An asynchronous complete method for general distributed constraint optimization

Pragnesh Jay Modi, Wei-Min Shen, Milind Tambe and Makoto Yokoo
Overview

- Problem description
- Previous work
- Simple Adopt
- Adopt
- Evaluation
- Conclusions
Problem Description

- **Constraint Satisfaction Problems (CSP)** – tuple \(<V, D, C>\)
- **DisCSP** – distributed version of CSP: in general, one variable per agent
- **DCOP** – solutions have a “quality” or “cost” (constraints are functions, not predicates)
DCOP: formalization

- DCOP = <V, D, C, F>
- Variables set: V = {v₁, v₂, …, vₙ}
- Domains set: D = {D₁, D₂, …, Dₙ}
- Constraints set: C = {fᵢⱼ: Dᵢ x Dⱼ → N U ∞, where i,j=1..n, i ≠ j}
- Evaluation function: \( F(A) = \sum_{xᵢ, xⱼ \in V} fᵢⱼ(dᵢ, dⱼ) \), where \( xᵢ \leftarrow dᵢ, xⱼ \leftarrow dⱼ \)
- Objective: minimize F(A) (ideally 0)
## Previous work

<table>
<thead>
<tr>
<th>Synchronous Branch &amp; Bound</th>
<th>Synchronous, sequential computation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Distributed greedy repair search algorithm</td>
<td>Incomplete, and requires a central agent for global state maintenance</td>
</tr>
<tr>
<td>Iterative thresholding (hierarchical + maximal DisCSP)</td>
<td>Cannot guarantee optimality</td>
</tr>
</tbody>
</table>
DCOP example

- $V = \{x_1, x_2, x_3, x_4\}$
- $D = \{\{0,1\}, \{0,1\}, \{0,1\}, \{0,1\}\}$

Constraints (can be different):

<table>
<thead>
<tr>
<th>$x_i$</th>
<th>$x_j$</th>
<th>$f_{ij}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
Simple Adopt: preamble

- All agents are in a fixed total priority order
- Definition: view $vw = \{(x_i, d_i), (x_j, d_j), \ldots\}$
- Definition: compatible views don’t differ in any variable assignments
- Definition: local cost $\delta(x_i, vw) = \sum_{x_j \in V} f_{ij}(d_i, d_j)$
  where $x_i \leftarrow d_i$, $x_j \leftarrow d_j$ in $vw$
Problem structure

Constraint graph

Priority ordering

Message flow

VALUE

VIEW

Adrian Petcu/LIA/EPFL

12/13/2002
Simple Adopt: algorithm overview

Each agent does:

- Choose a variable value from its domain
- Send it to linked descendents through VALUE
- Wait for incoming messages and respond:
  - When received “VALUE”:
    - Store the received value in Currentvw (its context)
    - Compute lower bound for this Currentvw
    - Send this bound to its parent through a “VIEW” message
  - When received “VIEW”:
    - if lower bound increased, try to find another value; send it
Simple Adopt: algorithm details 1

# Currentvw: Current view of linked ancestors’ values
# xᵢ/dᵢ : Local variable/value
# c(d): Current lower bound on cost for subtree rooted at child, given xᵢ chooses value d

proc Initialize:

- Currentvw ← {}; dᵢ ← null;
- for all d in Dᵢ :
  - c(d) ← 0;
- go to Hill_climb;
proc Hill_climb:

for all $d$ in $D_i$:

- $e(d)$ is $x_i$'s estimate of cost if it chooses $d$
- $e(d) \leftarrow \delta(x_i, \text{Currentvw } \cup \{(x_i,d)\}) + c(d)$;

- choose $d$ that minimizes $e(d)$
  - prefer current value $d_i$ in case of tie;

- $d_i \leftarrow d$;

- SEND VALUE((x_i,d_i)) to all linked descendents

- SEND VIEW(Currentvw, e(d_i)) to parent;
proc when_received_value(x_i,d_i):
- update Currentvw with (x_i,d_i)
- if(Currentvw changed) then
  - for all d in D_i
  - c(d) ← 0;
- go to Hill_climb;

proc when_received_view(vw,cost):
- d ← value of x_i in vw
- if(vw compatible with Currentvw U {(x_i,d)}) then
  - c(d) ← max(c(d), cost);
  - if( c(d) changed) then
    - go to Hill_climb;

Problem 1: resetting all discovered costs

Note 2: linear space reqs

Note 3: ignoring messages
Example run

<table>
<thead>
<tr>
<th>$x_i$</th>
<th>$x_i$</th>
<th>$f_{ii}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

- **$x_1 = 0$**
  - Cost = 1, $vw = (x_1, 0)$
  - Cost = 2, $vw = \{(x_1, 0), (x_2, 0)\}$
  - Cost = 1, $vw = \{(x_2, 0), (x_3, 0)\}$

- **$x_1 = 1$**
  - Cost = 1, $vw = (x_1, 0)$
  - Cost = 2, $vw = \{(x_1, 0), (x_2, 0)\}$
  - Cost = 1, $vw = \{(x_2, 0), (x_3, 0)\}$

- **$x_2 = 0$**
  - Cost = 0, $vw = \{(x_1, 1)\}$
  - Cost = 0, $vw = \{(x_2, 1), (x_3, 1)\}$

- **$x_2 = 1$**
  - Cost = 2, $vw = \{(x_1, 1), (x_2, 1)\}$
  - Cost = 2, $vw = \{(x_1, 1), (x_3, 1)\}$

- **$x_3 = 0$**
  - Cost = 0, $vw = \{(x_1, 1)\}$
  - Cost = 0, $vw = \{(x_2, 1), (x_3, 1)\}$

- **$x_3 = 1$**
  - Cost = 0, $vw = \{(x_2, 1), (x_3, 1)\}$

- **$x_4 = 0$**
  - Cost = 0, $vw = \{(x_1, 1)\}$
  - Cost = 0, $vw = \{(x_2, 1), (x_3, 1)\}$
**Simple Adopt → Adopt**

- **Improvement 1:**
  - Problem: unnecessary deletion of stored lower bounds at context changes (upon receiving value messages)
  - Solution: store context information for all domain values (linear space requirements, preserves completeness):
    - $X_2$: when received VIEW ($vw, a$):
      - store $c(d) = a$
      - store context($d$)=$vw$ ($vw$ doesn’t contain $x_1$)
    - $X_2$: when received VALUE($x_1 = newval$) check context($d$) against Currentvw of $x_2$; if compatible, then don’t delete $c(d)$
Simple Adopt → Adopt

**Improvement 2:**
- Problem: repeated exploration of portions of the search space at context changes (upon receiving value messages)
- Solution: send cost thresholds to linked descendents

\[ x_i = d_1, c(d_1) = a \]

\[ x_i = d_2 \]
Evaluation

- Graph coloring with 3 colors

### Table 1. GraphColor (Link density=2)

<table>
<thead>
<tr>
<th>n</th>
<th>SynchBB</th>
<th>SynchID</th>
<th>Adopt</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>767</td>
<td>212</td>
<td>125</td>
</tr>
<tr>
<td>10</td>
<td>2239</td>
<td>390</td>
<td>255</td>
</tr>
<tr>
<td>12</td>
<td>7401</td>
<td>544</td>
<td>345</td>
</tr>
<tr>
<td>14</td>
<td>20899</td>
<td>1062</td>
<td>423</td>
</tr>
<tr>
<td>16</td>
<td>&gt;50000</td>
<td>5880</td>
<td>1851</td>
</tr>
<tr>
<td>18</td>
<td>–</td>
<td>14604</td>
<td>3304</td>
</tr>
</tbody>
</table>

### Table 2. GraphColor (Link density=3)

<table>
<thead>
<tr>
<th>n</th>
<th>SynchBB</th>
<th>SynchID</th>
<th>Adopt</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>1955</td>
<td>2220</td>
<td>1717</td>
</tr>
<tr>
<td>10</td>
<td>5251</td>
<td>3046</td>
<td>2413</td>
</tr>
<tr>
<td>12</td>
<td>14713</td>
<td>7468</td>
<td>5589</td>
</tr>
<tr>
<td>14</td>
<td>61847</td>
<td>28610</td>
<td>17425</td>
</tr>
<tr>
<td>16</td>
<td>&gt;100000</td>
<td>46348</td>
<td>21714</td>
</tr>
<tr>
<td>18</td>
<td>–</td>
<td>127476</td>
<td>58846</td>
</tr>
</tbody>
</table>
Conclusions

- Adopt: algorithm for distributed constraint optimization
- Idea: increase lower bounds on solution quality
- Complete – converges to solution (proof provided)
- Asynchronous – enables concurrency
- Significant improvements over SynchBB
- *Still some (potential) problems*…
Thank you!
ANNEX: Adopt: algorithm details 1

# Currentvw: Current view of linked ancestors’ values
# x_i/d_i : Local variable/value
# c(d): Current lower bound on cost for subtree rooted at child, given x_i chooses value d

proc Initialize:
   Currentvw ← { }; d_i ← null; threshold ← 0
   for all d in D_i :
      c(d) ← 0;
      context(d) ← {}
   Hill_climb;
ANNEX: Adopt: algorithm details 2

proc Hill_climb:
- for all d in D_i:
  - e(d) ← δ(x_i, Currentvw U {(x_i,d)}) + c(d);
  - choose d that minimizes e(d)
    - prefer current value d_i in case of tie;
  - if e(d_i)>threshold
    - d_i ← d;
  - childLimit←max(c(d_i),threshold-δ(x_i,Currentvw U {(x_i,d_i)}))
  - SEND VALUE ((x_i; d_i), childLimit) to descendents
  - # only choose variables relevant to local cost
  - Neighborvw = {(x_j,d_j) in Currentvw | x_j neighbor of x_i}
  - viewContext←Neighborvw U{union of contexts of d_i in D}
  - # to preserve completeness – VIEW is for best value d, not current value di
  - SEND VIEW, viewContext, e(d)) to parent;
# proc when_received_value(xj,dj, limit):
- update Currentvw with (xj,dj)
- for all d in Di
  - if(context(d) incompatible with Currentvw)
    - c(d) ← 0;
    - context(d) ← {}
  - if(xj is parent)
    - threshold ← limit
  - go to Hill_climb;

ANNEX: Adopt: algorithm details 3
Change from Simple Adopt
ANNEX: Adopt: algorithm details 4

proc when_received_view(vw,cost):
    d ← value of \( x_i \) in vw
    if vw contains (\( x_i, d \)) //child is my neighbor
        remove (\( x_i, d \)) from vw;
        if vw compatible with Currentvw and cost > c(d) then
            c(d) ← cost;
            context(d) ← vw;
    else //child is not my neighbor
        for all d' in \( D_i \):
            if vw compatible with Currentvw and cost > c(d') then
                c(d') ← cost;
                context(d') ← vw;
        end if;
    end if;
    if( c(d_i) changed) then
        go to Hill_climb;