AAMAS-19 Tutorial on:

MULTI-AGENT DISTRIBUTED CONSTRAINED OPTIMIZATION

Hands on pyDCOP

Pierre Rust Gauthier Picard

Menu

Menu

Hands on PyDCOP I

- Install VirtualBox
- Import the pyDCOP Virtual Machine (`http://bit.ly/pyDCOP`)
  - It's a Debian image with everything preinstalled:
  - python3, pyDCOP, matplotlib, glpk, etc.
- Alternatively, follow
  `https://pydcop.readthedocs.io/en/latest/installation.html`

1. `https://pydcop.readthedocs.io/en/latest/tutorials/getting_started.html`
2. `https://pydcop.readthedocs.io/en/latest/tutorials/analysing_results.html`

Hands on PyDCOP I

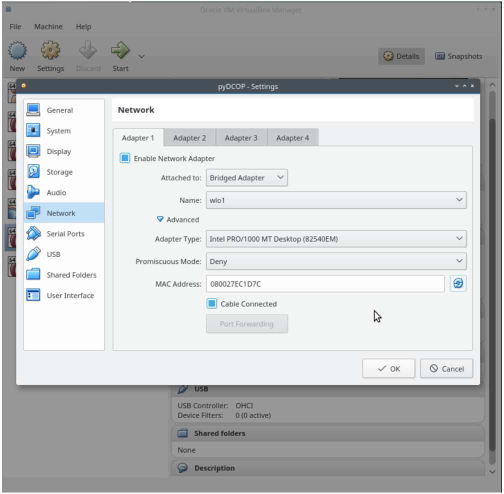
Virtual machine Setup

Before starting the VM:

- "Bridged adapter" mode
- Select wifi network adapter
- Reset MAC Address

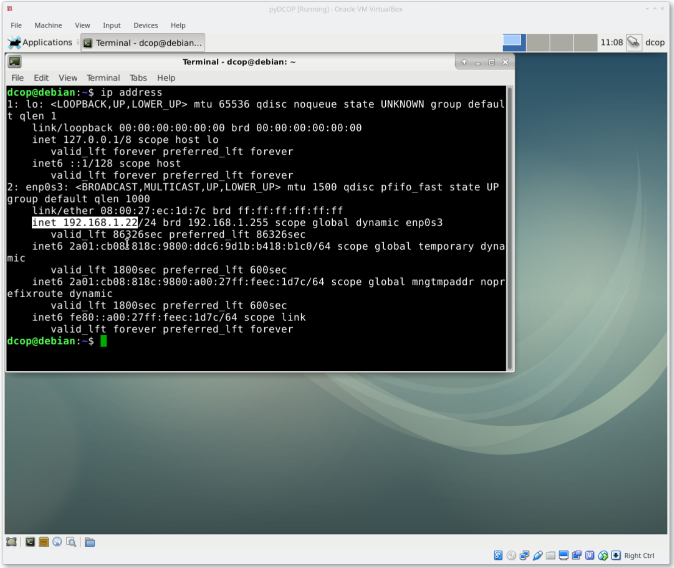
Then

- Start the VM
- login: `dcop / pyDCOP`
- Launch a terminal
- Note down the IP with `ip address`



Hands on PyDCOP I

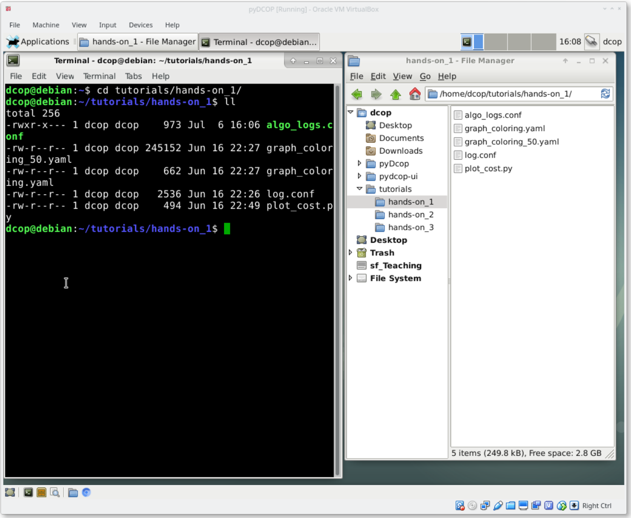
Virtual machine Setup



Hands on PyDCOP I

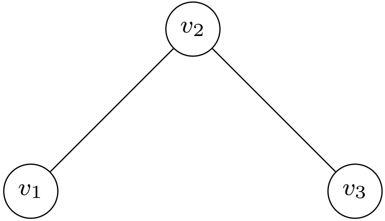
Files for the tutorials are in `/home/dcop/tutorials`.

```
$ cd /home/dcop/tutorials/hands-on_1
```

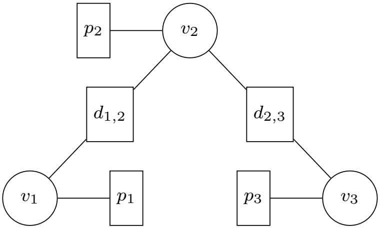


Hands on PyDCOP I

DCOP - Graph Coloring



(a) constraints graph



(b) factor graph

- **Objective:** minimize
- **Domain:** 2 colors R and B
- **Variables:** V_1, V_2, V_3
- **Constraints:** neighbors must have different colors + preferences
- **Agents:** 3 agents

Yaml representation

Hands on PyDCOP I

pyDCOP yaml format

graph_coloring.yaml

```yaml
name: graph coloring
objective: min

domains:
  colors:
    values: [R, G]

variables:
  v1:
    domain: colors
  v2:
    domain: colors
  v3:
    domain: colors
```

```yaml
constraints:
    pref_1:
      type: extensional
      variables: v1
      values:
        -0.1: R
        0.1: G

    pref_2:
      type: extensional
      variables: v2
      values:
        -0.1: G
        0.1: R

    pref_3:
      type: extensional
      variables: v3
      values:
        -0.1: G
        0.1: R

    diff_1_2:
      type: intention
      function: 10 if v1 == v2 else 0

    diff_2_3:
      type: intention
      function: 10 if v3 == v2 else 0

agents: [a1, a2, a3, a4, a5]
```

Hands on PyDCOP I

Solving the Graph Coloring DCOP

Command:

```
$ pydcop solve --algo dpop graph_coloring.yaml
```

Output:

```
 ...
"assignment": {
   "v1": "R",
   "v2": "G",
   "v3": "R"
 },
 "cost": -0.1,
 ...
```

With other algorithms:

```
$ pydcop --timeout 2 solve --algo dsa graph_coloring.yaml
$ pydcop solve --algo mgm --algo_params stop_cycle:20 \
        graph_coloring.yaml
```

Hands on PyDCOP I

Results

Full results :


```
{
  "agt_metrics": {
    ...
  },
  "assignment": {
    "v1": "R",
    "v2": "G",
    "v3": "R"
  },
  "cost": -0.1,
  "cycle": 20,
  "msg_count": 158,
  "msg_size": 158,
  "status": "FINISHED",
  "time": 0.03201029699994251,
  "violation": 0
}
```


Look at results from mgm and dsa, compared to dpop's results !

Hands on PyDCOP I

Logs

Simple:

use `-v 0..3`

```
$ pydcop -v 3 solve --algo dsa --algo_params stop_cycle:20 graph_coloring.
    yaml
```

Precise :

use `-log <log.conf>`

```
$ pydcop --log log.conf solve --algo dsa --algo_params stop_cycle:10
    graph_coloring.yaml
```

Now, look at `algo.log`

Hands on PyDCOP I

Run-time metrics

periodic: "--collect_on period --period <p>"

```
$ pydcop --log log.conf -t 10 solve \
   --collect_on period --period 1 --run_metric ./metrics.csv \
   --algo dsa graph_coloring.yaml
```

cycle: "--collect_on cycle_change"

Only supported with synchronous algorithms !

```
$ pydcop solve --algo mgm --algo_params stop_cycle:20 \
    --collect_on cycle_change --run_metric ./metrics.csv \
    graph_coloring_50.yaml
```

value: "--collect_on value_change"

```
$ pydcop -t 5 solve --algo mgm --collect_on value_change \
    --run_metric ./metrics_on_value.csv \
    graph_coloring_50.yaml
```

Hands on PyDCOP I

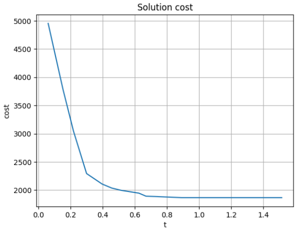
Run-time metrics

With a bigger graph coloring problem

```
$ pydcop solve --algo mgm --algo_params stop_cycle:20 \
            --collect_on cycle_change \
            --run_metric ./metrics.csv \
            graph_coloring_50.yaml
```

Plotting with matplotlib

```
$ python3 plot_cost.py ./metrics.csv
```

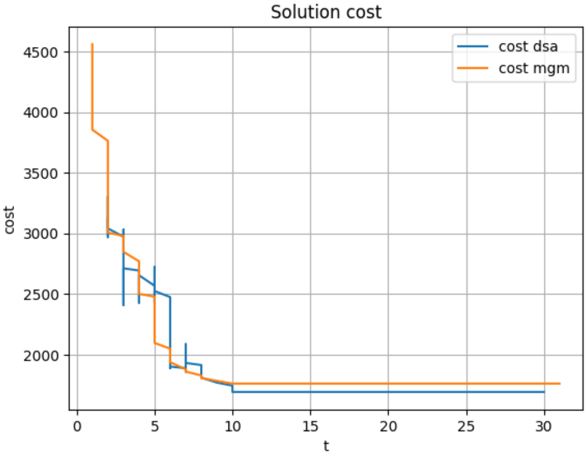


Do the same thing with DSA, look at the result, what do you see ?

Hands on PyDCOP I

Run-time metrics

MGM (1720) and DSA (1647) , both with 30 cycles



Hands on PyDCOP I

Web-ui

Web-base agent graphical interface:

- Run the web application

```
$ cd ~/pydcop-ui
$ python3 -m http.server
```

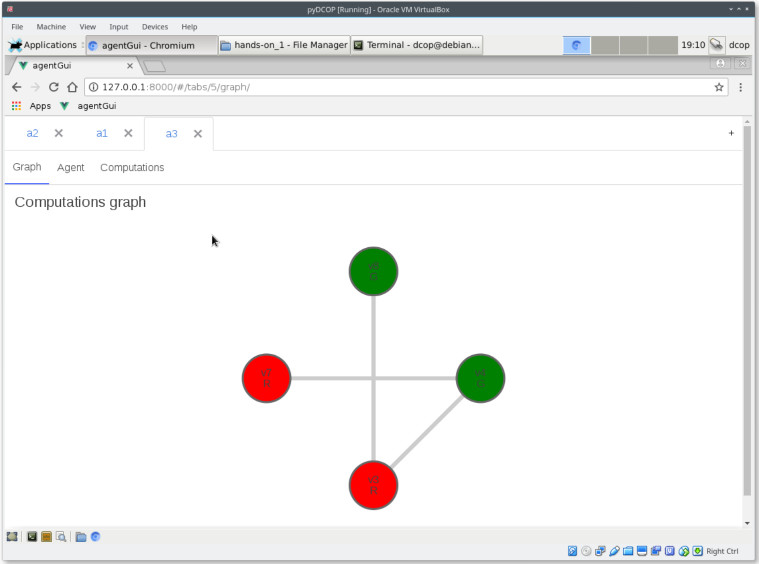
- Launch a browser on `http://127.0.0.1:8000`
- Solve the dcop with the option `--uiport <port>` (also, use `--delay <delay>`)

```
$ pydcop -v 3 solve -a mgm -d adhoc --delay 2 --uiport 10000
./graph_coloring_3agts_10vars.yaml
```

- Each agent exposes a web-socket, the web application connects to these websockets and display the agents' state.

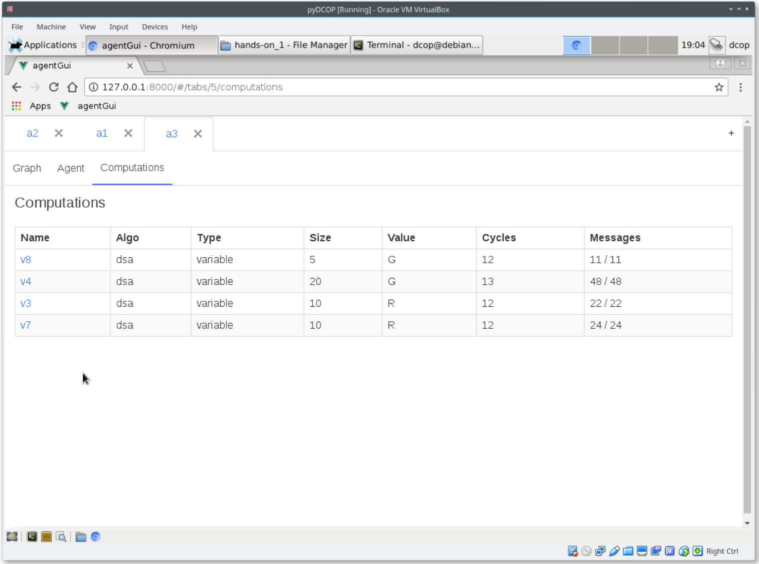
Hands on PyDCOP I

Web-ui



Hands on PyDCOP I

Web-ui



Menu

Hands on PyDCOP II

Developping with pyDCOP

pyDCOP is designed to make it easy to implement new DCOP algorithms

- All the infrastructure is provided:
  - agents,
  - messaging,
  - metrics,
  - etc.
- Base classes and utility functions for
  - constraints,
  - variables,
  - domains,
  - etc.
- Plugin mechanism to define new algorithms for DCOP, distribution and replication.

Implementing a DCOP algorithm with pyDCOP

Simple DSA implementation

Create a new python module in pydcop.algorithms

- Define a constant indicating the graphical representation used by your algorithm :
  GRAPH_TYPE = 'constraints_hypergraph'
- Define your message(s):
  message_type(<name>, [fields]);
- Subclass VariableComputation:
  - Annotate your message handler(s) with @register(<name>)
  - Send messages to your neighbors using self.post_msg or self.post_to_all_neighbors
  - Select a new value with self.value_selection
  - Start a new cycle with self.new_cycle()

Hands on PyDCOP II

Simple DSA implementation

One class, 3 main methods:

```
GRAPH_TYPE = 'constraints_hypergraph'
algo_params = []

DsaMessage = message_type("dsa_value", ["value"])

class DsaTutoComputation(VariableComputation):

    def __init__(self, computation_definition):
        ...

    def on_start(self):
        ...

    @register("dsa_value")
    def on_value_msg(self, variable_name, recv_msg, t):
        ...

```

Hands on PyDCOP II

Simple DSA implementation

Creating the computation instance.
a `computation_definition` contains: constraints, variable, neighbors, parameters, etc.

```
class DsaTutoComputation(VariableComputation):

    def __init__(self, variable, constraints, computation_definition):
        super().__init__(computation_definition.node.variable,
                         computation_definition)


        self.constraints = computation_definition.node.constraints
        self.current_cycle = {}
        self.next_cycle = {}

```

Hands on PyDCOP II
Simple DSA implementation

On startup, select a value at random and send it to all neighbors.

```
    def on_start(self):
        self.random_value_selection()
        self.post_to_all_neighbors(DsaMessage(self.current_value))
        self.evaluate_cycle()

```

Hands on PyDCOP II

Simple DSA implementation

Receiving values from our neighbors
DSA is a synchronous algorithm !

```python
    @register("dsa_value")
    def on_value_msg(self, variable_name, recv_msg, t):

        if variable_name not in self.current_cycle:
            self.current_cycle[variable_name] = recv_msg.value
            self.evaluate_cycle()

        else:  # The message is for the next cycle
            self.next_cycle[variable_name] = recv_msg.value


```

Hands on PyDCOP II

Simple DSA implementation

The actual DSA implementation: select a new value if the gain is positive.

```python
def evaluate_cycle(self):

    self.current_cycle[self.variable.name] = self.current_value
    current_cost = assignment_cost(self.current_cycle, self.constraints)
    arg_min, min_cost = self.compute_best_value()

    if current_cost - min_cost > 0 and 0.5 > random.random():
        self.value_selection(arg_min)

    self.new_cycle()
    self.current_cycle, self.next_cycle = self.next_cycle, {}
    self.post_to_all_neighbors(DsaMessage(self.current_value))

```

Hands on PyDCOP II

Simple DSA implementation

Last utility method: find the best value given the value from the neighbors.

```python
def compute_best_value(self):
    arg_min, min_cost = None, float('inf')
    for value in self.variable.domain:
        self.current_cycle[self.variable.name] = value
        cost = assignment_cost(self.current_cycle, self.constraints)
        if cost < min_cost:
            min_cost, arg_min = cost, value
    return arg_min, min_cost
```

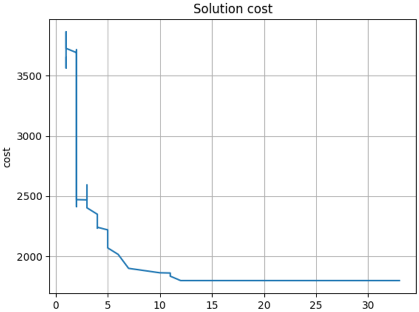
Hands on PyDCOP II

Simple DSA implementation

We can now use this new algorithm directly through the command line interface (except for `stop_cycle:20`):

```
$ pydcop --log log.conf -t 20 solve --algo dsatuto \
            --collect_on value_change \
            --run_metric ./metrics_tuto.csv \
            graph_coloring_50.yaml
```

Of course, it also works with the metrics, web-ui, etc.



Menu

SECP model

Smart Environment Configuration Problem [Rust et al., 2016]

- Example of applying DCOPs to a "real" problem
- Coordinate objects in the building
- Model
  - objects
  - relations between objects and environment
  - user objectives and requirements
- Formulate the problem as an optimization problem

SECP model

Smart Environment Configuration Problem [Rust et al., 2016]

Focus on smart lighting use cases

- **Objects:** anything that can produce light: light bulbs, windows with rolling shutter, etc.
- **User preferences:** having a predefined luminosity level in a room, under some conditions
- **Energy efficiency**

Linking objects and user preferences:

- How to model the luminosity in a room ? **variable**
- How to model the dependency between the light sources and the luminosity ? **function / constraint**

SECP model

Example application to ambient intelligence scenario

- **Actuators**
  - Connected light bulbs, TV, Rolling shutters, ...
- **Sensors**
  - Presence detector, Luminosity Sensor, etc.
- **Physical Dependency Models**
  - E.g. Living-room light model
- **User Preferences**
  - Expressed as rules :

$$\begin{array}{lllr}
\texttt{IF} & \texttt{presence_living_room} & = & 1 \\
\texttt{AND} & \texttt{light_sensor_living_room} & < & 60 \\
\texttt{THEN} & \texttt{light_level_living_room} & \leftarrow & 60 \\
\texttt{AND} & \texttt{shutter_living_room} & \leftarrow & 0
\end{array}$$

SECP model

Example application to ambient intelligence scenario

- **Actuators**
  - *Decision* variable x_i, domain $\mathcal{D}_{x_i}$
  - Cost function $c_i : \mathcal{D}_{x_i} \rightarrow \mathbb{R}$
- **Sensors**
  - *Read-only* variable s_l, domain $\mathcal{D}_{s_l}$
- **Physical Dependency Models** $\langle y_j, \phi_j \rangle$
  - Give the expected state of the environment from a set of actuator-variables influencing this model
  - Variable y_j representing the *expected* state of the environment
  - Function $\phi_j : \prod_{\varsigma \in \sigma(\phi_j)} \mathcal{D}_\varsigma \rightarrow \mathcal{D}_{y_j}$
- **User Preferences**
  - Utility function u_k
  - Distance from the current expected state to the target state of the environment

Formulating SECP as a DCOP

Multi-objective optimization problem

$$\min_{x_i \in \nu(\mathfrak{A})} \sum_{i \in \mathfrak{A}} c_i \quad \text{and} \quad \max_{\substack{x_i \in \nu(\mathfrak{A}) \\ y_j \in \nu(\Phi)}} \sum_{k \in \mathfrak{R}} u_k$$

$$\textbf{s.t.} \;\; \phi_j(x_j^1, \ldots, x_j^{\overline{\phi_j}}) = y_j \quad \forall y_j \in \nu(\Phi)$$

Then mono-objective DCOP formulation

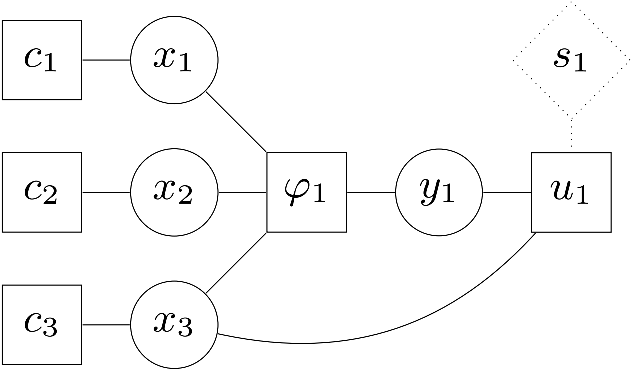
$$\max_{\substack{x_i \in \nu(\mathfrak{A}) \\ y_j \in \nu(\Phi)}} \quad \omega_u \sum_{k \in \mathfrak{R}} u_k - \omega_c \sum_{i \in \mathfrak{A}} c_i + \sum_{\varphi_j \in \boldsymbol{\Phi}} \varphi_j$$

with reformulation of hard constraints ϕ_j into soft ones:

$$\varphi_j(x_j^1, \ldots, x_j^{|\sigma(\phi_j)|}, y_j) = \begin{cases} 0 & \text{if } \phi_j(x_j^1, \ldots, x_j^{|\sigma(\phi_j)|}) = y_j \\ -\infty & \text{otherwise} \end{cases}$$

Formulating SECP as a DCOP

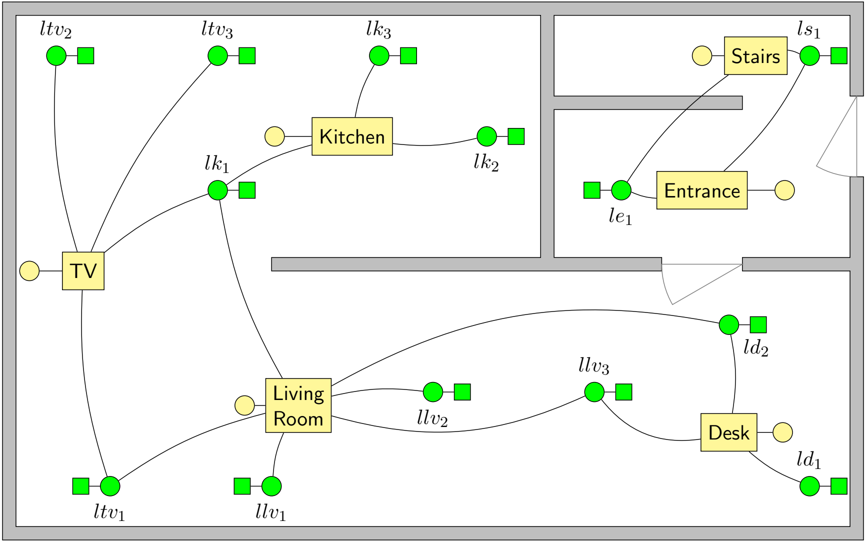
Representing a DCOP as a factor graph



Factor graph: Bipartite graph with nodes for variables **and constraints**

SECP Factor Graph

in a house (without rules)



Menu

Distribution of computations

Allocating computations to agents

- DCOP: $\langle \mathcal{A}, \mathcal{X}, \mathcal{D}, \mathcal{C}, \mu \rangle$
- μ : function mapping variables to their associated agent

Why is distribution needed ?

Common assumptions:

- computation $\equiv$ variable
- each agent controls exactly one variable (bijection)
- binary constraints

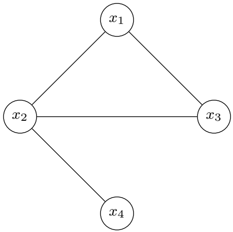
Real distributed problems:

- agents must be hosted on real devices
- the set of devices might be given by the problem
- for some variables the relation with an agent is obvious, but not always

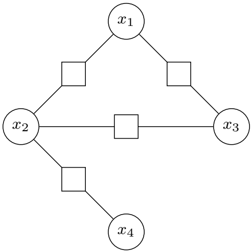
Distribution of computations

Several graph representations for the same DCOP.

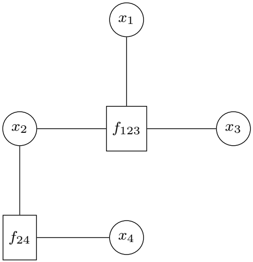
- Nodes in the graph = computations



(a) Simple constraint graph



(b) Factor graph

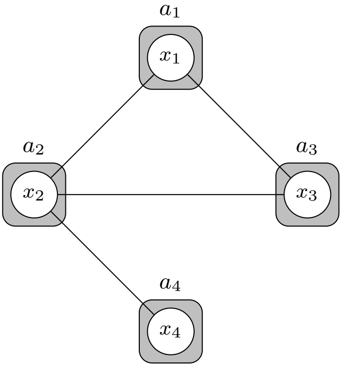


(c) Factor graph

Distribution computations

Computations

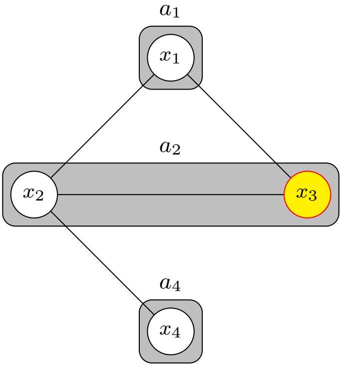
- **belong** to an agent :
  "natural" link,
  problem characteristics



Distribution computations

Computations

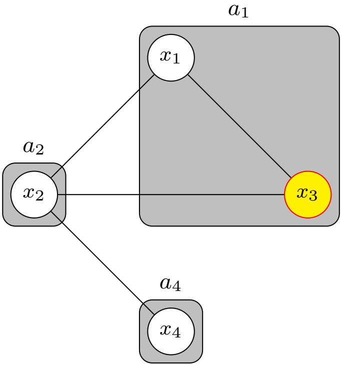
- **belong** to an agent
- **shared** decisions :
  modeling artifact, with no obvious agent relation (e.g. distributed meeting scheduling, SECP, etc.)



Distribution computations

Computations

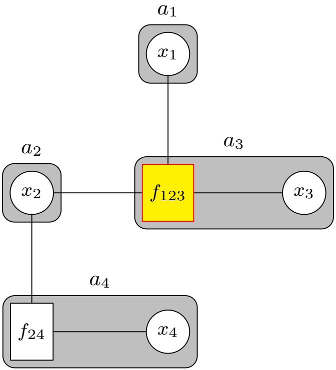
- **belong** to an agent
- **shared** decisions :
  modeling artifact, with no obvious agent relation (e.g. distributed meeting scheduling, SECP, etc.)



Distribution computations

Computations

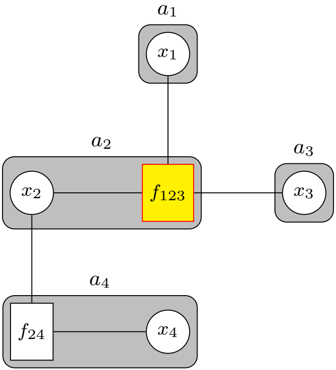
- **belong** to an agent
- **shared** decisions
- **factors**, in a factor graph:
  not representing a decision variable



Distribution computations

Computations

- **belong** to an agent
- **shared** decisions
- **factors**, in a factor graph:
  not representing a decision variable



Distribution of computations

Allocating computations to agents

- Distributing computations
  - computations depends on the graph model used by the algorithm
  - variables and / or factors

- Distribution impacts the system characteristics
  - speed,
  - communication load,
  - hosting costs, etc.
- Computing a distribution
  - heuristics
  - optimal ?

Distribution of computations
Optimal definition

Optimal distribution ?

- Problem dependent
- Optimization problem :
  find the best distribution, for your problem's criteria
- Optimal distribution $\equiv$ graph partitioning,
  NP-hard in general [Boulle, 2004]

Distribution of computations
Better definition

SECP distribution problem

- Devices have limited memory
- Communication is expensive and has limited bandwidth
- Variable related to an actuator are hosted by it
- Objective : **minimize overall communication between agents**

Optimization problem : define an ILP for it !

Binary ILP for computation distribution

- x_i^k , binary variables that map computations to agents and $\alpha_{ij}^{mn} = x_i^m \cdot f_j^n$

$$\forall x_i \in X, \quad \sum_{a_m \in \mathbf{A}} x_i^m = 1 \tag{1}$$

- Message's size between variable x_i and factor f_j: $msg(i, j)$

$$\underset{x_i^m}{\textsf{minimize}} \quad \sum_{(i,j) \in D} \sum_{(m,n) \in \mathbf{A}^2} \mathbf{msg}(i, j) \cdot \alpha_{ij}^{mn} \tag{2}$$

- Memory footprint of a computation: $\mathbf{weight}(e)$, and memory capacity for a device: $\mathbf{cap}(a_k)$

$$\forall a_m \in \mathbf{A}, \quad \sum_{x_i \in D} \mathbf{weight}(x_i) \cdot x_i^m \leq \mathbf{cap}(a_m) \tag{3}$$

- and a few linearization constraints

Binary ILP for computation distribution

More generic case:

- Add route cost: $\mathbf{com}(i, j, m, n)$

$$\forall x_i, x_j \in \mathbf{X}, \forall a_m, a_n \in \mathbf{A},$$

$$\mathbf{com}(i, j, m, n) = \begin{cases} \mathbf{msg}(i, j) \cdot \mathbf{route}(m, n) & \text{if } (i, j) \in D, m \neq n \\ 0 & \text{otherwise} \end{cases} \tag{4}$$

$$\underset{x_i^m}{\text{minimize}} \quad \sum_{(i,j) \in D} \sum_{(m,n) \in \mathbf{A}^2} \mathbf{com}(i, j, m, n) \cdot \alpha_{ij}^{mn} \tag{5}$$

- Add hosting costs : $\mathbf{host}(a_m, x_i)$

$$\underset{x_i^m}{\text{minimize}} \quad \sum_{(x_i, a_m) \in X \times \mathbf{A}} x_i^m \cdot \mathbf{host}(a_m, x_i) \tag{6}$$

Binary ILP for computation distribution

$$\underset{x_i^m}{\textbf{minimize}} \quad \omega_{\mathbf{com}} \cdot \sum_{(i,j)\in D} \sum_{(m,n)\in \mathbf{A}^2} \mathbf{com}(i,j,m,n) \cdot \alpha_{ij}^{mn}$$

$$+ \quad \omega_{\mathbf{host}} \cdot \sum_{(x_i,a_m)\in X\times\mathbf{A}} x_i^m \cdot \mathbf{host}(a_m, x_i) \tag{7}$$

subject to

$$\forall a_m \in \mathbf{A}, \quad \sum_{x_i\in D} \mathbf{weight}(x_i) \cdot x_i^m \leq \mathbf{cap}(a_m) \tag{8}$$

$$\forall x_i \in X, \quad \sum_{a_m\in \mathbf{A}} x_i^m = 1 \tag{9}$$

$$\forall x_i \in X, \quad \alpha_{ij}^{mn} \leq x_i^m \tag{10}$$

$$\forall x_j \in X, \quad \alpha_{ij}^{mn} \leq x_j^m \tag{11}$$

$$\forall x_i, x_j \in X, a_m \in A, \quad \alpha_{ij}^{mn} \geq x_i^m + x_j^n - 1 \tag{12}$$

Solving the ILP for computation deployment

- NP-hard, but can be solved with branch-and-cut
  LP solvers are very good at this
- Yet, only possible for small instances
- Gives us a reference for optimality: benchmarking
- When not solvable, still gives us a metrics to compare heuristics

Menu

Hands on PyDCOP III

Distribution and deployment

1. Deploy on several machines
   https://pydcop.readthedocs.io/en/latest/tutorials/deploying_on_machines.html
2. Running a single agent
   https://pydcop.readthedocs.io/en/latest/usage/cli/run.html
3. Distributing computations / tasks
   https://pydcop.readthedocs.io/en/latest/usage/cli/distribute.html

Hands on PyDCOP III

SECP

A Very simple SECP: single room

- 3 light bulbs, 1 model and 3 rules
- `/tutorials/hands-on_3/single_room.yaml`
- Solve with

```
pydcop --log log.conf -t 10 solve \
  --algo maxsum --algo_params damping:0.8 \
  --dist adhoc single_room.yaml
```

- Result : `"cost": 702.3000000000004, ...`
  - not that good ...
  - Look at the yaml definition
  - the rules contradict each other !
- Change the yaml definition
  - comment out rules to keep only one active
  - could be done with *'read-only'* variables
  - solve it again

Hands on PyDCOP III

SECP - Running on several machines

- We used `solve`
  - great for testing
  - everything run locally, in the same process
- Launching several agents:
  - One agent for each light bulb `a1`, `a2` and `a3` (change port for each agent)

```
pydcop -v3 agent -n a1 -p 9001 \
   --orchestrator 127.0.0.1:9000
```

  - an orchestrator

```
pydcop --log log.conf -t 10 orchestrator \
  --algo maxsum --algo_params damping:0.8 \
  --dist adhoc single_room.yaml
```

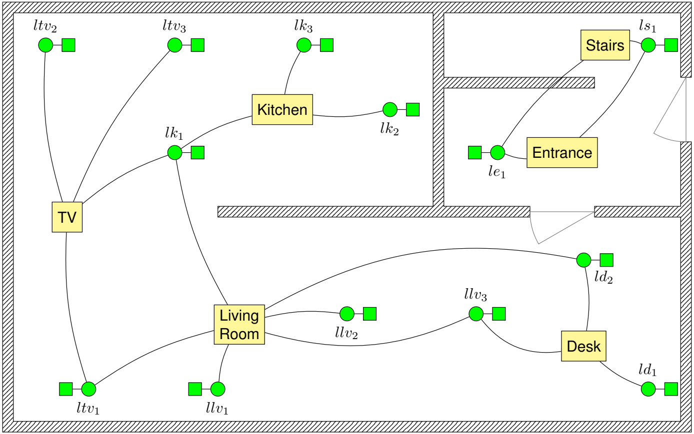
  - run the agents on different Virtual machines, different computers

Hands on PyDCOP III

A bigger SECP

in `/tutorials/hands-on_3/SimpleHouse.yml`

13 light bulbs, 6 models



Hands on PyDCOP III

Distributing a SECP

```
$ pydcop --output dist_house_fg_ilp.yaml distribute -d ilp_compref \
        -a maxsum SimpleHouse.yml
```

Need to specify the algorithm, used to deduce:

- the computation graph
- the computations' weight
- the size of computations' messages

On such a small system, we can compute the optimal distribution !

Hands on PyDCOP III

Distributing a SECP


```yaml
cost: 8725.0
distribution:
  a_d1: [mv_desk, mc_desk, l_d1, r_work, mc_livingroom, mv_livingroom]
  a_d2: [l_d2]
  a_e1: [mv_entry, r_entry, mv_stairs, l_e1, mc_entry, mc_stairs]
  a_e2: [l_e2]
  a_k1: [l_k1]
  a_k2: [l_k2]
  a_k3: [l_k3]
  a_lv1: [l_lv1]
  a_lv2: [mc_kitchen, l_lv2]
  a_lv3: [l_lv3]
  a_tv1: [l_tv1]
  a_tv2: [l_tv2]
  a_tv3: [r_lunch, l_tv3, mv_tv, r_cooking, r_homecinema, mc_tv, mv_kitchen
      ]
inputs:
  algo: maxsum
  dcop: [SimpleHouse.yml]
  dist_algo: ilp_compref
  graph: factor_graph
```

Menu

SECP and DCOP

So far we have:

- Designed a model for SECP
- Formulated this model as a DCOP
- Distributed the computation of the DCOP on devices / agents (bootstrap)
- Run our system to get self-configured devices

But what happens
in dynamic environments ?
if objects appear and disappear?

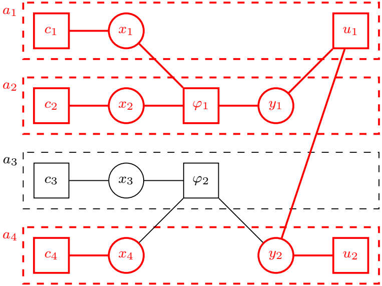
SECP is a dynamic problem

Dynamics in the infrastructure

- Devices can disappear
- New devices can be added to the system

At runtime..

- No powerful device available to solve the ILP
- The deployment must be repaired: self-adaptation
- Only consider a portion of the factor graph: the neighborhood



k-resilience

Dynamics in the infrastructure

Definition (k-resiliency)

k-resiliency is the capacity for a system to repair itself and operate correctly even in the case of the disappearance of up to k agents

- Two parts:
  - Do not loose the definition of the computations: **replication**
  - Migrate the orphaned computations to another agent: **selection** / activation
- Apply to any graph of computations, not only DCOP

Replication of computations

Replica distribution

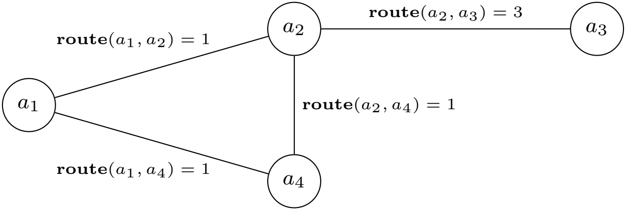
- For each computation, place k replica on k other agents
  replica $equiv$ definition of the computation
- Must be distributed !
- Optimal replication ? impact the set of available agents when repairing
  which criteria ? too hard (quadratic multiple knapsack problem)...

Distributed Replica Placement Method (DRPM)

- Heuristic : place replica on agents close (network) the active computation, while respecting capacity
- Distributed version of iterative lengthening (aka uniform cost search based on path costs)

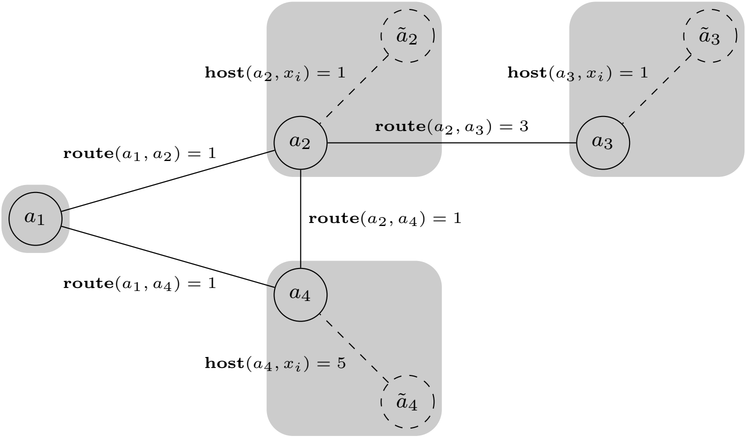
Replication of computations

iterative lengthening on route and hosting costs



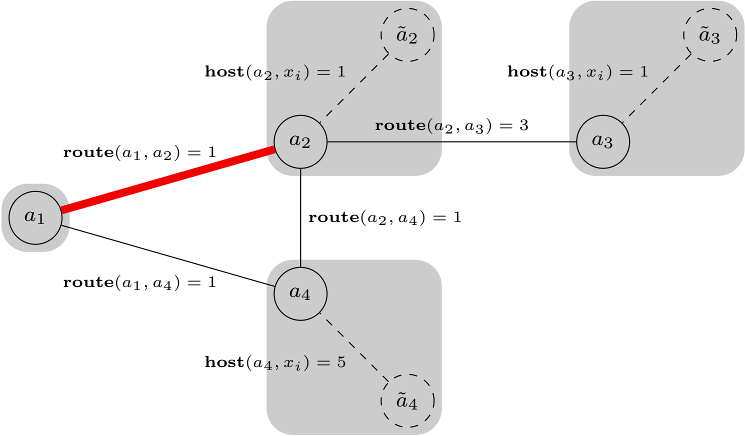
Replication of computations

iterative lengthening on route and hosting costs



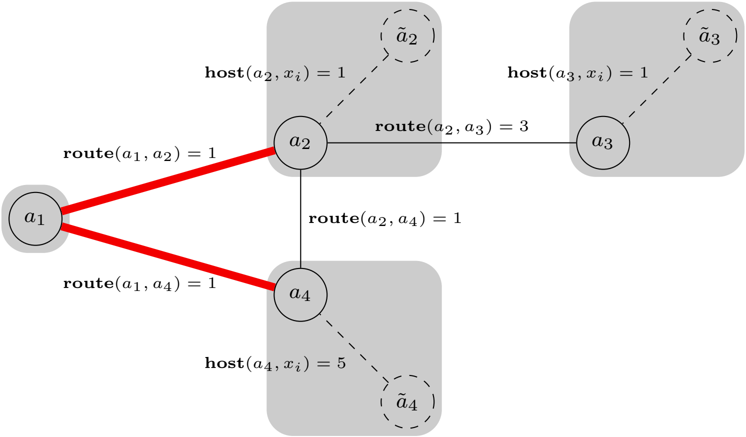
Replication of computations

iterative lengthening on route and hosting costs



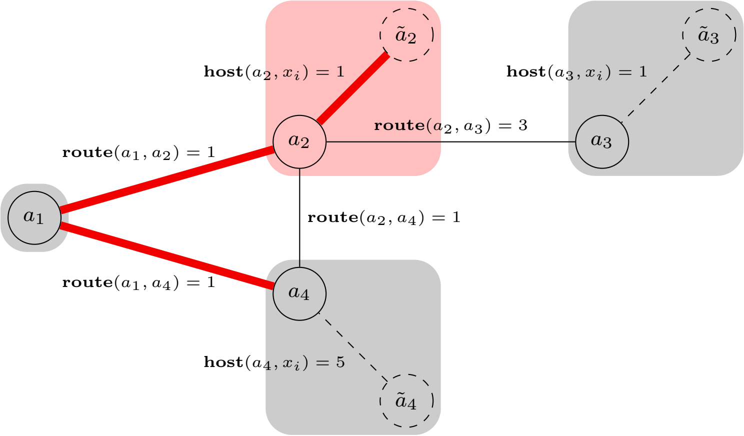
Replication of computations

iterative lengthening on route and hosting costs



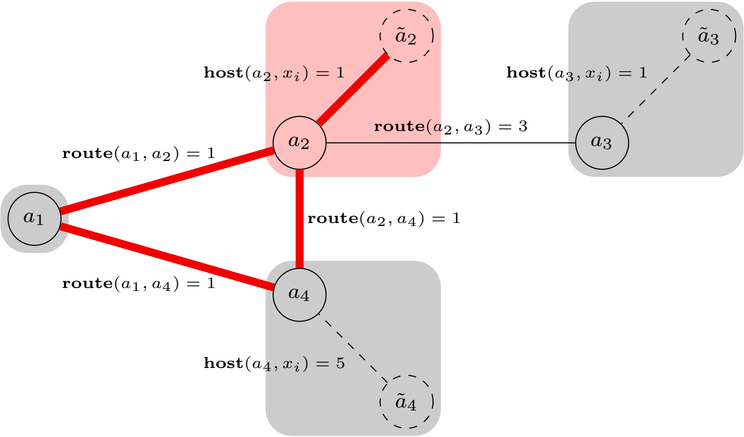
Replication of computations

iterative lengthening on route and hosting costs



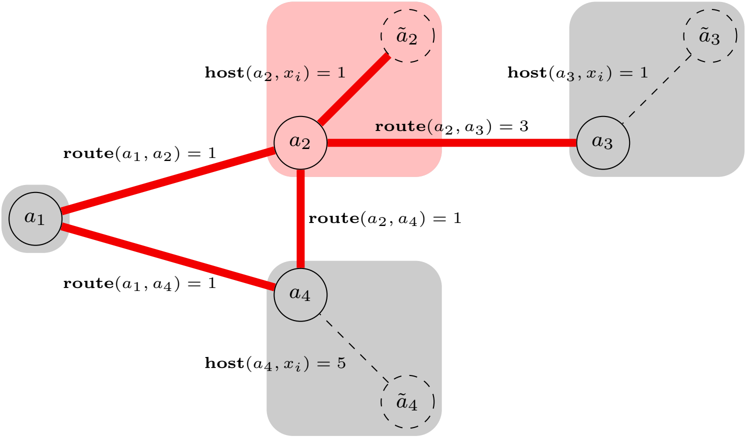
Replication of computations

iterative lengthening on route and hosting costs



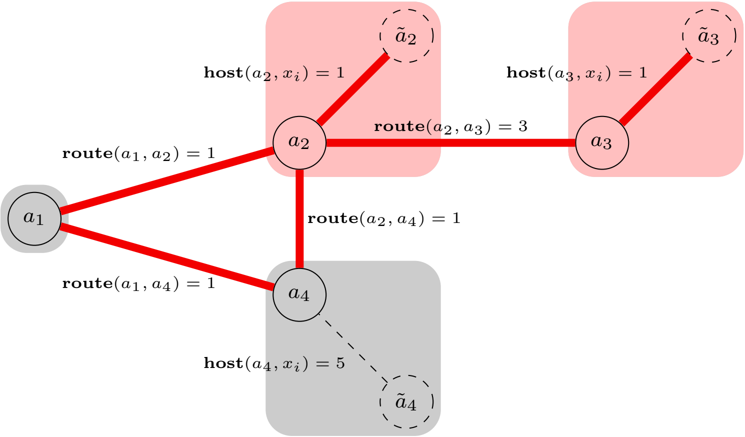
Replication of computations

iterative lengthening on route and hosting costs



Replication of computations

iterative lengthening on route and hosting costs



Migrating computations

Selecting an agent

Migrating a set of x_i computations X_c

- set of candidate agents A_c
- migrating the computation must not exceed agent's capacity
- for each computation, select the agent that minimize hosting and communication cost

Same optimization problem than for initial distribution, but on a subset of the of the graph

Distributed process !

Migrating computations
Selecting an agent

Distributed optimization problem $\Rightarrow$ let's use a DCOP!

- $\mathcal{A}$ is the set of candidate agents A_c
- $\mathcal{X}$ are the binary decision variables x_i^m
- $\mathcal{C}$ are the constraints ensuring that all computations are hosted, agent's capacities are respected and hosting and communication costs are minimized

Migrating computations

Selecting an agent

$$\sum_{a_m \in A_c^i} x_i^m = 1 \tag{13}$$

$$\sum_{x_i \in X_c^m} \mathbf{weight}(x_i) \cdot x_i^m + \sum_{x_j \in \mu^{-1}(a_m) \setminus X_c} \mathbf{weight}(x_j) \leq \mathbf{cap}(a_m) \tag{14}$$

$$\sum_{x_i \in X_c^m} \mathbf{host}(a_m, x_i) \cdot x_i^m \tag{15}$$

$$\begin{aligned} &\sum_{(x_i, x_j) \in X_c^m \times N_i \setminus X_c} x_i^m \cdot \mathbf{com}(i, j, m, \mu^{-1}(x_j)) \\ + &\sum_{(x_i, x_j) \in X_c^m \times N_i \cap X_c} x_i^m \cdot \sum_{a_n \in A_c^j} x_j^n \cdot \mathbf{com}(i, j, m, n)) \end{aligned} \tag{16}$$

Decentralized reparation

When agents are removed:

- computation to migrate = computation that were hosted on these agents
- candidate agents = remaining agents that posses a replica of these orphaned computation

Solving the migration DCOP

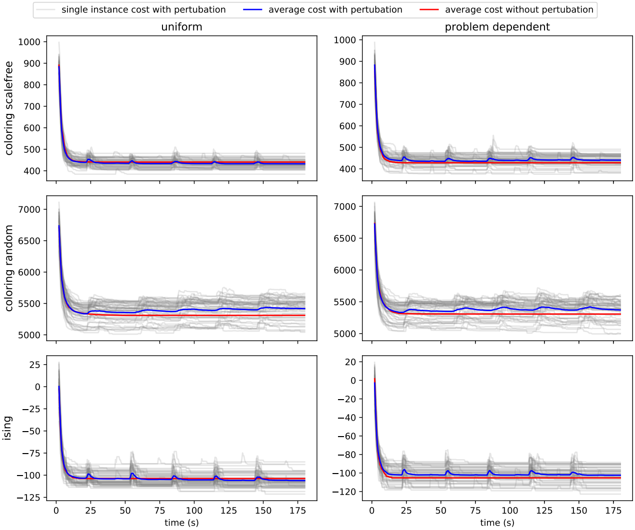
Which algorithm should we use ?

Criteria:

- lightweight
- fast (even if not optimal !)
- monotonic : mix of hard and soft constraints

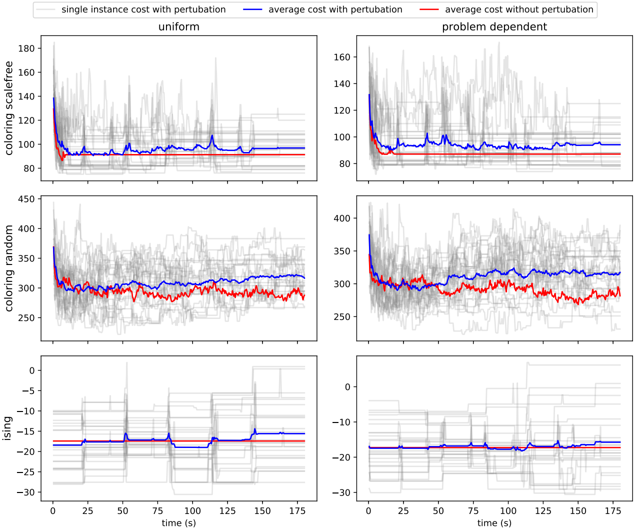
MGM-2 : like MGM, with 2-coordination

How does it behave, experimentally?



DSA

How does it behave, experimentally? (cont.)



MaxSum

Menu

To sum up

What we've seen today:

- Some generic concepts
  - How to model coordination problems using DCOP formalism
  - Some solution methods (complete and incomplete) to solve DCOP
- Some specificities of IoT-based apps
  - How to model a specific smart environment configuration problem as a DCOP
  - How to use PyDCOP to model, run, solve, and distribute DCOP
  - How to equip a system with resilience using replication and DCOP-based reparation

- Want to go deeper into DCOPs $\rightarrow$ OPTMAS-DCR workshop series (AAMAS/IJCAI), other tutorials at AAMAS/IJCAI

The End of Hands on PyDCOP

References

Boulle, M. (2004). “Compact Mathematical Formulation for Graph Partitioning”. In: *Optimization and Engineering* 5.3, pp. 315–333. issn: 1573-2924. doi: 10.1023/B:OPTE.0000038889.84284.c7. url: http://dx.doi.org/10.1023/B:OPTE.0000038889.84284.c7.

Rust, P., G. Picard, and F. Ramparany (2016). “Using Message-passing DCOP Algorithms to Solve Energy-efficient Smart Environment Configuration Problems”. In: *International Joint Conference on Artificial Intelligence (IJCAI)*. AAAI Press.