Multi-Agent Problem Solving

Gauthier Picard

MINES Saint-Étienne
LaHC UMR CNRS 5516

gauthier.picard@emse.fr

— Some contents taken from OPTMAS 2011 and OPTMAS-DCR 2014 Tutorials—
Contents

Introduction
  Constraint Satisfaction Problems
  Constraint Optimization Problems
  Application Domains

Multi-Agent Approaches to DisCSP
  ABT and Extensions
  Distributed Local Search Approaches

Multi-Agent Approaches to DCOP
  DCOP Framework
  Asynchronous Distributed Optimisation (ADOPT)
  Dynamic Programming Optimization Protocol (DPOP)

Cooperation for Problem Solving
  Cooperation: Engine of Distributed Problem Solving
  AMAS Framework
  AMAS Illustration

Synthesis
  Panorama
  Using Distributed Problem Solving
Introduction

Motivations

- Multi-agent systems are a way to model decentralised problem solving (privacy, distribution)
- Agents, having personal goals and constraints, negotiate as to reach a global equilibrium
  ⇒ distributed problem solving using agents

Approaches

- Classical CSP solver extensions
- Classical local search solver extensions
- Other point of view: Cooperation-based solving
**Constraint Satisfaction Problems** *(Dechter, 2003)*

**Definition (CSP)**

A CSP is a triplet \( \langle X, D, C \rangle \) such as:

- \( X = \{x_1, \ldots, x_n\} \) is the set of *variables* to instantiate.
- \( D = \{D_1, \ldots, D_m\} \) is the set of *domains*. Each variable \( x_i \) is related to a domain of value.
- \( C = \{c_1, \ldots, c_k\} \) is the set of *constraints*, which are relations between some variables from \( X \) that constrain the values the variables can be simultaneously instantiated to.

**Definition (Solution to a CSP)**

A solution to a CSP is a complete assignment of values from \( D \) to variables from \( X \) such that every constraint in \( C \) is satisfied.
Issues in CSP

Classical CSPs

- Constraint satisfaction is NP-complete in general
- Constraints are generally expressed as binary constraints
- The topology of a constraint-based problem can be represented by a constraint network, in which vertexes represent variables and edges represent binary constraints between variables

Extensions

- Distribution: variables, constraints
  - ex.: constraint $c_i$ belongs to stakeholder $j$, $\phi(c_i) = j$ (or $\text{belongs}(c_i, j)$)
- Dynamics: adding removing variables and/or constraints at runtime
Constraint Optimization Problems

Sometimes satisfaction is not possible

- Overconstrained problem
- Solution is not binary

Switch from satisfaction to optimization

- Minimizing the number of violated constraints
- Minimizing the cost of violated constraints
- Maximizing the overall utility of the system
- . . .
Application Domains
Contents

Introduction

Multi-Agent Approaches to DisCSP
   ABT and Extensions
   Distributed Local Search Approaches

Multi-Agent Approaches to DCOP

Cooperation for Problem Solving

Synthesis
Multi-Agent Approaches to CSP

- **Complete and asynchronous solvers** for combinatorial problems, within the DisCSP framework, such as Asynchronous Backtracking (ABT) or Asynchronous Weak-Commitment Search (AWCS)

- **Distributed local search** methods, such as Distributed Breakout Algorithm (DBA) or Environment, Reactive rules and Agents (ERA) approach

- **Self-organising methods** inspired by the previous ones
Asynchronous Algorithms for DisCSP

Idea

- Inspired by classical centralised algorithms to solve CSP
- Each agent is responsible for assigning one (or several) variables
- Agents propose values to some other agents (depending on the organisation i.e. constraint network)

Main algorithm: Asynchronous backtracking (ABT) (Yokoo, 2001)

- Agents will perform a distributed version of the backtracking procedure
- ABT is complete
- Extensions exist to handle dynamics

Definition (DisCSP or DCSP)

A DisCSP (or DCSP) is a 5-uplet \( \langle A, X, D, C, \phi \rangle \) where \( \langle X, D, C \rangle \) is a CSP, \( A \) is a set of agents and \( \phi : X \leftrightarrow A \) is a function assigning variables from \( X \) to agents from \( A \).
Centralised Backtracking

\[
i \leftarrow 0 \\
D'_i \leftarrow D_i \\
\text{while } 0 \leq i < n \text{ do} \\
\hspace{1em} x_i \leftarrow \text{null} \\
\hspace{1em} ok? \leftarrow \text{false} \\
\hspace{1em} \text{while not } ok? \text{ and } D'_i \text{ not empty do} \\
\hspace{2em} a \leftarrow \text{a value from } D'_i \\
\hspace{2em} \text{remove } a \text{ from } D'_i \\
\hspace{2em} \text{if } a \text{ is in conflict with } \{x_0, \ldots, x_{i-1}\} \text{ then} \\
\hspace{3em} x_i \leftarrow a \\
\hspace{3em} ok? \leftarrow \text{true} \\
\hspace{2em} \text{end} \\
\hspace{1em} \text{end} \\
\hspace{1em} \text{if } x_i \text{ is null then backtrack} \\
\hspace{2em} i \leftarrow i - 1 \\
\text{else} \\
\hspace{2em} i \leftarrow i + 1 \\
\hspace{2em} D'_i \leftarrow D_i \\
\text{end} \\
\text{end}
\]

**Algorithm 1:** A classical centralised backtracking search method
Asynchronous Backtracking (ABT) *(Yokoo, 2001)*

- First complete asynchronous algorithm for DisCSP solving
- Asynchronous:
  - All agents active, take a value and inform
  - No agent has to wait for other agents
- Total order among agents: to avoid cycles
  - $i < j < k$ means that: $i$ more priority than $j$, $j$ more priority than $k$
- Constraints are directed, following total order
- ABT plays in asynchronous distributed context the same role as backtracking in centralized
Directed: from higher to lower priority agents

Higher priority agent \((j)\) informs the lower one \((k)\) of its assignment

Lower priority agent \((k)\) evaluates the constraint with its own assignment
  - If permitted, no action
  - else it looks for a value consistent with \(j\)
    - If it exists, \(k\) takes that value
    - else, the agent view of \(k\) is a nogood, backtrack
ABT: Directed Constraints

- Directed: from higher to lower priority agents
- Higher priority agent \((j)\) informs the lower one \((k)\) of its assignment
- Lower priority agent \((k)\) evaluates the constraint with its own assignment
  - If permitted, no action
  - else it looks for a value consistent with \(j\)
    - If it exists, \(k\) takes that value
    - else, the agent view of \(k\) is a nogood, backtrack

generates nogoods: eliminate values of \(k\)
ABT: Nogoods

Definition (Nogood)
Conjunction of (variable, value) pairs of higher priority agents, that removes a value of the current one

Example

- $x \neq y$, $d_x = d_y = \{a, b\}$, $x$ higher than $y$
- When $[x \leftarrow a]$ arrives to $y$, this agent generates the nogood $[x = a \Rightarrow y \neq a]$ that removes value $a$ of $d_y$
- If $x$ changes value, when $[x \leftarrow b]$ arrives to $y$, the nogood $[x = a \Rightarrow y \neq a]$ is eliminated, value $a$ is again available and a new nogood removing $b$ is generated
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 the new nogood

Example

- $x \neq y, z \neq y, d_x = d_y = d_z = \{a, b\}, x, z$ higher than $y$

- $x = a \Rightarrow y \neq a$

- $x = a \land z = b$ is a nogood

- $z = b \Rightarrow y \neq b$

- $x = a \Rightarrow z \neq b$ (assuming $x$ higher than $z$)
How ABT works

- **ABT agents**: asynchronous action, spontaneous assignment
- **Assignment**: $j$ takes value $a$, $j$ informs lower priority agents
- **Backtrack**: $k$ has no consistent values with high priority agents, $k$ resolves nogoods and sends a backtrack message
- **New links**: $j$ receives a nogood mentioning $i$, unconnected with $j$; $j$ asks $i$ to set up a link
- **Stop**: “no solution” detected by an agent, stop
- **Solution**: when agents are silent for a while (quiescence), every constraint is satisfied $\rightarrow$ solution; detected by specialized algorithms
ABT: Messages

\[
\text{Ok}\quad ?(i \rightarrow k, a) \quad : \quad \text{▶ i informs k that it takes value a}
\]

\[
\text{Ngd}(k \rightarrow j, i = a \Rightarrow j \neq b) \quad : \quad \text{▶ all k values are forbidden}
\]

\[
\text{k requests j to backtrack}
\]

\[
\text{k forgets j value}
\]

\[
\text{k takes some value}
\]

\[
\text{j may detect obsolescence}
\]

\[
\text{Addl}(j \rightarrow i) : \quad \text{▶ set a link from i to j, to know i value}
\]

\[
\text{Stop} : \quad \text{▶ there is no solution}
\]
ABT: Messages

- **Ok? (i → k, a):**
  - i informs k that it takes value a
ABT: Messages

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

- **Ngd**(\(k \rightarrow j, i = a \Rightarrow j \neq b\))
  - all \(k\) values are forbidden
  - \(k\) requests \(j\) to backtrack
  - \(k\) forgets \(j\) value
  - \(k\) takes some value
  - \(j\) may detect obsolescence
**ABT: Messages**

- **Ok?**(i → k, a):
  - i informs k that it takes value a

- **Ngd**(k → j, i = a ⇒ j ≠ b)
  - all k values are forbidden
  - k requests j to backtrack
  - k forgets j value
  - k takes some value
  - j may detect obsolescence

- **Addl**(j → i):
  - set a link from i to j, to know i value
ABT: Messages

- **`Ok?(i → k, a)`**:  
  - `i` informs `k` that it takes value `a`

- **`Ngd(k → j, i = a ⇒ j ≠ b)`**:  
  - all `k` values are forbidden
  - `k` requests `j` to backtrack
  - `k` forgets `j` value
  - `k` takes some value
  - `j` may detect obsolescence

- **`AddI(j → i)`**:  
  - set a link from `i` to `j`, to know `i` value

- **Stop**:  
  - there is no solution
ABT Procedures

when received \((\text{ok}?, (x_j, d_j))\) do — (i)
  revise \(\text{agent\_view}\);
  check\_agent\_view;
end do;

when received \((\text{nogood}, x_j, \text{nogood})\) do — (ii)
  record \(\text{nogood}\) as a new constraint;
  when \(\text{nogood}\) contains an agent \(x_k\) that is not its neighbor
    do request \(x_k\) to add \(x_j\) as a neighbor,
        and add \(x_k\) to its neighbors; \textbf{end do};
  old\_value \(\leftarrow\) current\_value; check\_agent\_view;
  when old\_value = current\_value do
    send \((\text{ok}?, (x_j, \text{current\_value}))\) to \(x_j\); \textbf{end do}; \textbf{end do};

procedure check\_agent\_view
  when agent\_view and current\_value are not consistent do
    if no value in \(D_i\) is consistent with agent\_view then backtrack;
    else select \(d \in D_i\) where agent\_view and \(d\) are consistent;
        current\_value \(\leftarrow d\);
        send \((\text{ok}?, (x_j, d))\) to neighbors; \textbf{end if}; \textbf{end do};

procedure backtrack
  generate a nogood \(V\) — (iii)
  when \(V\) is an empty nogood do
    broadcast to other agents that there is no solution,
        terminate this algorithm; \textbf{end do};
    select \((x_j, d_j)\) where \(x_j\) has the lowest priority in a nogood;
    send \((\text{nogood}, x_j, V)\) to \(x_j\);
    remove \((x_j, d_j)\) from agent\_view;
    check\_agent\_view;

Algorithm 2: ABT Procedures
ABT: Correctness and Completeness

- **Correctness**
  - silent network $\Leftrightarrow$ 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 exists
ABT: Remarks

- Fixed ordered organisation
  - Agents only communicate with agents with lower priority for *ok*?
  - Agents only communicate with the agent with direct higher priority for *nogood*
- No termination procedure is given (but it is easily implemented using Dijkstra’s tokens)
- Really distributable
- What if $x_0$ disappears?...

Extensions and Filiation

- **Changing ordering** in every conflict with AWCS ([Yokoo, 2001](#))
- Satisfaction → **Optimisation** with ADOPT (Asynchronous B&B) ([Modi et al., 2005](#)) or APO ([Maillet and Lesser, 2006](#))
- **Adding new agents** at runtime in DynAPO ([Maillet, 2005](#))
Asynchronous Weak-Commitment Search (AWCS) (Yokoo, 2001)

Algorithm 3: AWCS Procedures

procedure check_agent_view
    when agent_view and current_value are not consistent do
        if no value in D, is consistent with agent_view then backtrack;
        else select d ∈ D, where agent_view and d are consistent
            and d minimizes the number of constraint violations
            with lower priority agents; — (i)
            current_value ← d;
            send (ok?, (x, d, current_priority)) to neighbors;
        end if;
    end do;

procedure backtrack
    when V is an empty nogood do
        broadcast to other agents that there is no solution,
        terminate this algorithm; end do;
    when V is a new nogood do — (ii)
        send V to the agents in the nogood;
        current_priority ← 1 + p_max,
        where p_max is the maximal priority value of neighbors;
        select d ∈ D, where agent_view and d are consistent,
        and d minimizes the number of constraint violations
        with lower priority agents;
        current_value ← d;
        send (ok?, (x, d, current_priority)) to neighbors; end do;
Distributed Local Search Approaches

Local Search (LS)

- *LS* algorithms explore the search space from state to state
- Always tend to improve the current state of the system
- Can naturally handle dynamics (adding constraints, changing values)
- Time efficient
- Not complete and require some subtle parameter tuning

choose an initial assignment \( s(0) \)

\[
\textbf{while } s(t) \text{ not terminal } \textbf{do}
\]

- select an acceptable move \( m(t) \) to another assignment
- apply move \( m(t) \) to reach \( s(t + 1) \)

\[
t := t + 1
\]

\textbf{end}

**Algorithm 4**: A generic centralised local search algorithm
Classical Centralised LS Algorithms

Common points

- Initial point (ex: randomly chosen)
- Termination criterion (ex: limit time, $\delta$ improvement)
- Acceptable move (ex: $+\epsilon$)

Famous LS Methods

- Tabu search (Glover and Laguna, 1997)
- Simulated annealing (Kirkpatrick et al., 1983)
- Iterative Breakout method (Morris, 1993)
Distributed Breakout Algorithm (DBA)

wait_ok? mode — (i)
when received (ok?, x_j, d_j) do
  add (x_j, d_j) to agent_view;
when received ok? messages from all neighbors do
  send_improve;
  goto wait_improve mode; end do;
  goto wait_ok mode; end do;

procedure send_improve
  current_eval ← evaluation value of current_value;
  my_improve ← possible maximal improvement;
  new_value ← the value which gives the maximal improvement;
  send (improve, x_i, my_improve, current_eval) to neighbors;

wait_improve? mode — (ii)
when received (improve, x_j, improve, eval) do
  record this message;
when received improve? messages from all neighbors do
  send_ok; clear agent_view;
  goto wait_ok mode; end do;
  goto wait_improve mode; end do;

procedure send_ok
  when its improvement is largest among neighbors do
    current_value ← new_value; end do;
  when it is in a quasi-local-minimum do
    increase the weights of constraint violations; end do;
  send (ok?, x_i, current_value) to neighbors;

Algorithm 5: DBA Message Handler
Distributed Breakout Algorithm (DBA) (cont.)

Principles of DBA (Yokoo, 2001)

- **Distribution difficulties:**
  1. If two neighbouring agents concurrently change their value, the system may oscillate.
  2. Detecting the fact that the whole system is trapped in local minimum requires the agents to globally exchange data.

- **DBA answers:**
  1. For a given neighbourhood, only the agent that can maximally improve the evaluation value is given the right to change its value.
  2. Agents only detect quasi-local-minimum, which is a weaker local-minimum that can be detected only by local interactions.
Distributed Breakout Algorithm (DBA) (cont.)

Remarks

- Distributed version of the iterative breakout algorithm
- Two-mode behaviour alternating between exchange of potential improvement and exchange of assignments
- There is no order over the agents society → neighbourhoods
- The system halts if a solution is found or if the weight of constraints have reached a *predefined upper bound*
  → the **only** difficult parameter to set
- DBA is not complete
- DBA is able to detect the termination or a global solution only by reasoning on local data.
Environment, Reactive rules and Agents (ERA) (Liu et al., 2002)

Components

- A discrete grid **environment**, that is used as a communication medium
- **Agents** that evolves in some regions of the grid (their domain)
  - Agents move *synchronously*
  - Agents cannot move in the domain of other agents, but can mark it with the number of potential conflicts
  - These marks represents therefore the number of violated constraints if an agent chooses the marked cell
- **Rules** (*moves*) that agent follow to reach an equilibrium
  - 3 possible actions
    - *least-move*: the next cell is the one with minimum cost
    - *better-move*: the next cell is randomly chosen and if it has less conflicts than the actual one the agent moves else the agent rests
    - *random-move*: the next cell is randomly chosen
  - A decision consists in a random Monte-Carlo choice of the action to perform
Environment, Reactive rules and Agents (ERA) (Liu et al., 2002) (cont.)

$t \leftarrow 0$
initialise the grid to 0 violation in each cell; foreach agent i do
| randomly move to a cell of row i
end
while $t < t_{max}$ and no solution do
    foreach agent i do
        select a move behaviour
        compute new position
        decrease markers in all cells with past violations
        increase markers in all cells with new violations
    end
    $t \leftarrow t + 1$
end

Algorithm 6: ERA Outline
Remarks

- The environment is the communication medium
  - ✔ There is no asynchronous mechanisms and message handling
  - ✗ Synchronisation point: high synchronous solving process with no benefit from distribution, in case of high connected constraint networks

- ERA quickly finds assignments close to the solution → repairing issues
- ✗ Redundant usage of random choices: non-guided method, close to random walk, and non complete
- ✗ Termination: ERA requires a time limit \( t_{\text{max}} \) (problem-dependant)
Contents

Introduction

Multi-Agent Approaches to DisCSP

Multi-Agent Approaches to DCOP
  DCOP Framework
  Asynchronous Distributed Optimisation (ADOPT)
  Dynamic Programming Optimization Protocol (DPOP)

Cooperation for Problem Solving

Synthesis
DCOP Framework

Motivations

- In dynamic and complex environments not all constraints can be satisfied completely
- Satisfaction → Optimisation (combinatorial)
  - ex: minimizing the number of unchecked constraints, minimizing the sum of the costs of violated constraints, etc.

Definition (DCOP)

A DCOP is a DCSP \( \langle A, X, D, C, \phi \rangle \) with

- a cost function \( f_{ij} : D_i \times D_j \mapsto \mathbb{N} \cup \infty \) for each pair \( x_i, x_j \)
- an objective function \( F : D \mapsto \mathbb{N} \cup \infty \) evaluating an assignment \( A \) with \( f_{ij}(d_i, d_j) \) for each pair \( x_i, x_j \)
DCOP Framework (cont.)

<table>
<thead>
<tr>
<th>Neighbors</th>
<th>x_i</th>
<th>x_j</th>
<th>f(x_i,x_j)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

(a) Example constraint graph. (b) An example ordering formed from the constraint graph in (a). (c) Flow of VALUE and VIEW messages between agents.

Objective Function

\[ F(A) = \sum_{x_i, x_j \in X} f_{ij}(d_i, d_j) \text{ where } x_i \leftarrow d_i \text{ and } x_j \leftarrow d_j \text{ in } A \]

In figure (a):
- \( F(\{(x_1,0),(x_2,0),(x_3,0),(x_4,0)\}) = 4 \)
- \( F(\{(x_1,1),(x_2,1),(x_3,1),(x_4,1)\}) = 0 \)
DCOP Algorithms

Complete Algorithms
- Search Algorithms: e.g., SBB, ADOPT, AFB
- Inference Algorithms: e.g., DPOP, Action-GDL

Incomplete Algorithms
- Search Algorithms: e.g., MGM, DBA, DSA
- Inference Algorithms: e.g., max-sum
- Sampling Algorithms: e.g., DUCT, D-Gibbs
Asynchronous Distributed Optimisation (ADOPT) (Modi et al., 2005)

ADOPT: DFS tree (pseudotree)

ADOPT assumes that agents are arranged in a DFS tree:

- constraint graph $\rightarrow$ rooted graph (select a node as root)
- some links form a tree / others are backedges
- two constrained nodes must be in the same path to the root by tree links (same branch)

Every graph admits a DFS tree: DFS graph traversal
ADOPT Features

- Asynchronous algorithm
- Each time an agent receives a message:
  - Processes it (the agent may take a new value)
  - Sends VALUE messages to its children and pseudochildren
  - Sends a COST message to its parent
- Context: set of (variable value) pairs (as ABT agent view) of ancestor agents (in the same branch)
- Current context:
  - Updated by each VALUE message
  - If current context is not compatible with some child context, the later is initialized (also the child bounds)
ADOPT Procedures

Initialize
(1) threshold ← 0; CurrentContext ← {}; 
(2) forall d ∈ D, s_j ∈ Children do 
(3) lb(d, s_j) ← 0; t(d, s_j) ← 0; 
(4) ub(d, s_j) ← Inf; context(d, s_j) ← {}; endif; 
(5) d_i ← d that minimizes LB(d); 
(6) backTrack; 

when received (THRESHOLD, t, context)
(7) if context compatible with CurrentContext: 
(8) threshold ← t; 
(9) maintainThresholdInvariant; 
(10) backTrack; endif; 

when received (TERMINATE, context)
(11) record TERMINATE received from parent; 
(12) CurrentContext ← context; 
(13) backTrack; 

when received (VALUE, (s_j, d_j))
(14) if TERMINATE not received from parent: 
(15) add (s_j, d_j) to CurrentContext; 
(16) forall d ∈ D, s_j ∈ Children do 
(17) if context(d, s_j) incompatible with CurrentContext: 
(18) lb(d, s_j) ← 0; t(d, s_j) ← 0; 
(19) ub(d, s_j) ← Inf; context(d, s_j) ← {}; endif; enddo; 
(20) maintainThresholdInvariant; 
(21) backTrack; endif; 

when received (COST, s_j, context, lb, ub)
(22) d ← value of s_j in context; 
(23) remove (s_j, d) from context; 
(24) if TERMINATE not received from parent: 
(25) forall (s_j, d_j) ∈ context and s_j is not my neighbor do 
(26) add (s_j, d_j) to CurrentContext; enddo; 
(27) forall d′ ∈ D, s_j ∈ Children do 
(28) if context(d′, s_j) incompatible with CurrentContext: 
(29) lb(d′, s_j) ← 0; t(d′, s_j) ← 0; 
(30) ub(d′, s_j) ← Inf; context(d′, s_j) ← {}; endif; enddo; endif; 
(31) if context compatible with CurrentContext: 
(32) lb(d, s_j) ← lb; 
(33) ub(d, s_j) ← ub; 
(34) context(d, s_j) ← context; 
(35) maintainChildThresholdInvariant; 
(36) maintainThresholdInvariant; endif; 
(37) backTrack; 

Algorithm 7: ADOPT Procedures
ADOPT Messages

- **value** \((parent \rightarrow children \cup pseudochildren, a)\): parent informs descendants that it has taken value \(a\)
- **cost** \((child \rightarrow parent, lowerbound, upperbound, context)\): child informs parent of the best cost of its assignment; attached context to detect obsolescence
- **threshold** \((parent \rightarrow child, t)\): minimum cost of solution in child is at least \(t\)
- **termination** \((parent \rightarrow children)\): sent when \(LB = UB\)
ADOPT Data Structures

1. **Current context** (agent view): values of higher priority constrained agents

2. **Bounds** (for each value, child)

   - lower bounds
   - upper bounds
   - thresholds
   - contexts

- Stored contextes must be active: \( context \in currentcontext \)
- If a context becomes no active, it is removed \( (lb \leftarrow 0, th \leftarrow 0, ub \leftarrow \infty) \)
ADOPT Bounds

\[ \delta(b) = \sum_{i \in \text{curctx}} c_{ij}(a, b) \]

\[ \delta(\text{value}) = \text{cost with higher agents} \]

\[ \begin{array}{c}
\text{lb}_1 \\
\text{ub}_1 \\
\text{lb}_2 \\
\text{ub}_2 \\
\text{lb}_3 \\
\text{ub}_3 \\
\end{array} = \text{cost of lower agents} \]

\[ \text{LB}(b) = \delta(b) + \sum_{x_k \in \text{children}} \text{lb}(b, x_k) \]

\[ \text{UB}(b) = \delta(b) + \sum_{x_k \in \text{children}} \text{ub}(b, x_k) \]

\[ \text{LB} = \min_{b \in d_j} \text{LB}(b) \]

\[ \text{UB} = \min_{b \in d_j} \text{UB}(b) \]
ADOPT Bounds

\[ \delta(value) = \text{cost with higher agents} \]

\[ \delta(b) = \sum_{i \in \text{curctx}} c_{ij}(a, b) \]

\[ \delta(b) = \sum_{i \in \text{curctx}} c_{ij}(a, b) \]

Multi-Agent Problem Solving
ADOPT Bounds

\[ \delta(\text{value}) = \text{cost with higher agents} \]

\[ \delta(b) = \sum_{i \in \text{curctx}} c_{ij}(a, b) \]

\[ \delta(b) = \sum_{i \in \text{curctx}} c_{ij}(a, b) \]

\[ OPT(x_j, ctx) = \min_{d \in d_j} \delta(d) + \sum_{x_k \in \text{children}} OPT(x_k, ctx \cup (x_j, d)) \]

\[ LB(b) = \delta(b) + \sum_{x_k \in \text{children}} lb(b, x_k) \]

\[ UB(b) = \delta(b) + \sum_{x_k \in \text{children}} ub(b, x_k) \]

Multi-Agent Problem Solving

39
ADOPT Bounds

\[ \delta(\text{value}) = \text{cost with higher agents} \]

\[ \delta(b) = \sum_{i \in \text{curctx}} c_{ij}(a, b) \]

\[
\begin{align*}
OPT(x_j, ctx) &= \min_{d \in \delta(d)} \sum_{x_k \in \text{children}} OPT(x_k, ctx \cup (x_j, d)) \\
\end{align*}
\]
ADOPT Bounds

\[ \delta(\text{value}) = \text{cost with higher agents} \]

\[ \delta(b) = \sum_{i \in \text{curctx}} c_{ij}(a, b) \]

\[ \text{OPT}(x_j, \text{ctx}) = \min_{d \in d_j} \delta(d) + \sum_{x_k \in \text{children}} \text{OPT}(x_k, \text{ctx} \cup (x_j, d)) \]

\[ [lb_k, ub_k] = \text{cost of lower agents} \]
ADOPT Bounds

\[ \delta(\text{value}) = \text{cost with higher agents} \]

\[ \delta(b) = \sum_{i \in \text{curctx}} c_{ij}(a, b) \]

\[ \delta(b) = \sum_{i \in \text{curctx}} c_{ij}(a, b) \]

\[ \text{OPT}(x_j, \text{ctx}) = \min_{d \in \mathcal{d}_j} \delta(d) + \sum_{x_k \in \text{children}} \text{OPT}(x_k, \text{ctx} \cup (x_j, d)) \]

\[ \text{LB}(b) = \delta(b) + \sum_{x_k \in \text{children}} \text{lb}(b, x_k) \]

\[ \text{UB}(b) = \delta(b) + \sum_{x_k \in \text{children}} \text{ub}(b, x_k) \]

\[ \text{OPT}(x_j, \text{ctx}) = \min_{d \in \mathcal{d}_j} \delta(d) + \sum_{x_k \in \text{children}} \text{OPT}(x_k, \text{ctx} \cup (x_j, d)) \]

\[ \text{LB}(b) = \delta(b) + \sum_{x_k \in \text{children}} \text{lb}(b, x_k) \]

\[ \text{UB}(b) = \delta(b) + \sum_{x_k \in \text{children}} \text{ub}(b, x_k) \]

\[ \text{LB} = \min_{b \in \mathcal{d}_j} \text{LB}(b) \]

\[ \text{UB} = \min_{b \in \mathcal{d}_j} \text{UB}(b) \]
ADOPT Value Assignment

- An ADOPT agent takes the value with minimum LB
- Eager behavior:
  - Agents may constantly change value
  - Generates many context changes
- Threshold:
  - lower bound of the cost that children have from previous search
  - parent distributes threshold among children
  - incorrect distribution does not cause problems: the child with minor allocation would send a COST to the parent later, and the parent will rebalance the threshold distribution
ADOPT Properties

- For any $x_i$, $LB \leq OPT(x_i, ctx) \leq UB$
- For any $x_i$, its threshold reaches $UB$
- For any $x_i$, its final threshold is equal to $OPT(x_i, ctx)$
- $\rightarrow$ ADOPT terminates with the optimal solution
ADOPT Example

- 4 variables (4 agents) \( x_1, x_2, x_3 \) and \( x_4 \) with \( D = \{a, b\} \)
- 4 binary identical cost functions

<table>
<thead>
<tr>
<th>( x_i )</th>
<th>( x_j )</th>
<th>cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>a</td>
<td>1</td>
</tr>
<tr>
<td>a</td>
<td>b</td>
<td>2</td>
</tr>
<tr>
<td>b</td>
<td>a</td>
<td>2</td>
</tr>
<tr>
<td>b</td>
<td>b</td>
<td>0</td>
</tr>
</tbody>
</table>

- Constraint graph

![Constraint Graph](image-url)
ADOPT Example (cont.)

\[ \begin{align*}
  x_1 &= a \\
  x_2 &= a \\
  x_3 &= a \\
  x_4 &= a
\end{align*} \]
ADOPT Example (cont.)
ADOPT Example (cont.)

\[
\begin{align*}
  x_1 &= a \\
  x_2 &= a \\
  x_3 &= a \\
  x_4 &= a
\end{align*}
\]

\[
\begin{align*}
  x_1 &= a \\
  x_2 &= a \\
  x_3 &= a \\
  x_4 &= a
\end{align*}
\]

\[
\begin{align*}
  x_1 &= b \\
  x_2 &= a \\
  x_3 &= a \\
  x_4 &= a
\end{align*}
\]
ADOPT Example (cont.)

\[ x_1 = a \]
\[ x_2 = a \]
\[ x_3 = a \]
\[ x_4 = a \]

\[ x_1 = a \]
\[ x_2 = a \]
\[ x_3 = a \]
\[ x_4 = a \]

\[ x_1 = b \]
\[ x_2 = b \]
\[ x_3 = b \]
\[ x_4 = a \]
ADOPT Example (cont.)

\[ x_1 = a \]
\[ x_2 = a \]
\[ x_3 = a \]
\[ x_4 = a \]

\[ x_1 = a \]
\[ x_2 = a \]
\[ x_3 = a \]
\[ x_4 = a \]

\[ x_1 = b \]
\[ x_2 = a \]
\[ x_3 = a \]
\[ x_4 = a \]

\[ x_1 = b \]
\[ x_2 = b \]
\[ x_3 = a \]
\[ x_4 = a \]

\[ x_1 = b \]
\[ x_2 = b \]
\[ x_3 = b \]
\[ x_4 = b \]
ADOPT Example (cont.)

\[ \begin{align*}
    x_1 &= a \\
x_2 &= a \\
x_3 &= a \\
x_4 &= a \\
\end{align*} \]

\[ \begin{align*}
    x_1 &= a \\
x_2 &= a \\
x_3 &= a \\
x_4 &= a \\
\end{align*} \]

\[ \begin{align*}
    x_1 &= b \\
x_2 &= a \\
x_3 &= a \\
x_4 &= a \\
\end{align*} \]

\[ \begin{align*}
    x_1 &= b \\
x_2 &= b \\
x_3 &= b \\
x_4 &= b \\
\end{align*} \]

\[ \begin{align*}
    x_1 &= b \\
x_2 &= b \\
x_3 &= b \\
x_4 &= b \\
\end{align*} \]

\[ \begin{align*}
    x_1 &= b \\
x_2 &= b \\
x_3 &= b \\
x_4 &= b \\
\end{align*} \]
Dynamic Programming Optimization Protocol (DPOP) *(PETCU and Faltings, 2005)*

3-phase distributed algorithm

<table>
<thead>
<tr>
<th>PHASES</th>
<th>MESSAGES</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. DFS Tree construction</td>
<td>token passing</td>
</tr>
<tr>
<td>2. Utility phase: from leaves to root</td>
<td><code>util</code> (child → parent, constraint table (-child))</td>
</tr>
<tr>
<td>3. Value phase: from root to leaves</td>
<td><code>value</code> (parent → children ∪ pseudochildren, parent value)</td>
</tr>
</tbody>
</table>
DFS Tree Phase

- **Distributed DFS graph traversal**: token, ID, neighbors($X$)
  1. $X$ owns the token: adds its own ID and sends it in turn to each of its neighbors, which become children.
  2. $Y$ receives the token from $X$: it marks $X$ as visited. First time $Y$ receives the token then parent($Y$) = $X$. Other IDs in token which are also neighbors($Y$) are *pseudoparent*. If $Y$ receives token from neighbor $W$ to which it was never sent, $W$ is pseudochild.
  3. When all neighbors($X$) visited, $X$ removes its ID from token and sends it to parent($X$).

- A node is selected as root, which starts
- When all neighbors of root are visited, the DFS traversal ends
DFS Tree Phase: Example

root

$x_1$ parent of $x_2$
DFS Tree Phase: Example

root

\[ x_1 \text{ parent of } x_2 \]

\[ x_2 \text{ parent of } x_3 \]
\[ x_1 \text{ pseudoparent of } x_3 \]
DFS Tree Phase: Example

root

\[ \begin{array}{c}
\text{x}_1 \\
\text{x}_2 \\
\text{x}_3 \\
\text{x}_4
\end{array} \]

- \( x_1 \) parent of \( x_2 \)
- \( x_2 \) parent of \( x_3 \)
- \( x_1 \) pseudoparent of \( x_3 \)
- \( x_3 \) parent of \( x_4 \)
- \( x_3 \) pseudoparent of \( x_1 \)
DFS Tree Phase: Example

root

$x_1$ parent of $x_2$

$x_2$ parent of $x_3$
$x_1$ pseudoparent of $x_3$

$x_3$ parent of $x_4$
$x_3$ pseudoparent of $x_1$

Multi-Agent Problem Solving
Util Phase

Agent $X$:

- receives from each child $Y_i$ a cost function: $C(Y_i)$
- combines (adds, joins) all these cost functions with the cost functions with $parent(X)$ and $pseudoparents(X)$
- projects $X$ out of the resulting cost function, and sends it to $parent(X)$

From the leaves to the root
Util Phase: Example
Util Phase: Example

\[
\begin{array}{c|c|c}
X & T \\
\hline
a & a & 1 \\
a & b & 2 \\
b & a & 2 \\
b & b & 0 \\
\end{array}
\]

\[
\begin{array}{c|c|c}
X & Y \\
\hline
a & a & 1 \\
a & b & 2 \\
b & a & 2 \\
b & b & 0 \\
\end{array}
\quad
\begin{array}{c|c|c}
X & Z \\
\hline
a & a & 1 \\
a & b & 2 \\
b & a & 2 \\
b & b & 0 \\
\end{array}
\]

parent

children
Util Phase: Example

All value combinations
Costs are the sum of applicable costs
Util Phase: Example

<table>
<thead>
<tr>
<th>X</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>a</td>
</tr>
<tr>
<td>a</td>
<td>b</td>
</tr>
<tr>
<td>b</td>
<td>a</td>
</tr>
<tr>
<td>b</td>
<td>b</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
<th>Z</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
</tr>
<tr>
<td>a</td>
<td>a</td>
<td>a</td>
<td>b</td>
</tr>
<tr>
<td>a</td>
<td>a</td>
<td>b</td>
<td>a</td>
</tr>
<tr>
<td>a</td>
<td>a</td>
<td>b</td>
<td>b</td>
</tr>
<tr>
<td>a</td>
<td>b</td>
<td>a</td>
<td>a</td>
</tr>
<tr>
<td>a</td>
<td>b</td>
<td>a</td>
<td>b</td>
</tr>
<tr>
<td>a</td>
<td>b</td>
<td>b</td>
<td>a</td>
</tr>
<tr>
<td>a</td>
<td>b</td>
<td>b</td>
<td>b</td>
</tr>
<tr>
<td>b</td>
<td>a</td>
<td>a</td>
<td>a</td>
</tr>
<tr>
<td>b</td>
<td>a</td>
<td>a</td>
<td>b</td>
</tr>
<tr>
<td>b</td>
<td>a</td>
<td>b</td>
<td>a</td>
</tr>
<tr>
<td>b</td>
<td>a</td>
<td>b</td>
<td>b</td>
</tr>
<tr>
<td>b</td>
<td>b</td>
<td>a</td>
<td>a</td>
</tr>
<tr>
<td>b</td>
<td>b</td>
<td>a</td>
<td>b</td>
</tr>
<tr>
<td>b</td>
<td>b</td>
<td>b</td>
<td>a</td>
</tr>
<tr>
<td>b</td>
<td>b</td>
<td>b</td>
<td>b</td>
</tr>
</tbody>
</table>

---

**Add**

All value combinations
Costs are the sum of applicable costs

Remove X
Remove duplicates
Keep the min cost
Value Phase

1. The root finds the value that minimizes the received cost function in the util phase, and informs its descendants (children ∪ pseudochildren)

2. Each agent waits to receive the value of its parent / pseudoparents

3. Keeping fixed the value of parent/pseudoparents, finds the value that minimizes the received cost function in the Util phase

4. Informs of this value to its children/pseudochildren

This process starts at the root and ends at the leaves
DTREE: DPOP for DCOPs without backedges
DTREE : DPOP for DCOPs without backedges

Optimal solution:
- linear number of messages
- message size: linear
DTREE : DPOP for DCOPs without backedges

Optimal solution:
linear number of messages
message size: linear
DTREE : DPOP for DCOPs without backedges

Optimal solution:
- linear number of messages
- message size: linear
DTREE : DPOP for DCOPs without backedges

Optimal solution:
linear number of messages
message size: linear
DTREE : DPOP for DCOPs without backedges

X

Y

Z

W

Optimal solution:

linear number of messages

message size: linear
DTREE: DPOP for DCOPs without backedges

Optimal solution: linear number of messages, message size: linear
DTREE : DPOP for DCOPs without backedges
DTREE : DPOP for DCOPs without backedges

Optimal solution:
- linear number of messages
- message size: linear
DPOP for any DCOP

Optimal solution: linear number of messages; message size: exponential
DPOP for any DCOP

\[
\begin{align*}
X & \quad Z \\
\hline
a & \quad a & 1 \\
a & \quad b & 2 \\
b & \quad a & 2 \\
b & \quad b & 0 \\
Y \\
\hline
X & \quad Y \\
\hline
a & \quad a & 1 \\
a & \quad b & 2 \\
b & \quad a & 2 \\
b & \quad b & 0 \\
\hline
Z \\
\hline
Y & \quad Z \\
\hline
a & \quad a & 1 \\
a & \quad b & 2 \\
b & \quad a & 2 \\
b & \quad b & 0 \\
\hline
Y & \quad W \\
\hline
a & \quad a & 1 \\
a & \quad b & 2 \\
b & \quad a & 2 \\
b & \quad b & 0 \\
\hline
W
\end{align*}
\]
DPOP for any DCOP

\[
\begin{array}{c|cc}
X & Z \\
\hline
a & a & 1 \\
a & b & 2 \\
b & a & 2 \\
b & b & 0 \\
\end{array}
\quad
\begin{array}{c|cc}
X & Y \\
\hline
a & a & 1 \\
a & b & 2 \\
b & a & 2 \\
b & b & 0 \\
\end{array}
\quad
\begin{array}{c|cc}
Y & Z \\
\hline
a & a & 1 \\
a & b & 2 \\
b & a & 2 \\
b & b & 0 \\
\end{array}
\quad
\begin{array}{c|cc}
Y & W \\
\hline
a & a & 1 \\
a & b & 2 \\
b & a & 2 \\
b & b & 0 \\
\end{array}
\quad
\begin{array}{c|cc}
Z \\
\hline
a & 2 & 2 \\
b & 2 & 0 \\
\end{array}
\quad
\begin{array}{c|cc}
W \\
\hline
\end{array}
\]
DPOP for any DCOP

\[
\begin{array}{c|cc}
X & Z \\
\hline
a & a & 1 \\
 a & b & 2 \\
b & a & 2 \\
b & b & 0 \\
\end{array}
\]

\[
\begin{array}{c|cc}
X & Y \\
\hline
a & a & 1 \\
 a & b & 2 \\
b & a & 2 \\
b & b & 0 \\
\end{array}
\]

\[
\begin{array}{c|cc}
Y & Z \\
\hline
a & a & 1 \\
 a & b & 2 \\
b & a & 2 \\
b & b & 0 \\
\end{array}
\]

\[
\begin{array}{c|cc}
Y & W \\
\hline
a & a & 1 \\
 a & b & 2 \\
b & a & 2 \\
b & b & 0 \\
\end{array}
\]
DPOP for any DCOP

Optimal solution: linear number of messages
DPOP for any DCOP

Optimal solution: linear number of messages, message size: exponential.
DPOP for any DCOP

\[
\begin{array}{c|c|c}
\hline
X & Z & 1 \\
\hline
a & a & 1 \\
a & b & 2 \\
b & a & 2 \\
b & b & 0 \\
\hline
\end{array}
\quad
\begin{array}{c|c|c}
\hline
X & Y & 1 \\
\hline
a & a & 1 \\
a & b & 2 \\
b & a & 2 \\
b & b & 0 \\
\hline
\end{array}
\quad
\begin{array}{c|c|c|c}
\hline
X & b & \\
\hline
\end{array}
\quad
\begin{array}{c|c|c|c}
\hline
Y & Z & \\
\hline
a & a & 1 \\
a & b & 2 \\
b & a & 2 \\
b & b & 0 \\
\hline
\end{array}
\quad
\begin{array}{c|c|c|c}
\hline
Y & W & \\
\hline
a & a & 1 \\
a & b & 2 \\
b & a & 2 \\
b & b & 0 \\
\hline
\end{array}
\quad
\begin{array}{c|c|c|c}
\hline
Y & b & 1 \\
\hline
\end{array}
\quad
\begin{array}{c|c|c|c}
\hline
Z & W & \\
\hline
\end{array}
\]

Optimal solution: linear number of messages, message size: exponential.
DPOP for any DCOP

Optimal solution: linear number of messages, message size: exponential

Multi-Agent Problem Solving
DPOP for any DCOP

Optimal solution:
- linear number of messages
- message size: exponential
Further Readings: Approximate Algorithms for DCOPs

Complete algorithms
- e.g. ADOPT and DPOP
  ▶ complete
  ▼ slow

Approximate algorithms exist (fast, but sub-optimal in many case)
- Search algorithms
  ▶ DBA, DSA (Zhang et al., 2005), MGM (Maheswaran et al., 2004)
- Inference algorithms
  ▶ Max-sum (Farinelli et al., 2008)
Contents

Introduction

Multi-Agent Approaches to DisCSP

Multi-Agent Approaches to DCOP

Cooperation for Problem Solving
  Cooperation: Engine of Distributed Problem Solving
  AMAS Framework
  AMAS Illustration

Synthesis
Cooperation: the Engine of Distributed Problem Solving

- The *nogoods* (conflictual configurations) and potential solutions communicated by agents to their neighbourhood in ABT or AWCS help agents to cooperatively solve a DisCSP.

- The *min-conflict heuristic* used in AWCS or ERA is a means to represent the fact that agents cooperatively act by minimising the negative impact of their actions.

- The *pheromone* deposited by agents in ERA gives relevant information about the region of the search space and modifies later the behaviour of the other agents.
Towards a Generic Cooperation-based Method (AMAS) *(Picard and Glize, 2006)*

- Cooperation can be viewed as a generic concept manipulated by problem solvers
- It transcends to all the aforementioned methods
- Taking inspiration from biological and socio-economic notions of cooperation
- An agent is unable to find alone the global solution and consequently it has to interact locally with its neighbours in order to find its current actions able to reach its individual goals and help its neighbours

⇒ **Cooperation-based algorithms**
Cooperation in Problem Solving

- Agents cooperate to find a global solution
  - ex: Agents assign values to variables as to find a global assignment without knowing the global state of the system
- Agents that have difficulties (critical level $\kappa$) to find a local solution are top-priority
  - ex: an agent that do not find a “good” value for a long time can choose its next value before the other agents in its neighbourhood ($\nu$)
- Agents acts cooperatively
  - ex: minimizing conflict, minimizing negative impact, etc.
Cooperation: Algorithm

**Algorithm 8**: An algorithm outline based on critical level ($\kappa$) in AMAS

```
foreach agent $i$ do
    set an initial assignment to $x_i$
    $x_i.\kappa \leftarrow 0$
end

while termination conditions not met do concurrently
    order the possible solutions according to their $\kappa$ value
    $x_{\text{worst}} \leftarrow \text{argmax} \{x_j.\kappa \mid x_j \in \nu(x_i)\}$
    assign cooperative value to $x_i$ such as $x_{\text{worst}}.\kappa$ decreases
    compute $x_i.\kappa$
    send $x_i.\kappa$ to neighbours in $\nu(x_i)$
end
```
Example of Cooperative Solving

Cooperative Solving of $n$-queens and $n^2/2$-knights problems (PICARD and GLIZE, 2006)
Contents

Introduction

Multi-Agent Approaches to DisCSP

Multi-Agent Approaches to DCOP

Cooperation for Problem Solving

Synthesis
  Panorama
  Using Distributed Problem Solving
# Panorama

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Type</th>
<th>Memory</th>
<th>Messages</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABT</td>
<td>CSP</td>
<td>Exponential</td>
<td>–</td>
<td>Complete, Static ordering</td>
</tr>
<tr>
<td>AWCS</td>
<td>CSP</td>
<td>Exponential</td>
<td>–</td>
<td>Complete (only with exponential space), Reordering, fast</td>
</tr>
<tr>
<td>DBA</td>
<td>Max-CSP</td>
<td>Linear</td>
<td>Bounded</td>
<td>Incomplete, Fast</td>
</tr>
<tr>
<td>ERA</td>
<td>Max-CSP</td>
<td>Polynomial</td>
<td>n/a</td>
<td>Incomplete, randomness</td>
</tr>
<tr>
<td>ADOPT</td>
<td>COP</td>
<td>Polynomial</td>
<td>Exponential</td>
<td>Complete</td>
</tr>
<tr>
<td>DPOP</td>
<td>COP</td>
<td>Exponential</td>
<td>Linear</td>
<td>Complete</td>
</tr>
</tbody>
</table>

**Table: DCSP and DCOP algorithms**
Using Distributed Problem Solving

Problem and Environment Characteristics

- Geographic distribution
  - ex: agents are physically distributed, and solving the whole problem is not possible in a centralised manner

- Constraint network topology
  - ex: bounded vertex degrees or large constraint graph diameter

- Knowledge encapsulation
  - ex: privacy preserving, limited knowledge

- Dynamics
  - ex: rather than solving the whole problem again, only repair sub-problems

Some Applications

- Frequency assignment
- Scheduling
- Resource allocation, Manufacturing control
- Supply chain
- ...

Multi-Agent Problem Solving
References


