An Overview of Distributed Constraint Satisfaction and Optimization

IRAD: Analyzing ULS System Behavior and Evolution through Agent-based Modeling and Simulation

Andres Diaz Pace – adiaz@sei.cmu.edu
Joseph Giampapa – garof@sei.cmu.edu
Mark Klein – mk@sei.cmu.edu
John Goodenough – jbg@sei.cmu.edu

31 March 2010
Agenda

Purpose:
• Become familiar with distributed problem-solving algorithms
  – D-COP algorithms as “toolbox” for agent-based modeling

From CSP to DisCSP to DCOP
• Basic concepts and techniques of constraint satisfaction problems (CSP)
• Extension of CSP to a distributed setting using agents
• Main algorithms for distributed CSP (DisCSP)
• Extension of DisCSP to distributed constraint optimization (DCOP)
• Main algorithms for DCOP
• Tradeoffs in DCOP
Why CSP + Agents?

An agent is an autonomous entity with a decision-making capability

- An agent manipulates a set of variables to reach some objective
- For now, we assume that agents are cooperative

CSP is a problem-solving framework for determining the values of a set of variables, subject to constraints on the variables

- Satisfaction of constraints involves costs/utilities (optimization)

Inter-related agent communities resonate with SoS/ULS ideas
Approaches to Distributed Agent Assignment

1. Constraint satisfaction
   • Find a value assignment for all the variables that fulfills all constraints

2. Constraint optimization
   • Find a value assignment for all the variables that optimizes an objective function $f \rightarrow R$
     – maximize utility or minimize cost
   • In practice, large agent networks lead to tradeoffs
     – find a quasi-optimal solution quickly
     – find an optimal solution sharing as little information as possible
     – find an optimal solution with little communication overhead (among agents)
     – find an optimal solution with fixed memory sizes in each agent
     – …
Types of Problems

N-queens
Graph coloring
Meeting scheduling

Several real-life problems can be modeled as constraint satisfaction or constraint optimization problems

• Sensor networks
• Robot patrolling
• Distributed resource allocation
• Distributed planning
• …
# The Path to D-COP

<table>
<thead>
<tr>
<th>Technique</th>
<th>Problem type</th>
<th>Planning Algorithm</th>
<th>Knowledge</th>
</tr>
</thead>
<tbody>
<tr>
<td>CSP</td>
<td>Satisfaction • hard constraints</td>
<td>Centralized</td>
<td>Total knowledge in a single agent</td>
</tr>
<tr>
<td>Dis-CSP</td>
<td>Satisfaction • hard constraints</td>
<td>Distributed • agents • synchronous or asynchronous</td>
<td>• Knowledge distributed among agents • Each agent knows its neighborhood • Partial centralization is possible</td>
</tr>
<tr>
<td>DCOP</td>
<td>Optimization • soft constraints • objective function to be maximized/minimized</td>
<td>Distributed • agents • asynchronous</td>
<td>• Knowledge distributed among agents • Each agent knows its neighborhood • Partial centralization is possible</td>
</tr>
</tbody>
</table>

**CSP (Constraint Satisfaction Problem)**
- Satisfaction problems with hard constraints
- Centralized algorithm
- Total knowledge in a single agent

**Dis-CSP (Distributed Constraint Satisfaction Problem)**
- Satisfaction problems with hard constraints
- Distributed algorithm with synchronous or asynchronous
- Knowledge distributed among agents
- Each agent knows its neighborhood
- Partial centralization is possible

**DCOP (Distributed Constraint Optimization Problem)**
- Optimization problems with soft constraints
- Distributed algorithm
- Knowledge distributed among agents
- Each agent knows its neighborhood
- Partial centralization is possible
## Features in D-COP Algorithms

<table>
<thead>
<tr>
<th></th>
<th>CSP</th>
<th>DIS-CSP</th>
<th>DCOP</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Control &amp; degree of distribution</strong></td>
<td>n/a</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>• Partially-centralized control, distributed control</td>
<td>--</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>• Priority schema</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td><strong>Constraint graph structure</strong></td>
<td>n/a</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>• Dense versus sparse graphs</td>
<td>--</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>• Tree-shaped graphs (e.g., DAGs, or DFS-induced ordering)</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>– If cycles, translate graph to a cycle-free form</td>
<td>--</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>– No constraints among siblings</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td><strong>Communications</strong></td>
<td>n/a</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>• Number of messages sent, size of messages</td>
<td>--</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>• Links to neighboring agents</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td><strong>Quality of solutions</strong></td>
<td>n/a</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>• Complete versus incomplete (heuristic) algorithms</td>
<td>--</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>• Bounds on solution cost</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>
Map of Algorithms

CSP → DIS-CSP → DCOP

CSP

backtracking
local search
...

DIS-CSP

ABT
AWCS
Distributed
breakout

DCOP

branch &
bound

OptAPO

DPOP Family

ADOPT

PC-DPOP

DTree
M-POP
A-POP
...

(partial centralization + backtracking)

NCBB

ADOPT Family

OptAPO

DPOP Family

ADOPT

(partial centralization + backtracking)

NCBB

ADOPT Family

OptAPO

DPOP Family

ADOPT

(partial centralization + backtracking)

NCBB

ADOPT Family

OptAPO

DPOP Family

ADOPT

(partial centralization + backtracking)

NCBB

ADOPT Family

OptAPO

DPOP Family

ADOPT

(partial centralization + backtracking)

NCBB

ADOPT Family

OptAPO

DPOP Family

ADOPT

(partial centralization + backtracking)

NCBB

ADOPT Family

OptAPO

DPOP Family

ADOPT

(partial centralization + backtracking)

NCBB

ADOPT Family
Part I: Constraint Satisfaction (CSP)
CSP Formulation

set of **variables** $X_1, X_2, \ldots, X_n$

each variable has a non-empty **domain** $D(X_i)$ of possible **values**
(usually discrete variables with finite domains)

set of **constraints** $C_1, C_2, \ldots, C_m$

each constraint involves some subset of variables and specifies allowable combinations of values for that subset

The constraints define a **constraint network**

**Diagram:**

- $D(X_1) = \{a, b, c, \ldots\}$
- $D(X_2) = \{a, d, e, \ldots\}$
- $D(X_3) = \{b, e, f, \ldots\}$
- $D(X_n) = \{g, \ldots\}$

**Equations:**

$C(X_i, X_j) = X_i \neq X_j$
(true/false)

- $C_1(X_1, X_2)$
- $C_2(X_2, X_n)$
- $C_3(X_2, X_3)$
- $C_m(X_1, X_n)$

**Graph:**

- Variables $X_1, X_2, X_3, X_n$
- Constraints $C_1, C_2, C_3, C_m$
Solving CSP

state of the problem: assignment of values to some (or all) variables

• consistent: an assignment that does not violate any of the constraints
• complete: an assignment that mentions all the variables

A solution is an assignment that is both consistent and complete
Finding a solution involves search (in the domain space)
Centralized CSP Approach

Traditionally, there is **one central agent** for solving the problem that
- Knows about all the variables, domains, and constraints of the problem
- Executes an algorithm to find a solution

**Main technique**: Backtracking
- Depth-first search for possible assignments of variables to values
- Start with a valid assignment for one variable, and progressively add assignments to variables without violating any constraint
Complexity of Solving CSP

Related to the structure of constraint graph

- Tree-structured problems can be solved in linear time (in # variables)
- **Strategy**: try to reduce problems to trees

1) Remove nodes

- assign values to variables so that the remaining nodes form a tree.
- cycle cutset (when the graph is nearly a tree)
- in general, finding the smallest cycle cutset is NP-hard (heuristics exist)

2) Collapse nodes together

- construct a “tree decomposition” of the constraint graph into a set of connected components (sub-problems), and solve each sub-problem independently
- each sub-problem should be as small as possible (tree width)
- finding a good tree decomposition is NP-hard (heuristic exist)
Note: “Binarization” of Constraints

A constraint can affect any number of variables.
If all the constraints are binary, the variables and constraints can be represented in a constraint graph.

- CSP algorithms can exploit graph search techniques.

A constraint of higher arity can be expressed in terms of binary constraints. Hence, binary CSPs are representative of all CSPs.

- Choice between binary or non-binary constraints depends on the algorithm and the supporting toolkit.

\[
\text{allDifferent}(X,Y,Z)
\]

\[
\begin{align*}
X & \quad Y \\
Y & \quad Z \\
Z & \quad X \\
\end{align*}
\]

\[
\begin{align*}
X & \quad \text{notEqual} \\
Y & \quad \text{notEqual} \\
Z & \quad \text{notEqual} \\
\end{align*}
\]
Map of Algorithms - CSP

CSP

DIS-CSP

DCOP

OptAPO

DPOP Family

ADOPT

ABT
AWCS
Distributed breakout

backtracking
local search
...

branch & bound

(partial centralization +backtracking)

PC-DPOP

DTree
M-POP
A-POP

NCBB
**Example: 4-queens**

Variables: A, B, C, D
Domain: \{1, 2, 3, ..., N\} N = 4
Constraints: for all pairs of queens → noThreat(Q_i, Q_j)

\[
\begin{align*}
A &= \{1, 2, 3, 4\} \\
B &= \{1, 2, 3, 4\} \\
C &= \{1, 2, 3, 4\} \\
D &= \{1, 2, 3, 4\}
\end{align*}
\]
4-queens: Solutions

\[ A = \{1, 2, 3, 4\} \]
\[ B = \{1, 2, 3, 4\} \]
\[ C = \{1, 2, 3, 4\} \]
\[ D = \{1, 2, 3, 4\} \]

\[ A = \{2\} \]
\[ B = \{4\} \]
\[ C = \{1\} \]
\[ D = \{3\} \]

legal solution

\[ A = \{1\} \]
\[ B = \{3\} \]
\[ C = \{2\} \]
\[ D = \{X\} \]

no solution

\[ A = \{1, 2, 3, 4\} \]
\[ B = \{1, 2, 3, 4\} \]
\[ C = \{1, 2, 3, 4\} \]
\[ D = \{1, 2, 3, 4\} \]
Backtracking algorithm

- Start with a “partial solution” that assigns values to some variables in such a way it satisfies the constraints
- Expand the partial solution by adding variables one by one, until you have a complete solution
- **Backtracking operation**: when the value of variable doesn’t satisfy a constraint, change the value. If no values are left, then fail.
Alternative to Backtracking: Iterative search

“Improve what you have until you can't make it better”

Assign a value to every variable, and try to “repair” a flawed solution and produce a valid one

- At each step, apply local search to change the value of one variable
- Select the value that minimizes the number of violated constraints \(\rightarrow\) min-conflict heuristic

\[
\begin{align*}
A &= \{1\} \\
B &= \{2\} \\
C &= \{1\} \\
D &= ?
\end{align*}
\]

\[
\begin{align*}
A &= \{1\} \\
B &= ? \\
C &= \{1\} \\
D &= 2
\end{align*}
\]

\[
\begin{align*}
A &= \{1\} \\
B &= 3 \\
C &= \{1\} \\
D &= \{3\}
\end{align*}
\]

\[
\begin{align*}
A &= \{1\} \\
B &= \{4\} \\
C &= \{1\} \\
D &= \{3\}
\end{align*}
\]
Observations on Iterative Search

The algorithm works well, given a “reasonable” initial state.
The structure of the constraint network is important.
  • e.g., the n-queens defines a “dense” space.

Tradeoff between efficiency and completeness:
  • Local search is faster than a systematic search using backtracking.
  • But finding a valid solution cannot be guaranteed.

Local search techniques can be used in a setting where the problem changes (e.g., queens, positions, dimensions of chessboard) over time.
  • Local search will “repair” the current solution.
  • Not need to start search from scratch for every change.
Complexity of Solving CSP

CSP is a combinatorial problem

- Time complexity is exponential in the number of variables/size of domains in the worst case

However, there are heuristics to make search more efficient

- Order the variables and their values
  - Select variable with the fewest “legal” values
  - Min-conflict: each variable has a tentative value that should satisfy as many constraints as possible
- Use consistency checks to prune infeasible values of domains

Structure of the constraint graph

- Tree-shaped problems can be solved in time linear in number of variables
- Strategy: try to reduce graphs to trees
  - Remove nodes in cycles to form “trees” (cut set)
  - Collapse nodes together into “strongly connected components”
Part II: Distributed Constraint Satisfaction (Dis-CSP)
Distributed CSP (Dis-CSP)

Variables/constraints are distributed among agents

- first studied by Yokoo et al. [Yokoo98]

An agent is an **autonomous** entity that

- makes “local” decisions to assign its variables
- communicates with neighboring agents via messages

\[ A = \{1, 2, 3, 4\} \]
\[ B = \{1, 2, 3, 4\} \]
\[ C = \{1, 2, 3, 4\} \]
\[ D = \{1, 2, 3, 4\} \]
Motivation behind Dis-CSP

Dis-CSP provides a framework for modeling distributed problems with constraint satisfaction characteristics

- distributed resource allocation (e.g., in a communication network)
- distributed scheduling
- sensor networks
- ...

Dis-CSP (and DCOP!) can be a suitable problem-solving approach for different reasons:

- Take advantage of knowledge that is inherently distributed among agents
- Exploit potential parallelism in constraint networks that define loosely-connected sub-problems
- A central “authority” is not practical or feasible
- Privacy/security concerns
- Robustness against failures
- ...
Naïve Approach – Synchronous Backtracking (ST)

Basic algorithm

- Order the agents based on a priority schema
  - priorities impose a total order for the agents
- “token passing” protocol
- Each agent receives a partial solution from its previous agent(s), and either
  - pass its assignment to the following agent
  - or send a “nogood” message (back to a previous agent) if no legal value is found
- The agent that receives a “nogood” message performs a backtracking step and changes its value

SB distributes the knowledge of the problem, but does not take advantage of parallelism
4-queens: Synchronous Backtracking

A = \{1, 2, 3, 4\} 
B = \{1, 2, 3, 4\} 
C = \{1, 2, 3, 4\} 
D = \{1, 2, 3, 4\}

ok? sol \{1, ?, ?, ?\}

ok sol \{1, 3, ?, ?\}

nogood

ok sol \{1, 4, 2, ?\}

ok? sol \{1, 4, 2, ?\}

D = \{1, 2, 3, 4\}
Asynchronous Backtracking (ABT)

First described in [Yokoo98]

Idea: Allow agents to run concurrently and asynchronously

- Every constraint is a directed link between a first agent that is assigned to the constraint and a second agent that receives the first agent’s value
- Agents are assigned to priorities
  - define directionality of constraints (links) in the constraint graph
  - create a fixed hierarchy among agents

\[
\begin{align*}
\text{constraint}(\text{Ag}_1, \text{Ag}_2) \\
\text{High priority (value sender)} \\
\text{notEqual}(\text{Ag}_1, \text{Ag}_2) \\
\text{Ag}_1 = A \\
\text{Ag}_2 = B \\
\text{notEqual}(A,B)\
\end{align*}
\]
4-queens: Constraint Graph with Priorities

Variables: A, B, C, D
Domain: \{1, 2, 3, 4 \}

Constraints:
for all pairs of queens
\( \rightarrow \) noThreat(Q_i, Q_j)

Priority schema:
Alphabetic order of variable identifiers

A < B < C < D
Assumptions in Dis-CSP

Additional assumptions made by Yokoo et al. [Yokoo98]

- An agent can send messages if it knows the addresses of the recipients
- Possible random delay in message transmission
- Messages are received in the order in which they are sent
- Each agent has exactly one variable (*)
- All constraints are binary (*)
- Each agent knows all constraint predicates relevant to its variable

(*) can be relaxed in some formulations, although at the expenses of efficiency.
ABT: Agent Communications

value sender

IN messages
var A ← value
my Priority = H
Assignments of neighbors
my Constraints

OUT messages
nogood (A←value)

evaluator

var B ← ...

IN messages
ok? (A←value)

OUT messages

var C ← ...

Assignments of neighbors
my Constraints

my Priority = L
ABT: Messages and Algorithm

Types of messages

- *ok?:* a value-sending agent asks whether its assignment is acceptable
- *nogood:* a constraint-evaluating agent indicates a constraint violation
- *add_link:* request to add a new link (constraint discovered while solving the problem)

Two phases:

- Each agent instantiates its variable concurrently and sends the value to agents connected by outgoing links with an *ok?* message
- Agents wait for and respond to messages
ABT: Rules for Handling Messages

1. When an agent changes its value, it sends an ok? message to its (lower-priority) neighboring agents
2. An agent changes its assignment if its current value is not consistent with the assignments of higher priority agents
3. If there’s no possible assignment that is consistent with the higher priority agents, then the agent sends a nogood message to the higher priority agents
4. After receiving a nogood message, the higher priority agent tries to change its value
Example of ABT - 1

priorities: A < B < C < D

Indicates reason for move
Example of ABT - 2


distributed Constraint Satisfaction and Optimization (for internal use) - march 2010

(c) 2010 Carnegie Mellon University

Software Engineering Institute | Carnegie Mellon

A = \{1\}
B = \{4\}
C = \{2\}
D = \{2\}

B, \{C,D\}

{}: Implicit consideration
Indicates reason for move

A = \{1\}
B = \{4\}
C = \{2\}
D = \{3\}

A,D

A = \{2\}
B = \{4\}
C = \{2\}
D = \{3\}

C

A = \{2\}
B = \{4\}
C = \{2\}
D = \{3\}

{}: Implicit consideration
Indicates reason for move
ABT: Adding New Links

A nogood message can be seen as a constraint derived from the original constraints

- by incorporating derived constraints, agents can avoid repeating the same mistake
- The full constraint network doesn't need to be specified at the beginning of the algorithm

High-priority agents X1 and X2 communicate their values to low-priority agent X3

Agent X3 cannot find a feasible value, so it asks their “superiors” to get an agreement

A new constraints between X2 and X1 is established
ABT: Detecting Termination Conditions

The agents will reach a **stable state** (if such state exists)

- All the assignments of variables to values satisfy all the constraints
- All the agents are waiting for an incoming message
- Determining whether the agents as a whole have reached a stable state is not contained in the ABT algorithm
  - a separate (distributed) termination algorithm is needed for this
Asynchronous Weak-Commitment Search (AWCS)

In ABT, if a high priority agent makes a bad value choice, the lower priority agents need to perform an exhaustive search to “revise” the bad decision.

**Intuition:** teams with flexible hierarchies (i.e., a priority schema) perform better (converge quicker to a solution) than teams with rigid hierarchies.

**Idea:** Use **min-conflict heuristic** (local search) in ABT for ordering values.

- An agent prefers the value that minimizes the number of constraint violations with variables of lower priority agents.
- Rules for allowing an agent to increment its priority at runtime:
  - Priorities are initially set to 0.
  - A priority value is changed if and only if there’s no consistent value for the agent’s value (i.e., a new *nogood* is found).
  - The new priority value is communicated to the rest of the agents.
**Example of AWCS**

A = \{1\}

B = \{1\}

C = \{1\}

D = \{1\}

**Diagram:**

- **p0:** A = \{1\}, B = \{1\}, C = \{1\}, D = \{1\}
- **p1:** A = \{1\}, B = \{4\}, C = \{4\}, D = \{3\}
- **p2:** A = \{1\}, B = \{4\}, C = \{1\}, D = \{3\}
- **p3:** A = \{2\}, B = \{4\}, C = \{1\}, D = \{3\}
Observations on ABT and AWCS

Empirical results show that AWCS performs better than ABT in practice.

Both algorithms are complete:
- **Completeness:** The algorithm finds a solution if one exists, and terminates with failure when there is no solution.
- ABT is worst-case time exponential in the number of variables, and its space complexity depends on the number of values to be recorded for a variable (i.e., polynomial).
- AWCS is also worst-case time exponential in the number of variables, but its space complexity is exponential in the number of variables.
  - In practice, the number of *nogoods* recorded by each agent can be limited to a fixed number (~10). If so, space complexity is improved but the theoretical completeness cannot be guaranteed.
Extension: Handling Multiple Local Variables

Method 1: Each agent finds all solutions to its local problem

- The problem can be re-formalized as a Dis-CSP in which each agent has one local variable whose domain is the cross product of the domains of the local solutions
  - when a local problem is large/complex, finding all solutions is difficult

Method 2: An agent creates multiple virtual agents

- Each virtual agent corresponds to one local variable
- The concurrent activities of these virtual agents are simulated by the parent agent
  - simulating concurrent activities at the local level can be expensive

Both methods are neither efficient nor scalable to larger problems
Part III: Distributed Constraint Optimization (DCOP)
Distributed Constraint Optimization (DCOP)

set of **variables** $X_1, X_2, \ldots, X_n$
each variable has a non-empty **domain** $D(X_i)$ of possible **values**
set of **valued constraints** $C_1, C_2, \ldots, C_m$
Constraints quantify a “degree of satisfaction” (soft constraints)
each constraint maps any instantiation of its variables to a real number (cost or utility)
Solving DCOP is more than “Satisfaction”

\[ F = C_1 + C_2 + \ldots + C_m = 2 + 2 + 1 + 3 = 8 \]

A solution is an assignment of all the variables to values. Finding a solution still involves search, but the goal is not just to find any solution.

**Goal**: find the best solution such that the sum of the constraint costs is minimized (or the sum of the utilities is maximized) → objective function
DFS Tree Ordering of the Constraint Graph

In D-COP algorithms, the agents need to be prioritized in a depth-first search (DFS) tree, such that

- Each agent has a single parent and multiple children
- Constraints are only allowed between agents that are in an ancestor-descendant relationship in the tree
- An agent has only information about ancestors it shares a constraint with

The purpose of the DFS tree is to decompose the global cost function

- Given the assignments to all ancestors,
- an agent in a given sub-tree can work on minimizing its part of the solution
- this work can be done independently of agents in other sub-trees
Example of DFS Ordering

Any constraint graph can be ordered into some DFS tree using a distributed algorithm.

- finding the best DFS ordering is NP though
DCOP: Requirements and Issues

1. Agents need to optimize a global objective function in a distributed fashion using local communication (neighbors)
2. Allow agents to operate asynchronously (as in Dis-CSP)
3. **Quality guarantees**
   - A method to find provably optimal solutions (cost/utility) when possible

**Main challenge:** combining items 2) & 3)

Building consensus among agents

- **Top-down:** higher-rank agents decide first, lower-rank agents have to find ways to comply with higher-rank agents
- **Bottom-up:** agents build solutions starting with small pieces (produced by lower-rank agents) that get bigger and bigger as the reach higher-rank agents

(A tree structure that organizes the agents is assumed)
Two Search Strategies for a DFS Tree

Backtracking (top-down)
- Assumes an ordering of the variables/agents
- Control shifts between different agents during selection of values
- Requires little memory
- Exponential number of messages - with linear message size
- e.g., ADOPT [Modi05], OptAPO [Mailler04]

Dynamic programming (bottom-up)
- Assumes an ordering of the variables/agents
- Agents incrementally compute all partial solutions (i.e., all possible values), when the solutions are complete, agents pick the best solution
- Requires more memory
- Linear number of messages - but with exponential message size (vectors)
- e.g., RTree, DPOP [Petcu04] and variants
Backtracking Skeleton

1. Receive value message from ancestor agents (if not root)
2. Choose a feasible value for my variable
3. Inform value choice to children agents (if not leaf)
4. Wait for cost messages from children (if not leaf)
   - If costs are not good enough, ask children to change their values
   - If children cannot find good values, change current value and go to 3)
5. Combine costs and report my aggregated cost to parent
   - cost received from children (if not leaf)
   - plus cost of my constraints with the ancestor agents
6. Wait for further messages from ancestors
7. Go to 1)
...
8. At some point, root agent decides that the optimum has been found or that there’s no solution, and stops the algorithm
Example of Values/Costs in Top-down Search

Domain = \{a, b, c\}

(*) Each agent can perform backtracking (pick a different value) if the costs reported by its children are too high.
Dynamic Programming Skeleton

1. Leaf agent: Send utility vector for its possible values to parent agent
2. Intermediate agent: Receive utility vectors from children
   - Combine these utilities with my own utility table (based on constraints with my parent) → construct intermediate utility table
   - Send aggregated utility vector to parent agent
3. Root agent: Receive all the utility vectors and construct a global utility table
   - Pick my optimal value from the global utility table
   - Inform my value choice to children agents
4. Intermediate agent:
   - Pick my (local) optimal value, based on the value message from my parent and my intermediate utility table
   - Inform my value choice to children agents
5. Leaf agent: Pick my (local) optimal value and stop

Agent doesn’t know the value its parent will pick
Example of Costs/Values in Bottom-up Search

- **root**
  - value-msg\(_{\text{root}}\) = a
  - costs-msg\(_{\text{int1}}\) = o ([5,7,12]. ...) = [8, 10, 23]

- **int1**
  - value-msg\(_{\text{int1}}\) = b
  - costs-msg\(_{\text{int2}}\) = o ([2, 6, 2], [4, 3, 1], ...) = [5, 7, 12]
  - costs-msg\(_{\text{leaf2}}\) = [2, 6, 2]
  - value-msg\(_{\text{int2}}\) = b

- **leaf1**
  - value\(_{\text{leaf1}}\) = c

- **leaf2**
  - value\(_{\text{leaf2}}\) = c
  - costs-msg\(_{\text{leaf2}}\) = [4, 3, 1]

**Domain** = \{a, b, c\}
Map of Algorithms - DCOP

CSP

backtracking
local search
...

DIS-CSP

ABT
AWCS
Distributed
breakout

DCOP

branch &
bound

OptAPO

(partial centralization
+backtracking)

DPOP Family

PC-DPOP

ADOPT

DTree
M-POP
A-POP
...

NCBB
Asynchronous Distributed Optimization (ADOPT)

Developed by Modi et al. [Modi05],

- Combination of ideas from ABT (satisfaction) and Branch & Bound (optimization)

Previous approaches backtrack only when sub-optimality is proven

- **Branch and bound**: It backtracks when cost exceeds upper bound
  - Limitations
    - sequential,
    - synchronous,
    - computing cost upper bounds requires global information

- **Asynchronous backtracking**: It backtracks when a constraint is unsatisfiable
  - Limitations
    - only “hard constraints” are allowed
Relaxing Backtracking

A root agent aggregates global costs

**Weak backtracking**: Opportunistic best-first search

- Agents can go ahead and make decisions based on local information
- **Cost lower bounds** of solutions are suitable for asynchronous search
  - an initial lower bound is computable based on local information
  - each agent chooses an assignment with smallest lower bound
- ADOPT backtracks when a lower bound gets too high
  - instead of when quality of best solution of sub-problem is determined
ADOPT: Problem Formulation

Given

- Variables \{x_1, x_2, \ldots, x_n\}, each assigned to an agent
- Finite, discrete domains D_1, D_2, \ldots, D_n
- For each \(x_i, x_j\), valued constraint \(f_{ij}: D_i \times D_j \to \mathbb{N}\) (binary constraint)

Goal: Find complete assignment \(A\) that minimizes \(F(A)\) where

\[
F(A) = \sum f_{ij}(d_i,d_j), \quad x_i \leftarrow d_i, \; x_j \leftarrow d_j \text{ in } A
\]
ADOPT: Assumptions

Aggregation operator
- The sum of the constraints is associative, commutative and monotonic
- **Monotonicity** requires that the cost of a solution can only increase as more costs are aggregated

Constraints are (at most) binary
- This can be extended to non-binary constraints

Each agent is assigned to a single variable
- This can be extended to several variables

Preprocessing
- At the beginning, all the agents must be arranged in depth-first-search (DFS) **tree structure**
**ADOPT: Algorithm**

Agents are ordered in a tree
- constraints happen between ancestors/descendants
- no constraints between siblings

**Basic algorithm:**
- choose value with minimum cost
- loop until termination-condition true:
  - when receive message:
    - choose value with min cost
    - send **VALUE** message to descendants (like *ok?* in ABT)
    - send **COST** message to parent (like *nogood* in ABT)
    - send **THRESHOLD** message to child

Adapted from slides [Modi05], Agents@USC
Example of ADOPT

- Concurrently choose value, send value to descendents
- Report lower bounds (costs)
- x1 switches value to “white” and propagates it
- X2 propagates its new value

**Constraint Graph**

<table>
<thead>
<tr>
<th>$d_i$</th>
<th>$d_j$</th>
<th>$f(d_i,d_j)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>● ●</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>● ●</td>
<td>●</td>
<td>2</td>
</tr>
<tr>
<td>● ●</td>
<td>●</td>
<td>2</td>
</tr>
<tr>
<td>● ●</td>
<td>○</td>
<td>0</td>
</tr>
</tbody>
</table>

Adapted from slides [Modi05], Agents@USC
Key Ideas in ADOPT

Opportunistic best-first search

• No global information is required, only local interactions
• Allow each agent to change a variable value whenever it detects there is a possibility that some other solution may be better than the current under investigation
  – this doesn’t guarantee the new value to be better though
• Each agent picks the value with smallest lower bound

Backtrack threshold

• When an agent has to revisit a previous solution, it uses a stored lower bound to increase efficiency
• A variable/agent doesn’t change its assignment as long as its total cost (i.e., lower bound) is less that the backtrack allowance

Bounded error approximation

• Sort of “quality control” for approximate solutions
Weak Backtracking

Suppose a domain with 2 possible values: “white” and “black”

Explore “white” first

Receive cost message

Now explore “black”

Receive cost message

Go back to “white”

Termination Condition True

Adapted from slides [Modi05], Agents@USC
Computing Lower/Upper Bounds for Costs

For a given agent/variable $X_i$

- $X_i$ calculates cost as local cost plus any cost feedback received from its children
- $\delta(\text{value})$ = sum of the costs from constraints between $X_i$ and higher neighbors
- $\text{LB}(\text{value})$ = lower bound for the subtree rooted at $X_i$, when $X_i$ chooses the value
- $\text{UB}(\text{value})$ = upper bound for the subtree rooted at $X_i$, when $X_i$ chooses the value

The lower bound (LB) for $X_i$ is the minimum $\text{LB}(\text{value})$ over all value choices for $X_i$

The upper bound (UB) for $X_i$ is the minimum $\text{UB}(\text{value})$ over all value choices for $X_i$
Backtracking Threshold – Parent

Parent informs children nodes about lower bound (for search)

Explore “white” first

Now explore “black”

Return to “white

The parent knows (from previous experience) that cost >= 10
So, it informs its children not to bother searching for solutions whose cost is less than a threshold of 10

Adapted from slides [Modi05], Agents@USC
Backtracking Threshold – Child

Suppose Agent x received threshold = 10 from its parent

Explore “white” first

Receive cost message

Stick with “white”

Receive more cost messages

Now try “black”

Key point:
Agent doesn’t change value until
LB(current value) > threshold

Adapted from slides [Modi05], Agents@USC
ADOPT: Termination Condition

Termination is a built-in mechanism in the algorithm

- Bound intervals for tracking progress towards the optimal solution
- When the size of the bound interval shrinks to zero, the cost of the optimal solution has been determined and agents can safely terminate
- Intervals are also used for bound-error approximation

Caveat

- Some centralization is needed: a root agent aggregates global costs and detects termination

Termination condition (cost = 10)

$\text{LowerBound}(w) = 10 = \text{UpperBound}(w)$

$\text{LowerBound}(b) = 12$
ADOPT: Bounded Error Approximation

Generate solutions whose quality is within a user-specified distance from the optimal solution

- It usually takes less time than required to find the optimum
- Based on lower bound kept during search

**Example:** If an optimal solution to an over-constrained graph coloring requires violating 3 constraints, \( b = 5 \) indicates that 8 constraints is an acceptable solution for the user.

If user provides error bound \( b \)

Find any solution \( S \) where

\[
\text{cost}(S) \leq \text{cost(optimal solution)} + b
\]

Adapted from slides [Modi05], Agents@USC
Observations on ADOPT

Algorithm is proven to be sound and complete

Worst-case time complexity is exponential in the number of variables
  • but it only requires polynomial space at each agent
  • Experiments show that sparse graphs can be solved optimally and efficiently (communication messages grow linearly in low density graphs)
Map of Algorithms - DCOP

CSP

backtracking
local search
...

DIS-CSP

ABT
AWCS
Distributed
breakout

DCOP

branch &
bound

OptAPO
(partial centralization
+backtracking)

DPOP Family

ADOPT

NCBB

PC-DPOP

DTree
M-POP
A-POP
...

Distributed Constraint Satisfaction and Optimization (for internal use) - March 2010
© 2010 Carnegie Mellon University
NCBB: No Commitment Branch and Bound

Improvement over ADOPT, developed by [Chechetka06]

Speed up the search by exploiting the distributed aspect of the problem
- have different agents exploring non-intersecting regions of the search space simultaneously

Ideas: Reduce synchronization overhead and prune the search space faster
- A (new) Search message allows each agent to search different sub-trees than its siblings
- Computation of tighter upper bounds on solution
- Eager propagation of changes in cost lower bounds

NCBB performance is better than ADOPT, and memory is still polynomial
Map of Algorithms - DCOP

- **CSP**
  - backtracking
  - local search
  - ...

- **DIS-CSP**
  - **ABT**
  - **AWCS**
  - Distributed
  - breakout

- **DCOP**

- **OptAPO**
  - (partial centralization + backtracking)

- **DPOP Family**
  - **DTree**
  - **M-POP**
  - **A-POP**
  - ...

- **ADOPT**

- **NCBB**
Distributed Tree-shaped Networks (DTree)

DTree algorithm described in [Petcu04]

Ideas:
• Use a dynamic-programming style of exploration (utility vectors)
• Tree-shaped constraint networks (no cycles, yet)

Algorithm has two phases (assuming a tree structure)

1. UTIL propagation from the leaves of the tree all the way up to the root
   – Each utility message summarizes the optimal values that can be achieved by the sub-tree rooted at the agent’s node for each domain value of the parent agent

2. VALUE propagation from the root of the tree downwards to the leaves
   – Once all the utility messages are received from the neighboring agents, the parent agent can choose the optimal value
DTree: K-1 Propagation Rule

If node (agent) Xi has k neighbors

- Xi will send out a message to its k\textsuperscript{th} neighbor only after having received the other k-1 messages
- Xi will send out the rest of the m-1 messages after having received the messages from the k\textsuperscript{th} neighbor

Each node knows whether it is a leaf or not

utility propagation

k neighbors for X1
DTree: Example of Flow of Messages

(a) Simple DCOP with 4 agents, 3 relations
(b) Relations of $X_2$ and $X_3$ with $X_1$
(c) Messages to $X_1$ and join with relation with $X_0$
(d) final result
From slides [Petcu04], IJCAI’05
DTree: Utility Computation Example - 2

From slides [Petcu04], IJCAI’05
DTree: Utility Computation Example - 3

From slides [Petcu04], IJCAI’05
Distributed Pseudo-tree Optimization (DPOP)

Application of DTREE phases to arbitrary constraint topologies
- Constraint graphs can have cycles (agents other than parent/children)
- Dynamic programming technique is still useful with some modifications

Ideas: Transform graph into a pseudo-tree by traversing it in DFS order
- Phase 0: Topology probing to fix the pseudo-tree arrangement
- Break the problem into cycle-free parts → choice of cycle cutset
- Utility messages are now “multi-dimensional”

Message size is still exponential, but reduced from $\text{dom}^n$ to $\text{dom}^k$
- where $n = \#\text{nodes}$ and $k = \#\text{induced width}$ of the DFS (tree) arrangement
- the more a problem has a tree-like structure, the lower its induced width
- $n >> k$ for large and loosely coupled problems
DPOP Variants – Efficiency

Approximations on solution quality
- Tradeoff between solution quality and computational effort
- The user can specify a maximal error bound on the solution
- **A-DPOP**: Approximation of the solution by adapting the message size

Local search
- Start with some (random) solution and gradually improve it
- **LS-DPOP**: Nodes make decisions based only on local information
- Applicable in large neighborhoods

Partial centralization
- Use mediation (like in OptAPO) for tightly connected clusters
- Each mediator can take advantage of an efficient CSP solver
- **PC-DPOP**: Identify difficult sub-problems and centralize them in relevant nodes of the problem

Time-space tradeoff
- **MB-DPOP**: memory bound extension
DPOP Variants – Dynamic Problems

Self-stabilization property: Given enough time between changes/failures, the algorithm will converge to the optimal D-COP solution, and then will maintain that state

- Changes in topology
- Changes in the valuations of the agents
- Temporary communication problems

DPOP extensions: Apply self-stabilization protocols to DPOP phases

- SS-DPOP: based on self-stabilizing algorithm introduced by Kollin et al.
- RS-DPOP: continuous-time problem by reacting to problem changes and generating new solutions with low costs
  - DynDCOP: DCOP formulation extended with “stability constraints” and “commitment deadlines”
Thoughts: Hybrid Algorithms

In practice, D-COP algorithms have to make different tradeoffs among:

- Message size, number of messages, memory usage
- Degree of distribution (mediation vs. full decentralization)
- Solution quality (complete vs. incomplete algorithms)
- Adaptation to changes in (dynamic) environments
- Agents that might be not cooperative
- Privacy concerns when the agents reveal information

How to deal with non-binary constraints and local variables efficiently?
How to model “social choice” or “agent preferences” using D-COP?

Incomplete algorithms can be a scalable option for agents that form small groups and optimize within these groups.
References - 1


References - 2


Some Implementations

Choco (CSP in Java)
ADOPT (Java)
Frodo (DPOP platform in Java)