROVE}(i, my\text{-improve}, cost)\)
HANDLE-IMPROVE(*j*, *improve*, *eval*)

1. \(\text{received-improve}[j] \leftarrow \text{improve}\)
2. \(\text{if } \forall_{k \in \text{neighbors}} \text{received-improve}[k] \neq \text{NONE}\)
3. \(\text{then } \text{SEND-OK}\)
4. \(\text{agent-view} \leftarrow \emptyset\)
5. \(\forall_{k \in \text{neighbors}} \text{received-improve}[k] \leftarrow \text{NONE}\)
SEND-OK()

1. if $\forall k \in \text{neighbors} \text{ my-improve} \geq \text{received-improve}[k]$
2. then $x_i \leftarrow \text{new-value}$
3. if $\text{cost} > 0 \land \forall k \in \text{neighbors} \text{ received-improve}[k] \leq 0 \triangleright \text{quasi-local opt.}$
4. then increase weight of constraint violations
5. $\forall k \in \text{neighbors} \ k \text{.HANDLE-OK?}(i \cdot x_i)$
Distributed Breakout Example
Properties

**Theorem (Distributed Breakout is not Complete)**

Distributed breakout can get stuck in local minimum. Therefore, there are cases where a solution exists and it cannot find it.
Why to use DCOPs?

Well-defined problem
- Clear formulation that captures most important aspects
- Many solution techniques
  - Optimal: ABT, ADOPT, DPOP, ...
  - Approximate: DSA, MGM, Max-Sum, ...

Solution techniques that can handle large problems
- approximate
When to Apply DCSP/DCOP?

- **Hard-to bound** problems
- No agreement on a **common model**
- No **trusted** third party / **Privacy** concerns
- **Resilience** / **Robustness**
- Limited **communication**
- High **dynamism**

*Efficiency* typically not the reason!
Conclusions

(Distributed) constraint satisfaction (CSP) is a general, widely applicable framework to model problems in terms of Boolean constraints over variables.

Distributed CSP is required if there are constraints on communication or disclosure of private information, problem is difficult to formalize centrally or the system needs to be resilient.

Top-down and bottom-up techniques exist:
- top-down are complete but computationally more intensive on most problems
- bottom-up are faster but can get stuck in local minima

Very active areas of research with a lot of progress – new algorithms emerging frequently.