# pydcop Documentation

***Release 0.1.2a1***

**Pierre Rust**

**Jun 02, 2020**

# CONTENTS

CHAPTER

ONE

# OVERVIEW

pyDCOP is a library and command line application for **Distributed Constraints Optimization Problems** (aka DCOP).

A Distributed Constraints Optimization Problems is traditionally represented as a tuple $\langle \mathcal{A}, \mathcal{X}, \mathcal{D}, \mathcal{C}, \mu \rangle$ , where:

- $\mathcal{A} = \{a_1, \ldots, a_{|A|}\}$ is a set of agents;
- $\mathcal{X} = \{x_1, \ldots, x_n\}$ are variables owned by the agents;
- $\mathcal{D} = \{\mathcal{D}_{x_1}, \ldots, \mathcal{D}_{x_n}\}$ is a set of finite domains, such that variable $x_i$ takes values in $\mathcal{D}_{x_i} = \{v_1, \ldots, v_k\}$;
- $\mathcal{C} = \{c_1, \ldots, c_m\}$ is a set of soft constraints, where each $c_i$ defines a cost $\in \mathbb{R} \cup \{\infty\}$ for each combination of assignments to a subset of variables (a constraint is initially known only to the agents involved);
- $\mu : \mathcal{X} \rightarrow \mathcal{A}$ is a function mapping variables to their associated agent.

A *solution* to the DCOP is an assignment to all variables that minimizes the overall sum of costs.

pyDCOP has already been used for several scientific papers.

CHAPTER

TWO

# FEATURES

- pyDCOP provides implementations of many classic DCOP algorithms (DSA, MGM, MaxSum, DPOP, etc.).
- pyDCOP allows you to implement our own DCOP algorithm easily, by providing all the required infrastructure: agents, messaging system, metrics collection, etc.
- Agents can run on the same computer or on different machines, making real distributed experiments easy.
- Multi-platform : pyDCOP can run on windows, Mac and Linux.
- pyDCOP is especially suited for IoT use-case and can run agents on single-board computers like the Raspberry Pi.
- In addition to classical DCOP algorithm, pyDCOP also provide novel approaches for using DCOP in IoT systems: several strategies are available to distribute DCOP computations on agents and achieve resiliency.

CHAPTER

THREE

# DOCUMENTATION

## 3.1 Installation

PyDCOP runs on python >= 3.6. We recommend using `pip` and installing pyDCOP in a python venv:

```
python3 -m venv ~/pydcop_env
source ~/pydcop_env/bin/activate
```

Then you can simply install using pip:

```
git clone https://github.com/Orange-OpenSource/pyDcop.git
cd pyDcop
pip install .
```

When developing on pyDCOP, for example to implement a new DCOP algorithm, one would rather use the following command, which installs pyDCOP in development mode with test dependencies:

```
pip install -e .[test]
```

To generate documentation, you need to install the corresponding dependencies:

```
pip install -e .[doc]
```

Additionally, for computations distribution, pyDCOP uses the glpk linear program solver, which must be installed on the system (as it is not a python library, which could be installed as a dependency by *pip*). For example, on an Ubuntu/Debian system:

```
sudo apt-get install glpk-utils
```

**Note:** On many linux distribution, `pip` is not installed by default. On ubuntu for example, install using:

```
sudo apt-get install python3-setuptools
sudo apt-get install python3-pip
```

**Note:** When installing pyDCOP over many machines (or virtual machines), for a really distributed system, we recommend automating the process. We provide ansible playbook that can help you with this task. See the guide *Provisioning pyDCOP on many machines*.

## 3.2 Tutorials

### 3.2.1 Getting started

This small tutorial will guide you to solve your first DCOP using pyDCOP.

#### Solving your first DCOP

Once you have *installed pyDCOP* (and activated the python venv you have installed it in), create a text file `graph_coloring.yaml` with following content:

```
name: graph coloring
objective: min

domains:
  colors:
    values: [R, G]

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

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
```

(continues on next page)

(continued from previous page)

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

You don't need for the moment to understand everything in this file, it's enough to know that it represents a graph coloring problem, modeled as a DCOP with 3 variables (this example is taken from [FRPJ08] ).

Now you can simply run the following command to *solve* this DCOP with the *DPOP algorithm*:

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

This should output a result simular to this:

```
{
  // Rather long list of information, not included here ....
  "assignment": {
    "v1": "R",
    "v2": "G",
    "v3": "R"
  },
  "cost": -0.1,
  "cycle": 1,
  "msg_count": 4,
  "msg_size": 8,
  "status": "FINISHED",
  "time": 0.008432154994807206,
  "violation": 0
}
```

Congratulations, you have solved your first DCOP using pyDCOP !!

Of course, you can solve it with any other DCOP algorithm implemented by pyDCOP. Some algorithms have no default termination condition, in this case you can stop the execution with `CTRL+C` or use the `--timeout` option:

```
pydcop --timeout 3 solve --algo mgm graph_coloring.yaml
```

You may notice that with this command the assignment in the result is not always the same and not always the result we found using DPOP. This is because *MGM* is a *local search* algorithm, which can be trapped in a local minimum. On the other hand DPOP is a *complete algorithm* and will always return the optimal assignment (if your problem is small enough to use DPOP on it !).

Now that you have run your first DCOP, you can head to the next tutorial to learn how to *analyse the results*.

### 3.2.2 Analysing results

#### End results

In the *first tutorial* you solved a very simple DCOP using the following command:

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

This command outputs many results, given in json so that you can parse then easily, for example when writing more complex scripts for experimentations. You can also get these results directly in a file using the `--output` global option (see. *pyDcop cli dcoumentation*):

```
pydcop --output results.json solve --algo dpop graph_coloring.yaml
```

You may have noticed that, even though the DCOP has only 3 variables and a handful of constraints, the results is quite long and contains a lot of information.

At the top level you can find global results, which apply to the DCOP as a whole:

```
{
  "agt_metrics": {
    // metrics for each agent, see below
  },
  "assignment": {
    "v1": "R",
    "v2": "G",
    "v3": "R"
  },
  "cost": -0.1,
  "cycle": 1,
  "msg_count": 29,
  "msg_size": 8,
  "status": "FINISHED",
  "time": 0.006329163996269926,
  "violation": 0
}
```

- `status`: indicates how the command was stopped, can be one of
  - `FINISHED`, when all computation finished,
  - `TIMEOUT`, when the command was interrupted because it reached timeout set with the `--timeout` option,
  - `STOPPED`, when the command was interrupted with `CTRL+C`.
- `assignment`: contains the assignment that was found by the DCOP algorithm.
- `cost`: the cost for this assignment (i.e. the sum of the cost of all constraints in the DCOP).
- `violation`: the number of violated hard constraints, when using DCOP with a mix of soft and hard constraints (hard constraints support is still in experimental stage and not documented).
- `msg_size`: the total size of messages exchanged between agents.
- `msg_count`: the number of messages exchanged between agents.
- `time`: the elapsed time, in second.
- `cycles`, the number of cycles for DCOP computations. In this example it is always 0 has DPOP has no notion of cycle.

The `agt_metrics` section contains one entry for each of the agents in the DCOP. Each of these entries contains several items:

```
"a1": {
  "activity_ratio": 0.24987247830102727,
  "count_ext_msg": {
    "v2": 2
  },
  "cycles": {
    "v2": 0
  },
  "size_ext_msg": {
    "v2": 4
```

(continues on next page)

(continued from previous page)

```
    }
},
```

- `count_ext_msg` and `size_ext_msg`, which contain count and size of messages sent and received by each DCOP computations hosted on this agent
- `cycles`, the number of cycles for each DCOP computation hosted on this agent. In this example it is always 0 has DPOP has no notion of cycle.
- **`activity_ratio`, the ratio of *active time* on total elapsed time,** where active time is defined as the time the agent spent handling a message (as opposed to waiting for messages).

### Run-time metrics

The output of the *solve* command only gives you the end results of the command, but sometime you need to know what happens *during* the execution of the DCOP algorithm. pyDCOP is able to collect metrics at different events during the DCOP execution. You must use the `--collect_on` option to select when you want metrics to be captured:

- `cycle_change`: metrics are collected every time the algorithm switch from one cycle (aka iteration) to the next one. This mode only makes sense with algorithms which have a notion of cycle.
- `period`: metrics are collected periodically , for each agent (meaning your will have line of metric for each agent and for each period). When using this mode, you **must** also set `--period` to set the metrics collection period (in second).
- `value_change`: metrics are collected every time a new value is selected for a variable.

When collecting run-time metrics, you also need to use the `--run_metrics <file>` option to indicates the file where the metrics must be written. One csv line is written in this file for each metric collection. These lines contains the following columns: `time, cycle, cost, violation, msg_count, msg_size, status` , where the definition of these metrics are the same than for end-results.

For example, this line contains metrics from cycle 14 and the solution at this point has a cost of 0.1:

```
time, cycle, cost, violation, msg_count, msg_size, status
0.10148727701744065, 14, 0.1, 0, 112, 112, RUNNING
```

**Warning:** Be careful when collecting metrics with a large number of agents. For example, A `--period` of 0.1 means each agent will send its metrics 10 times per second. If you have 100 agents, collecting 1000 metrics per second will slow you down quite a bit and could even give invalid results if all these metrics could not be handled and written before the end of the timeout.

If you want to assess the overall time needed to solve a problem, it is better not to collect run time metrics at the same time. In that case, you should only rely on the end-metrics, which does not influence the execution of the algorithm.

### Examples

For more interesting results, we use a bigger DCOP in these samples: `graph_coloring_50.yaml` It's a graph coloring problem with 50 variables, generated with the generate command :

Solving with MGM (stopping after 20 cycles), collecting metrics on every cycle change:

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

Solving with MGM during 5 seconds, collecting metrics every 0.2 second:

```
pydcop -t 5  solve --algo mgm --collect_on period --period 0.2 \
                   --run_metric ./metrics_on_period.csv \
                   graph_coloring_50.yaml
```

Solving with MGM during 5 seconds, collecting metrics every time a new value is selected:

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

### Plotting the results

pyDCOP has not builtin utility to plot the metrics generated by the *solve* command. However, using the generated csv files, it's very easy to generate graphs for these metrics using any of the commonly used plot utility like gnu-plot, R, matplotlib, etc.

For example, if you generate cycle metrics when solving the graph coloring dcop with MGM:

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

This should give you a metric file similar to `this one`. You can now plot the cost of the solution over cycles. Notice that the cost is always decreasing, as MGM is monotonous:

```
import matplotlib.pyplot as plt
import numpy as np

data = np.genfromtxt('metrics_cycle.csv', delimiter=',',
                     names=['t', 'cycle', 'cost', 'violation' ,
                            'msg_count', 'msg_size', 'status'])

fig, ax = plt.subplots()
ax.plot(data['t'], data['cost'], label='cost MGM')
ax.set(xlabel='cycle', ylabel='cost')
ax.grid()
plt.title("MGM cost")

fig.savefig("mgm_cost.png", bbox_inches='tight')
plt.legend()
plt.show()
```

For course, before running this example, you need to install matplotlib:

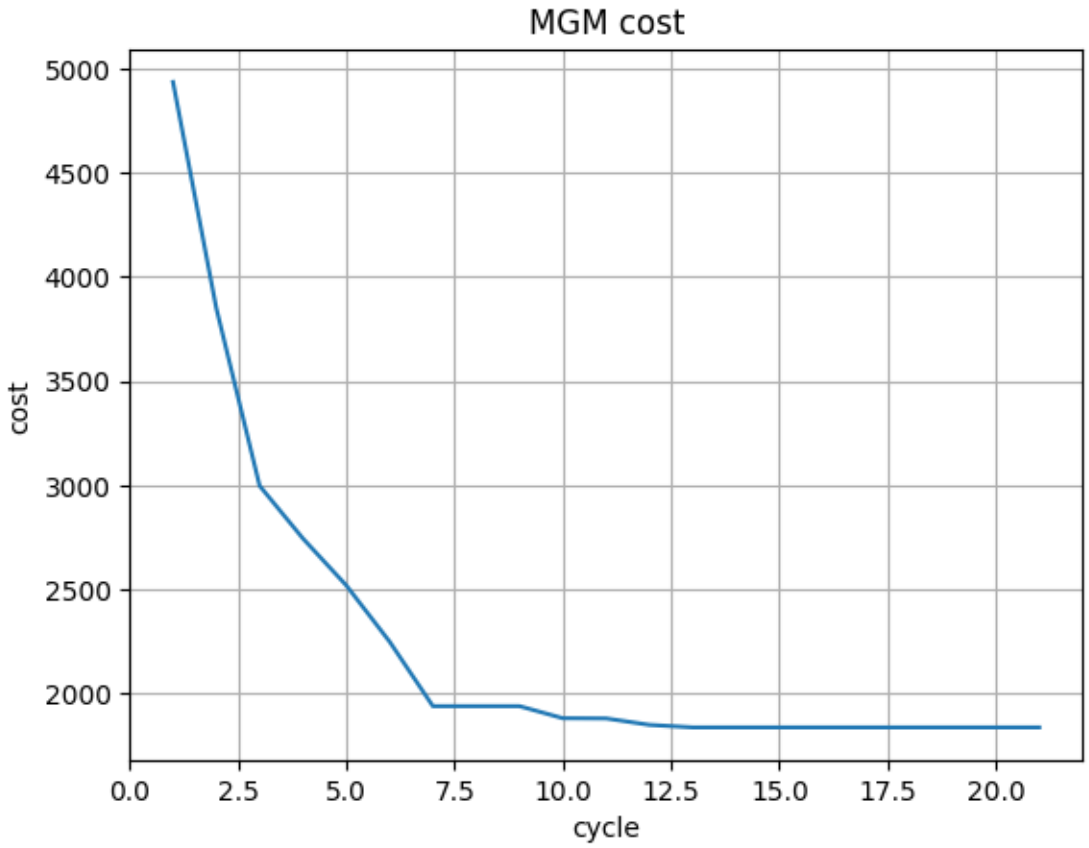


Fig. 3.1: MGM solution cost over 20 cycles.

```
pip install matplotlib
```

### Logs

By default, the *solve* command (like all other pyDCOP commands) only outputs the results (here, the end metrics) and does not output any log, except if there are errors. You can enable logs by adding the `-v` *global option* with the requested level:

```
pydcop -v 2 solve --algo dpop graph_coloring.yaml
```

Level 1 displays only warnings messages, level 2 displays warnings and info messages and level 3 all messages (and can be quite verbose! )

For more control over logs, you can use the `--log <conf_file>` *option*, where `conf_file` is a standard python log configuration file:

```
pydcop --log algo_logs.conf solve --algo dpop graph_coloring.yaml
```

For example, using `this long configuration file`, all logs from DPOP computations will be logged in a `agents.log` file, without any log from the pyDCOP infrastructure (discovery, messaging, etc.). This can be very useful to analyse an algorithm's behavior. When solving our graph coloring problem with DPOP, you should get a log file containing something similar to this:

```
pydcop.algo.dpop.v3 -  Leaf v3 prepares init message v3 -> v2
pydcop.algo.dpop.v2 -  Util message from v3 : NAryMatrixRelation(None, ['v2'], [-0.1
↪0.1])
pydcop.algo.dpop.v2 -  On UTIL message from v3, send UTILS msg to parent ['v3']
pydcop.algo.dpop.v1 -  Util message from v2 : NAryMatrixRelation(None, ['v1'], [0. 0.
↪])
pydcop.algo.dpop.v1 -  ROOT: On UNTIL message from v2, send value msg to childrens [
↪'v2']
pydcop.algo.dpop.v1 -  Selecting new value: R, -0.1 (previous: None, None)
pydcop.algo.dpop.v1 -  Value selected at v1 : R - -0.1
pydcop.algo.dpop.v2 -  v2: on value message from v1 : "DpopMessage(VALUE,
↪([Variable(v1, None, VariableDomain(colors))], ['R']))"
pydcop.algo.dpop.v2 -  Slicing relation on {'v1': 'R'}
pydcop.algo.dpop.v2 -  Relation after slicing NAryMatrixRelation (joined_utils, ['v2
↪'])
pydcop.algo.dpop.v2 -  Selecting new value: G, 0.0 (previous: None, None)
pydcop.algo.dpop.v2 -  Value selected at v2 : G - 0.0
pydcop.algo.dpop.v3 -  v3: on value message from v2 : "DpopMessage(VALUE,
↪([Variable(v2, None, VariableDomain(colors))], ['G']))"
pydcop.algo.dpop.v3 -  Slicing relation on {'v2': 'G'}
pydcop.algo.dpop.v3 -  Relation after slicing NAryMatrixRelation(joined_utils, ['v3'])
pydcop.algo.dpop.v3 -  Selecting new value: R, 0.1 (previous: None, None)
pydcop.algo.dpop.v3 -  Value selected at v3 : R - 0.1
```

### 3.2.3 Deploying on several machines

In the first tutorials, you have been using the *solve command* to run your DCOPs. This command is very convenient as it handles a lot of *plumbing* details for you, but it only works if you want to run the whole system on a single machine. In this tutorial, you will learn how to really distribute your system, by running different agents on different machines.

#### Running independent agents

If you want to use several machine to run your DCOP (remenber, the **D** stands for Distributed !) you need to use the *agent* and *orchestrator* commands.

#### Orchestrator

The **Orchestrator** is a special agent that is not part of the DCOP: it's role is to bootstrap the solving process by distributing the computations on the agents. It also collects metrics for benchmark purpose. Once the system is started (and if no metric is collected), the orchestrator could be removed. In any case, the orchestrator never participates in the coordination process, which stays fully decentralised.

The *orchestrator* command looks very much like the *solve command* ; it takes a DCOP yaml file as input and supports the same `--algo`, `--ditribution` options. The main difference is that the orchestrator command only launches an orchestrator, which then waits for agents to enter the system. The DCOP algorithm will only be started once all required agents have been started.

For example, using `this graph coloring problem definition file`, you can start an orchestrator:

```
pydcop -v 3 orchestrator --algo mgm --algo_param stop_cycle:20 \
                         graph_coloring_3agts.yaml
```

Once the DCOP algorithm finishes, or when reaching the timeout, the command outputs the end-results. The content and format is the same than what is described in *Analysing results*.

All metrics-collection options can also be used with the *orchestrator* and works the same way than with the *solve command* command.

#### Agents

The *agent* command launches an agent on the local machine (actually it can also launch several agents, see the *detailed command documentation*). Initially, this agent does not know anything about the DCOP (variables, constraints, etc. ). It only knows the address of an **orchestrator**, which is responsible for sending DCOP information to all agents in the system:

```
pydcop -v 3 agent -n a1 -p 9001 --orchestrator 192.168.1.10:9000
```

#### Example

Instead of using solve, you can run the very simple DCOP used in *the first tutorial* on different machines. For easier setup, we reduces the agents number to 3 in this file : `graph_coloring_3agts.yaml`.

First launch the orchestrator on a machine:

```
pydcop -v 3 orchestrator --algo mgm --algo_param stop_cycle:20 \
                         graph_coloring_3agts.yaml
```

You must check in the logs the ip address and port the orchestrator is listening on, or you can set it using `--address` and `--port`

Now launch on 3 different machines (or virtual machines) the following commands to run 3 agents that all use the orchestrator started before (make sure you give them the right IP address and port!):

```
# Machine 1 runs agent a1
pydcop -v 3 agent -n a1 -p 9001 --orchestrator 192.168.1.10:9000
# Machine 2 runs agent a2
pydcop -v 3 agent -n a2 -p 9001 --orchestrator 192.168.1.10:9000
# Machine 3 runs agent a3
pydcop -v 3 agent -n a3 -p 9001 --orchestrator 192.168.1.10:9000
```

Each agent receives the responsibility for one of the variables from the DCOP and runs MGM for 20 cycles. Once each agent has performed 20 cycles, the agents and the orchestrator commands return.

---

**Note:** If you know in advance the IP address and port the orchestrator will use, you can launch the agents before the orchestrator. In that case, agents will periodically attempt to connect to the orchestrator, until they can reach it.

---

#### Provisioning pyDCOP

You may have noticed that the previous section silently assumed that pyDCOP was installed on every machine you want to use in your system. Indeed, we use the `pydcop` command line application, which is only available if you have installed pyDCOP!

Of course, you can simply follow the *installation instructions* to install manually pyDCOP on all your machines, but the process is rather tedious and error prone. Moreover, if you are working on DCOP algorithms, you will probably make changes in pyDCOP implementation (at least in the implementation of your algorithm), which requires updating it on all your machine, copying the new development version on all machines, reinstalling it, etc.

When running a large system, one needs to automate this kind of tasks. To help you with this, we provide as set of ansible playbooks that automates the installation process. See the *Provisioning* guide for full details.

### 3.2.4 Modeling problems as DCOPs

What we have seen so far in previous tutorials may seems very theoretical and it might not be obvious how DCOP could be used to solve real-world problems. pyDCOP is meant to be domain-independent, but we hope to convince you that the DCOP algorithms implemented and studied with it can be applied to real applications.

As a topic in the Multi-Agent System field, DCOP are obviously best suited to problems that are distributed in nature. They have been applied to a wide variety of applications, including disaster evacuation [KSL+08], radio frequency allocation [MPP+12], recommendation systems [LdSFB08], distributed scheduling [MTB+04], sensor networks [ZWXW05], intelligent environment [RPR16], [FYP17], smart grid [CPRA15], etc.

When using a DCOP approach on a distributed problem, the first steps are always to cast your problem into an **optimization problem** and to **identify your agents**. Then you can select, and probably benchmark, the best algorithm for the settings of your problem. In this tutorial, we will present one way of modelling a target tracking problem as a DCOP.

### Example: Target tracking problem

Detecting and tracking mobile objects is a problem with many real applications like surveillance and robot navigation, for example. The goal of such a system is to detect foreign objects as quickly and as many as possible.

The system is made of several sensor scattered in space. Each sensor, for example a small Doppler effect sensor, can only scan a fixed radius around itself at any given time; it has to select with area it operates on.

In an open environment, sensors used in tracking systems usually run on battery, which means they must use as little energy as possible, in order to increase the system operation's lifetime. This includes switching them-self off and on whenever possible, in a way that does not affect the system's detection performance.

These sensors are also lightweight devices with limited memory and computation capability. They communicate one with another through radio signals, which may not be reliable. Each sensor can only only communicates with neighboring sensors and has no global information on the whole system.

The overall goal is thus to provide the **best detection** possible, while **preserving energy** as much as possible. To achieve this goal, sensors can act on several parameters:

- selecting which area to scan
- selecting when to switch on and off

### Example: Target tracking DCOP model

Each sensor is controlled by one agent, which decides the sector the sensor is scanning. These agents coordinate in order to plan an efficient scanning strategy ; this problem is actually a distributed planning system.

Let $S = \{S_1, ...S_n\}$ be the set of **n sensors**. Each agent $S_i$ can select the area to scan among **k sectors** $s_i = \{s_i^1, ...s_i^k\}$.

The agents plan their action over a horizon $T$ made of $|T| = t$ time slots. For each time slot, each agent $S_i$ has to select one action : either scan one of it's $s_i^j$ sectors or sleep.

The $s_i^k$ are our **decision variables**, whose value represents the sector scanned by a sensor at a given time. These variables take their value from a domain $D = \{1, ...t\}$ ; when the variable $s_i^k$ takes the value $t$, it means that the sensor $S_i$ will scan the sector $s_i^k$ during the time slot $t$.

Of course, a sensor can only scan a single sector at any given time. This can be modelled by defining a set of constraints (3.1) ensuring that two sectors from the same sensor cannot take the same value:

$$\forall s_i^p, s_i^q \in s_i \times s_i, p \neq q \Rightarrow s_i^p \neq s_i^q \tag{3.1}$$

For an efficient scanning process, we want to avoid several sensors scanning simultaneously the same sector. For this we define a function $w$ between a pair of sectors $s_i^p, s_j^q$ where $w(s_i^p, s_j^q)$ is the surface of the area common to these two sectors. Then we use this function to define constraints (3.2) between sectors, where the cost of the constraints is this surface, if the sensors of these two sector at scanning at the same time.

$$c(s_i^p, s_j^q) = \begin{cases} w(s_i^p, s_j^q) & \text{if} s_i^p == s_j^q \\ 0 & \text{otherwise} \end{cases} \tag{3.2}$$

With all these definitions, we can formulate the target tracking problem as a DCOP $\langle \mathcal{A}, \mathcal{X}, \mathcal{D}, \mathcal{C}, \mu \rangle$ , where:

- $\mathcal{A} = \{S_1, ...S_n\}$ is the set of sensors;

- $\mathcal{X} = \{s_i^p\}, \quad S_i \in \mathcal{A}, \quad 0 \leq p \leq k$ is the set of variables, for the k sectors of these n sensors;
- $\mathcal{D} = \{0, ...t\}$ is the domain for these variable, made of the time slots in the forecasted horizon;
- $\mathcal{C}$ is the set of constraints over these variables, made of constraints (3.1) and (3.2);
- $\mu$ is a mapping function that assign each $s_i^p$ variable to the agent $S_i$.

We can now use a DCOP algorithm to solve this problem in a distributed manner. Of course, the choice of the algorithm depends on the problem and the environment characteristics; given that sensors have limited cpu and memory and that the communication channel has a low bandwidth, lightweight local search algorithm like DSA and MGM are good candidates. The original article this model comes from, [ZXWW03], evaluates DSA and DBA and shows that, if controlled properly, DSA is significantly superior to DBA, finding better solutions with less computational cost and communication overhead.

---

**Note:** In order to keep this tutorial short and relatively easy to read, the model presented here is a simplified version of the model exposed in [ZXWW03]. As you may have noticed, we do not take into account the possibility for an agent to *'sleep'* in order to save energy ; we only optimize the tracking to avoid inefficiencies. Moreover, the original model allows selecting several time slots for the same sector, which maps the target tracking problem to a multicoloring graph problem.

---

### 3.2.5 Dynamic DCOPs

All the DCOPs we have seen in previous tutorial are *static DCOPs* : the problem definition (variables and constraints) and the set of agents are supposed to be stable during the solving process.

However, as for most distributed systems, DCOPs are often used on problems where the environment is dynamic in nature. Agents may enter or leave the system at any time, the problem itself may evolve *while* it is been worked on, and the system needs to adapt to these changes. We call such problems **dynamic DCOP** (see. *Resilient DCOP*).

The *solve* command deals with static DCOP, the *run* command supports running **dynamic DCOP**.

#### DCOP and scenario

Running a dynamic DCOP requires injecting events in the system, otherwise it would be the exact same thing as running a static DCOP using the *solve* command.

In order to run and evaluate DCOP solution methods in dynamic environments, pyDCOP defines the notion of **scenario**, which is an ordered collection of events that are injected in the system while it is running. Scenario can be *written in yaml*. Each event in the scenario is either a **delay**, during which the system runs without any perturbation, of a collection of **actions**. For example, in the following scenario, the agent a008 is removed after a 30 seconds delay:

```
events:
  - id: w1
    delay: 30

  - id: e1
    actions:
      - type: remove_agent
        agent: a008
```

Removing an agent is the only action supported at the moment (but that should change soon!) . Other actions could be:

- changing a constraint value table (the cost of the constraint for a given assignment)

- modifying the scope and arity of a constraint
- changing the value of a *read-only* variable

Currently pyDCOP has a partial implementation for these kinds of events, which are not yet available through the command line interface.

Also, pyDCOP implementation *simulates* dynamic DCOPs, which allows evaluating algorithms (for resilient DCOP for example) but does entirely support a real dynamic behavior where events happen in the environment and are *sensed* at agent level.

The consequence it that removing an agent can only be done through a scenario: if you manually remove an agent from the system (by stopping it's `pydcop agent` command) the computations will not be migrated and the system will malfunction. Supporting removal like this would require a distributed *discovery mechanism*, with *keep alive* messages, in order to be able to detect when an agent is not available. This mechanism is not (yet) implemented in pyDCOP and all events are injected through the orchestrator.

### Resiliency and replication

As pyDCOP implementation is currently focused on resilience, the run command automatically deploy and run a resilient DCOP.

This means that all the computations for the DCOP are automatically replicated on several agents. Without this step, we would loose computation when injecting `remove_agent` events, and the system would stop working properly. The resiliency level and replication method can be selected when using the *run* command, but for now, you don't need to bother with these as safe default values are provided.

For more information on resiliency, replication and reparation, see *Resilient DCOP*

### Running the dynamic DCOP

We can now run our dynamic DCOP using the following command:

```
pydcop -t 120 --log log.conf run \
       --algo maxsum --algo_params damping:0.9 \
       --collect_on value_change --run_metric metrics_value_maxsum.csv \
       --distribution heur_comhost \
       --scenario scenario_2.yaml \
       graph_coloring_20.yaml
```

When running this command, the following steps are performed :

- Building the computation graph required for the DCOP algorithm. Here it is a factor graph with one computation for each variable and factor (aka constraint) in the DCOP
- Computing an initial distribution of these computations on the the agents, using the heuristic `heur_comhost` based on communication and hosting costs
- replicating all computations on other agents, using the DRPM replication method (from [RPR18])
- injecting delays and events from the scenario. After each event the system is repaired, which includes:
  - For each of the computations that were hosted on a departed agent, activating one of its replicas
  - Repairing the replication by creating new replicas for all computations that were replicated on a departed agent

You can try it on your computer using the following files:

- dcop: `graph_coloring_20.yaml` We use here a weighted graph coloring problem. Agents defined in this DCOP have a *capacity*, *hosting costs* and *communication costs*, which are used when computing the distribution
- scenario: `scenario_2.yaml` this scenario contains 3 events, where
- log configuration file: `log.conf`

### Results

We can plot the results as we did in *a previous tutorial*. Use `this script` if you're not confortable with matplotlib. As we can see, the system keeps running, and the solution improves, even though we removed agents:

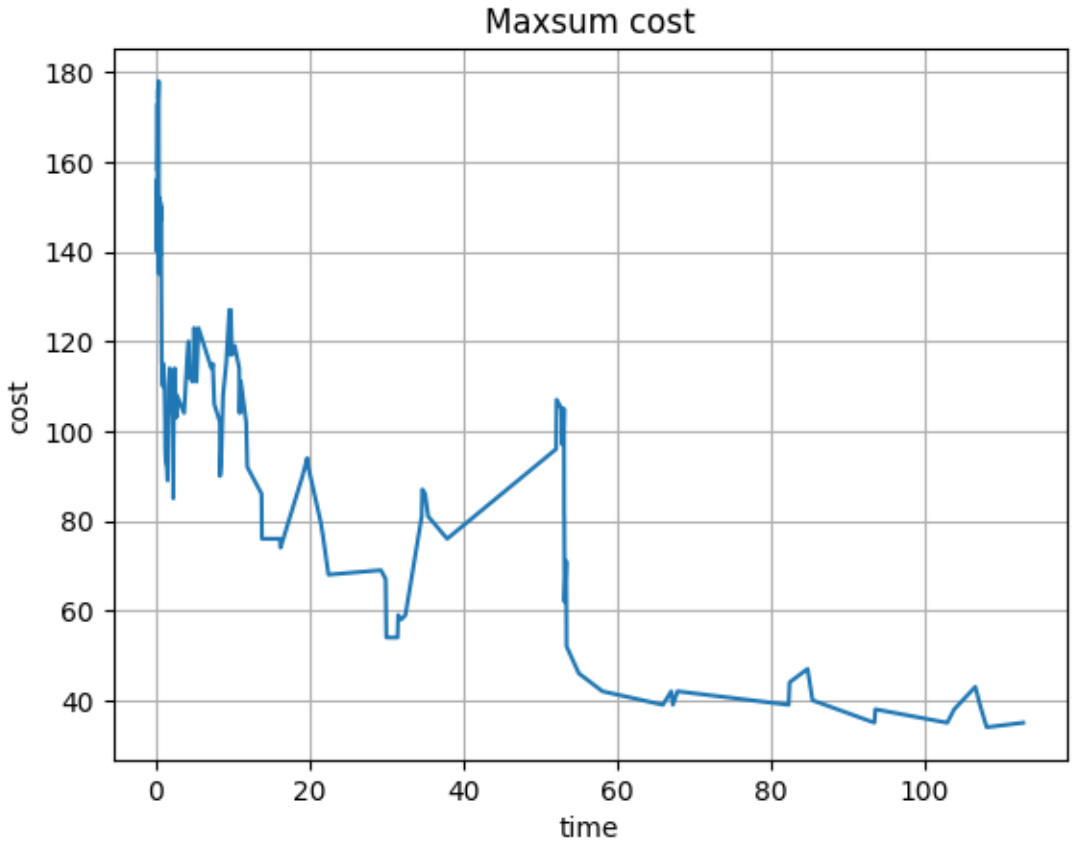


Fig. 3.2: Solution cost when running MaxSum on dynamic DCOP

In addition to the usual run-time metrics, the *run* command also outputs, after each event and reparation, the new distribution of the system. The *format* is the same as with the *distribute* command.

With our example, 3 distribution files are created: `evtdist_0.yaml`, `evtdist_1.yaml`, `evtdist_2.yaml`. You can open them and see how computations were moved from one agent to another after each event.

Notice that we use here *maxsum* to solve our distributed DCOP ; as we remove agents at run-time, the algorithm must be able to cope with message loss, which is not the case with synchronous algorithms like DSA or MGM.

Finally, you may have noticed that the end results contains some variables that were not part of our initial DCOP , like for example `Bc001_005_a002`.

These are binary variables used to select the agent where an orphaned computation should be migrated. As a matter of fact, the reparation of our DCOP is a distributed problem, which is it-self implemented with another DCOP ! See.

[RPR18] for more details:


```
{
  ...
  "assignment": {
    "Bc001_005_a002": 1,
    "Bc001_005_a008": 0,
    "Bc001_005_a015": 0,
    "Bc001_010_a002": 0,
    "Bc001_010_a011": 0,
    "Bc001_010_a018": 1,
    "Bc003_004_a002": 0,
    "Bc003_004_a014": 1,
    "Bc003_004_a018": 0,
    "Bc003_010_a013": 1,
    "Bc003_010_a016": 0,
    "Bc003_010_a018": 0,
    "Bc005_016_a004": 0,
    "Bc005_016_a005": 1,
    "Bc005_016_a013": 0,
    "Bc010_011_a000": 0,
    "Bc010_011_a005": 0,
    "Bc010_011_a011": 1,
    "Bc010_012_a004": 1,
    "Bc010_012_a005": 0,
    "Bc010_012_a011": 0,
    "Bc010_017_a002": 0,
    "Bc010_017_a004": 0,
    "Bc010_017_a007": 1,
    "Bc010_019_a000": 0,
    "Bc010_019_a003": 1,
    "Bc010_019_a013": 0,
    "Bv001_a005": 0,
    "Bv001_a009": 1,
    "Bv001_a018": 0,
    "Bv005_a005": 1,
    "Bv005_a009": 0,
    "Bv005_a016": 0,
    "Bv010_a009": 0,
    "Bv010_a015": 0,
    "Bv010_a018": 1,
    "Bv017_a011": 0,
    "Bv017_a016": 0,
    "Bv017_a017": 1,
    "Bv019_a012": 0,
    "Bv019_a014": 1,
    "Bv019_a019": 0,
    "v000": 2,
    "v001": 0,
    "v002": 5,
    "v003": 7,
    "v004": 2,
    "v005": 2,
    "v006": 2,
    "v007": 0,
    "v008": 5,
    "v009": 4,
    "v010": 5,
```


(continues on next page)

(continued from previous page)


```
    "v011": 2,
    "v012": 0,
    "v013": 9,
    "v014": 4,
    "v015": 8,
    "v016": 0,
    "v017": 5,
    "v018": 6,
    "v019": 3
  },
  "cost": 35,
  "cycle": 0,
  "msg_count": 23529,
  "msg_size": 470580,
  "status": "TIMEOUT",
  "time": 120.01458694699977,
  "violation": 0
}
```


### 3.2.6 Agent's GUI

Start the Web UI for agent.

cd ~/pyDcop-ui/dist python3 -m http.server 4001

Run a dynamic DCOP, using the extra using the following command:

```
pydcop --log log.conf orchestrator \
       --algo maxsum --algo_params damping:0.9 \
       --distribution heur_comhost \
       --scenario scenario_2.yaml \
       graph_coloring_20.yaml

pydcop agent --names a000 a001 a002 a003 a004 a005 a006 a007 a008 a009 a010 \
  a011 a012 a013 a014 a015 a016 a017 a018 a019 a020 \
   -p 9001 --orchestrator 127.0.0.1:9000 --uiport 10001
```

### 3.2.7 Implementing a DCOP algorithm

One of pyDCOP goals is to help the study and development of DCOP algorithms. For this purpose, pyDCOp allows you to implement easily your own algorithms and provide all the required infrastructure so that you can concentrate on your algorithm logic.

To demonstrate this we will implement a very simple DCOP algorithm in this tutorial, the Distributed Stochastic Algorithm (DSA).

#### Distributed Stochastic Algorithm

DSA is a synchronous stochastic local search algorithm that works on a constraints graph. At startup, each variable takes a random value from it's domain and then run the same procedure in repeated steps.

At each step, each variable send its value to its neighbors. Once a variable has received the value from all it's neighbors, it evaluates the gain it could obtain by picking another value. If this gain is positive, it decides to change it's value or to keep the current one. This decision is made stochastically: a variable change its value with probability **p** (if doing so can improve the state quality).

The algorithm stops after a predefined number of steps.

#### Note

For example purpose, we only consider here a a very simple version of DSA, which implements DSA-A as described in [ZWXW05]. pyDCOP also has a full implementation of DSA, with several variants, both synchronous and asynchronous, which can be used as `dsa` and `adsa` in the *solve*, *run* and *orchestrator* commands.

#### Implementation with pyDCOP

#### Basic setup

In pyDCOP, each algorithm is implemented as a module in the `pydcop.algorithms` package. Thus for our DSA implementation, we simply create a `dsa_tuto.py` file in the directory `pydcop/algorithms`.

We must then declare wich *graph model* our algorithm works on. This is done by declaring a `GRAPH_TYPE` variable:

```
# Type of computations graph that must be used with dsa-tuto
GRAPH_TYPE = 'constraints_hypergraph'
```

Then we need to define the message(s) for this algorithm. A message is defined by a class that inherits *pydcop.infrastructure.computations.Message* but you can use the *message_type()* function, which creates the subclass automatically for you. DSA uses one single type of message, which contains the value of the variable sending the message:

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

### Algorithm implementation class

Generally speaking, an algorithm is implemented as a subclass of *DcopComputation*. For algorithms that define computations for variables (i.e. most classic algorithms), you must subclass *VariableComputation*, which defines several utility methods for handling variables:

```
class DsaTutoComputation(VariableComputation):
```

One instance of `DsaTutoComputation` will be created by pyDCOP for each variable in our DCOP.

The constructor of our class must accept a `ComputationDef` instance as argument. As its name implies, a `ComputationDef` instance is an object that fully describes a computation an can be used to instanciate one. It is made of a `ComputationNode` and a `AlgorithmDef`. You don't need to bother with this for now, these instances will be automatically be created by pyDCOP and passed to your constructor. With this, we can now write our computation's constructor:

```
class DsaTutoComputation(VariableComputation):

    def __init__(self, computation_definition: ComputationDef):
        # Always call the super class constructor !
        super().__init__(computation_definition.node.variable,
                         computation_definition)

        # Constraints involving this variable are available on the
        # ComputationNode:
        self.constraints = computation_definition.node.constraints

        # The assignment of our neighbors for the current and next cycle
        self.current_cycle = {}
        self.next_cycle = {}
```

### Startup

When pyDCOP starts a computation its `on_start` method is automatically called. You can use it for any startup logic. In the case of DSA, the computation must pick a value for the variable it represents:

```
def on_start(self):
    # This picks a random value form the domain of the variable
    self.random_value_selection()

    # The currently selected value is available through self.current_value.
    self.post_to_all_neighbors(DsaMessage(self.current_value))
    self.evaluate_cycle()  # Defined later
```

### Message handling

Once started, computations communicate one with another by sending messages. In order to handle the messages sent to by the computation, you must register a message handler using the *register()* decorator : `@register("dsa_value")`.

For DSA, when receiving a message, we store the value and check if we received a value from all our neighbors, in which case we can evaluate whether we should pick a new value for our variable. Note that there might be an offset of one cycle with our neighbor.

Here is the corresponding message handler:

```
@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
        if self.is_cycle_complete():
            self.evaluate_cycle()

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

Finally, we can decide if we should select another value by computing the potential gain and drawing a random number:

```
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():
        # Select a new value
        self.value_selection(arg_min)

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

def is_cycle_complete(self):
    # The cycle is complete if we received a value from all the neighbors:
    return len(self.current_cycle) == len(self.neighbors)

def compute_best_value(self) -> Tuple[Any, float]:
    # compute the best possible value and associated cost
    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
```

### Running the algorithm

You now have a full working implementation of DSA. For reference, this implementation is also available in this file : `dsa-tuto.py`.

If you did not follow the tutorial, you can simply copy this file in `pydcop/algorithms`.

This implementation can be used with any pydcop command, for example for solving a graph coloring DCOP for 50 variables (`graph_coloring_50.yaml`) you can use:

```
pydcop --timeout 10 -v 3 solve --algo dsa-tuto graph_coloring_50.yaml
```

Note that this tutorial only covers the basics of DCOP algorithms implementation, for more details, look at *DCOP algorithm Implementation*.

## 3.3 Usage

pyDCOP can be used as a command-line application or as an library, using its API.

### 3.3.1 pyDCOP command line reference

#### pydcop agent

`pydcop agent` runs one or several standalone agent(s).

#### Synopsis

```
pydcop agent --names <names>
             [--address <ip_address>] [--port <start_port>]
             --orchestrator <ip[:<port>]>
             [--uiport <start_uiport>]
             [--restart]
             [--delay <delay>]
```

#### Description

Starts one or several agents. No orchestrator is started, you must start it separately using the *:ref:`pydcop_commands_orchestrator* command. The orchestrator might be started after or before the agents, if it is not reachable when starting the agents, they will wait until it is available.

All agents are started in the same process and communicate with one another using an embedded http server (each agent has its own http server). You can run this command several time on different machines, all pointing to the same orchestrator ; this allows to run large distributed systems.

The ui-server is a websocket server (one for each agent) that gives access to an agent internal state. It is only needed if you intend to connect a graphical user interface for an agent ; while it is very useful it also may have some impact on performance and is better avoided when running a system with a large number of agents.

**See also:**

**Command:** *pydcop orchestrator*

**Tutorial:** *Deploying on several machines*

#### Options

**-n <names> / --names <names>** The names of the agent(s). Notice that this needs to match the name of the agents expected by the orchestrator.

**--orchestrator <ip[:<port>]>** The address of the orchestrator as <ip>:<port> where the port is optional and defaults to 9000.

**--address <ip_address>** Optional IP address the agent will listen on. If not given we try to use the primary IP address.

**-p <start_port> / --port <start_port>** The port on which the agent will listen for messages. If several agents names are started (when giving several names) this port is used for the first agent and increment for each subsequent agent. Defaults to 9001

**-i <start_uiport> / --uiport <start_uiport>** The port on which the ui-server will be listening (same behavior as `--port` when starting several agents). If not given, no ui-server will be started for these/this agent(s).

**--restart** When setting this flag, agent(s) will restarted when when they have all stopped. Useful when running *pydcop agent* as daemon on a remote machine.

**--delay <delay>** An optional delay between message delivery, in second. This delay only applies to algorithm's messages and is useful when you want to observe (for example with the GUI) the behavior of the algorithm at runtime.

Note that the agent's port defaults to 9001 to avoid conflicts with the orchestrator, whose port defaults to 9000.

### Examples

Running a single agent on port 9001, with an ui-server on port 10001:

```
pydcop -v 3 agent -n a1 --orchestrator 127.0.0.1:9000 --uiport 10001
```

Running 5 agents, listening on port 9011 - 9016 (without ui-server):

```
pydcop -v 3 agent -n a1 a2 a3 a4 a5 -p 9011 --orchestrator 127.0.0.1:9000
```

## pydcop orchestrator

`pydcop orchestrator` runs an orchestrator.

### Synopsis

```
pydcop orchestrator --algo <algo> [--algo_params <params>]
                    --distribution <distribution>
                    [--address <ip_addr>] [--port <port>]
                    [--uiport <uiport>]
                    [--collect_on <collect_mode>] [--period <p>]
                    [--run_metrics <file>]
                    [--end_metrics <file>]
                    <dcop_files>
```

### Description

Runs an orchestrator, which waits for agents, deploys on them the computations required to solve the DCOP with the requested algorithm and collects selected values from agents. Agents must be run separately using the `agent` command (see. *pydcop agent*).

The `orchestrator` command support the global `--timeout` argument and can also be stopped using `CTRL+C`.

When the orchestrator stops, it request all agents to stop and displays the current DCOP solution (with associated cost) in yaml.

**See also:**

**Commands:** *pydcop agent*, *pydcop solve*

**Tutorials:** *Analysing results* and *Deploying on several machines*

### Output

This commands outputs the end results of the solve process. A detailed description of this output is described in the *Analysing results* tutorial.

### Options

**--algo <dcop_algorithm> / -a <dcop_algorithm>** Name of the dcop algorithm, e.g. 'maxsum', 'dpop', 'dsa', etc.

**--algo_params <params> / -p <params>** Optional parameter for the DCOP algorithm, given as string `name:value`. This option may be used multiple times to set several parameters. Available parameters depend on the algorithm, check *algorithms documentation*.

**--distribution <distribution> / -d <distribution>** Either a *distribution algorithm* (`oneagent`, `adhoc`, `ilp_fgdp`, etc.) or the path to a yaml file containing the distribution. If not given, `oneagent` is used.

**--collect_on <collect_mode> / -c** Metric collection mode, one of `value_change`, `cycle_change`, `period`. See *Analysing results* for details.

**--period <p>** When using `--collect_on period`, the period in second for metrics collection. See *Analysing results* for details.

**--run_metrics <file>** File to store store metrics. See *Analysing results* for details.

**--end_metrics <file>** End metrics (i.e. when the solve process stops) will be appended to this file (in csv).

**--address <ip_address>** Optional IP address the orchestrator will listen on. If not given we try to use the primary IP address.

**--port <port>** Optional port the orchestrator will listen on. If not given we try to use port 9000.

**--uiport <port>** Optional port the orchestrator's ui-server (only needed when using the GUI). If not given no ui-server is started.

**<dcop_files>** One or several paths to the files containing the dcop. If several paths are given, their content is concatenated as used a the *yaml definition* for the DCOP.

### Examples

Running an orchestrator for 5 seconds (on default IP and port), to solve a graph coloring DCOP with `maxsum`. Computations are distributed using the `adhoc` algorithm:

```
pydcop --timeout 5 orchestrator -a maxsum -d adhoc graph_coloring.yaml
```

Running an orchestrator that collects metrics every 0.2 second and run the *MGM algorithm* on agents for 20 cycles:

```
pydcop -v 3 orchestrator --algo mgm --algo_param stop_cycle:20 \
                         --collect_on period --period 0.2 \
                         --run_metrics ./orch_metrics_period.csv \
                         --address 192.168.1.2 --port 10000 \
                         graph_coloring_3agts.yaml
```

### pydcop run

Running a (dynamic) DCOP

### Synopsis

```
pydcop run --algo <algo> [--algo_params <params>]
             [--distribution <distribution>]
             [--replication_method <replication method>]
             [--ktarget <resiliency_level>]
             [--mode <mode>]
             [--collect_on <collect_mode>]
             [--period <p>]
             [--run_metrics <file>]
             [--end_metrics <file>]
             --scenario <scenario_file>
             <dcop_files>
```

### Description

The `run` command run a dcop, it is generally used for dynamic dcop where various events can occur during the life of the system.

Most options are basically the same than the options of the *pydcop solve* command. The main differences are the optional options for resilent DCOP : `--ktarget` and `--replication method` and the scenario that contains events. See *Scenario file format* for information on the scenario file format.

When using the `run` command, you should use the global `--timeout` option. Note that the `--timeout` is used as a timeout for the solve process only. Bootstrapping the system and gathering metrics take additional time, which is not accounted for in the timeout. This means that the run command may take more time to return than the time set with the global `--timeout` option.

You can always stop the process manually with `CTRL+C`. Here again, the system may take a few seconds to stop.

**See also:**

**Commands:** *pydcop solve*, *pydcop distribute*

**Tutorials:** *Analysing results* and *Dynamic DCOPs*

### Options

**`--algo <dcop_algorithm>` / `-a <dcop_algorithm>`** Name of the dcop algorithm, e.g. 'maxsum', 'dpop', 'dsa', etc.

**`--algo_params <params>` / `-p <params>`** Optional parameter for the DCOP algorithm, given as string `name:value`. This option may be used multiple times to set several parameters. Available parameters depend on the algorithm, check *algorithms documentation*.

**`--distribution <distribution>` / `-d <distribution>`** Either a *distribution algorithm* (`oneagent`, `adhoc`, `ilp_fgdp`, etc.) or the path to a yaml file containing the distribution (see *yaml format*). If not given, `oneagent` is used.

**`--mode <mode>` / `-m`** Indicated if agents must be run as threads (default) or processes. either `thread` or `process`

**--collect_on <collect_mode> / -c** Metric collection mode, one of `value_change`, `cycle_change`, `period`. See *Analysing results* for details.

**--period <p>** When using `--collect_on period`, the period in second for metrics collection. See *Analysing results* for details.

**--run_metrics <file>** File to store store metrics. See *Analysing results* for details.

**--replication_method <replication method>** Optional replication method. Defaults to `replication method`, which is the only replication method currently implemented in pyDCOP.

**--ktarget <resiliency_level>** Optional replication level (aka number of replicas for each computation). Defaults to 3

**--scenario <scenario_files>** Path to the files containing the scenario. *yaml definition* for the format.

**<dcop_files>** One or several paths to the files containing the dcop. If several paths are given, their content is concatenated as used a the *yaml definition* for the DCOP.

### Examples

Runnig the DCOP from the file `dcop.yaml`, using the initial ditribution from `dist.yaml`

```
pydcop -v 2 run --algo dsa  \
                --distribution dist.yaml \
                --scenario scenario.yaml \
                --collect_on period \
                --period 1 \
                --run_metrics run_dcop.csv \
                dcop.yaml
```

## pydcop solve

`pydcop solve` solves a static DCOP by running locally a set of agents.

### Synopsis

```
pydcop solve --algo <algo> [--algo_params <params>]
             [--distribution <distribution>]
             [--mode <mode>]
             [--collect_on <collect_mode>]
             [--period <p>]
             [--run_metrics <file>]
             [--end_metrics <file>]
             [--delay <delay>]
             [--uiport <port>]
             <dcop_files>
```

### Description

The `solve` command is a shorthand for the *agent* and *orchestrator* commands. It solves a DCOP on the local machine, automatically creating all the agents as specified in the dcop definitions and the orchestrator (required for bootstrapping and collecting metrics and results).

When using `solve` all agents run on the same machine. Depending on the `--mode` parameter, agents will be created as threads (lightweight) or as process (heavier, but better parallelism on a multi-core cpu). Using `--mode thread` (the default) agents communicate in memory (without network), which scales easily to more than 100 agents.

### Notes

Depending on the DCOP algorithm, the solve process may or may not stop automatically. For example, *DPOP* has a clear termination condition and the command will return once this condition is reached. On the other hand, some other algorithm like *MaxSum* have no clear termination condition.

Some algorithm have an optional termination condition, which can be passed as an argument to the algorithm. With *MGM* for example, you can use `--algo_params stop_cycle:<cycle_count>`

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

For algorithms with no termination condition, you should use the global `--timeout` option. Note that the `--timeout` is used as a timeout for the solve process only. Bootstrapping the system and gathering metrics take additional time, which is not accounted for in the timeout. This means that the solve command may take more time to return than the time set with the global `--timeout` option.

You can always stop the process manually with `CTRL+C`. Here again, the system may take a few seconds to stop.

**See also:**

**Commands:** *pydcop agent*, *pydcop orchestrator*

**Tutorials:** *Analysing results* and *Deploying on several machines*

### Output

This commands outputs the end results of the solve process. A detailed description of this output is described in the *Analysing results* tutorial.

### Options

**`--algo <dcop_algorithm>` / `-a <dcop_algorithm>`** Name of the dcop algorithm, e.g. 'maxsum', 'dpop', 'dsa', etc.

**`--algo_params <params>` / `-p <params>`** Optional parameter for the DCOP algorithm, given as string `name:value`. This option may be used multiple times to set several parameters. Available parameters depend on the algorithm, check *algorithms documentation*.

**`--distribution <distribution>` / `-d <distribution>`** Either a *distribution algorithm* (`oneagent`, `adhoc`, `ilp_fgdp`, etc.) or the path to a yaml file containing the distribution. (see *yaml format*.) If not given, `oneagent` is used.

**`--mode <mode>` / `-m`** Indicated if agents must be run as threads (default) or processes. either `thread` or `process`

**--collect_on <collect_mode> / -c** Metric collection mode, one of `value_change`, `cycle_change`, `period`. See *Analysing results* for details.

**--period <p>** When using `--collect_on period`, the period in second for metrics collection. See *Analysing results* for details.

**--run_metrics <file>** Path to a file or file name. Run-time metrics will be written to that file (csv format). If the value is a path, the directory will be created if it does not exist. Otherwise the file will be created in the current directory.

**--end_metrics <file>** Path to a file or file name. Result's metrics will be appended to that file (csv format). If the value is a path, the directory will be created if it does not exist. Otherwise the file will be created in the current directory.

**--delay <delay>** An optional delay between message delivery, in second. This delay only applies to algorithm's messages and is useful when you want to observe (for example with the GUI) the behavior of the algorithm at runtime.

**--uiport** The port on which the ui-server will be listening. This port is used for the orchestrator and incremented for each following agent. If not given, no ui-server will be started for any agent.

**<dcop_files>** One or several paths to the files containing the dcop. If several paths are given, their content is concatenated as used a the *yaml definition* for the DCOP.

### Examples

The simplest form is to simply specify an algorithm and a dcop yaml file. Beware that, depending on the algorithm, this command may never return and need to be stopped with CTRL+C:

```
pydcop solve --algo maxsum  graph_coloring1.yaml
pydcop -t 5 solve --algo maxsum  graph_coloring1.yaml
```

Solving with MGM, with two algorithm parameter and a log configuration file:

```
pydcop --log log.conf solve --algo mgm --algo_params stop_cycle:20 \
                            --algo_params break_mode:random  \
                            graph_coloring.yaml \
```

## pydcop graph

`pydcop graph` outputs some metrics for a graph model for a DCOP.

### Synopsis

```
pydcop graph --graph <graph_model> <dcop_files>
```

### Description

Outputs some metrics for a graph model for a DCOP:

- constraints_count
- variables_count
- density
- edges_count
- nodes_count

### Options

**--graph <graph_model> / -g <graph_model>** The computation graph model, one of `factor_graph`, `pseudotree`, `constraints_hypergraph` (see. *DCOP graph models*) The set of computation to distribute depends on the graph model used to represent the DCOP.

**--display** Display a graphical representation of the constraints graph using networkx and matplotlib.

**<dcop-files>** One or several paths to the files containing the dcop. If several paths are given, their content is concatenated as used a the yaml definition for the DCOP.

### Example

```
pydcop graph --graph factor_graph graph_coloring1.yaml
```

Example output:

```
constraints_count: 2
density: 0.4
edges_count: 4
nodes_count: 5
status: OK
variables_count: 3
```

### pydcop replica_dist

Distributing computation replicas

### Synopsis

```
pydcop orchestrator
```

### Description

To distribute computations' replicas, one need :

- a computation graph => we need to known the list of computations and which computation is communicating with which (edges in the computation graph)
- computations distribution => we could pass a distribution method and compute it or directly pass a distribution file ( agent -> [computations])
- computations weight ans msg load => these depends on the dcop algorithm we are going to use
- route costs & agents preferences => these are given in the dcop definition yaml file

### Options

TODO

### Examples

Passing the computation distribution:

```
pydcop replica_dist -r dist_ucs -k 3
                    -a dsa --distribution dist_graphcoloring.yml
                    graph_coloring_10_4_15_0.1.yml
```

## pydcop distribute

`pydcop distribute` distributes a the computations for a DCOP over a set of agents.

### Synopsis

```
pydcop distribute --distribution <distribution_method>
                  [--cost <distribution_method_for cost>]
                  [--graph <graph_model>]
                  [--algo <dcop_algorithm>] <dcop-files>
```

### Description

Distributes the computation used to solve a DCOP.

The distribution obtained is written in yaml on standard output. It can also be written into a file by using the `--output` global option. The output file can be used as an input for several commands that accept a distribution (e.g. *orchestrator*, *solve*, *run*)

**See also:**

*DCOP graph models*, *DCOP computation distribution*

### Options

**--distribution <distribution_method> / -d <distribution_method>** The distribution algorithm (`oneagent`, `adhoc`, `ilp_fgdp`, etc., see *DCOP computation distribution*).

```
--cost <distribution_method_for_cost>
```

A distribution method that can be used to evaluate the cost of a distribution. If not given, defaults to `<distribution_method>`. If the distribution method does not define cost, a cost None will be returned in the command output.

**--algo <dcop_algorithm> / -a <dcop_algorithm>** The (optional) algorithm whose computations will be distributed. It is needed when the distribution depends on the computation's characteristics (which depend on the algorithm). For example when the distribution is based on computations footprint of communication load, the dcop algorithm is needed.

**--graph <graph_model> / -g <graph_model>** The (optional) computation graph model, one of `factor_graph`, `pseudotree`, `constraints_hypergraph` (see. *DCOP graph models*) The set of computation to distribute depends on the graph model used to represent the DCOP. When the `--algo` option is used, it is not required as the graph model can be deduced from the DCOP algorithm.

**<dcop-files>** One or several paths to the files containing the dcop. If several paths are given, their content is concatenated as used a the yaml definition for the DCOP.

### Examples

Distributing a DCOP for dsa, hence modeled as a constraints graph, using the `ilp_compref` distribution method:

```
pydcop distribute -d ilp_compref -a dsa \
                  graph_coloring_10_4_15_0.1_capa_costs.yml
```

Distributing a DCOP modelled as a factor graph. The DCOP algorithm is not required here as the `oneagent` distribution algorithm does not depends on the computation's characteristics (as it simply assign one computation to each agent):

```
pydcop distribute --graph factor_graph \
                  --dist oneagent graph_coloring1.yaml
```

The following command gives the same result. Here, we can deduce the required graph model, as maxsum works on a factor graph:

```
pydcop distribute --algo maxsum \
                  --dist oneagent graph_coloring1.yaml
```

Example output:

```
cost: 0
distribution:
  a1: [v3]
  a2: [diff_1_2]
  a3: [diff_2_3]
  a4: [v1]
  a5: [v2]
inputs:
  algo: null
  dcop: [tests/instances/graph_coloring1.yaml]
```

(continues on next page)

(continued from previous page)

```
  dist_algo: oneagent
  graph: factor_graph
```

pyDCOP command line interface can be called either using `dcop_cli.py` or `pydcop` (which is simply a shell script pointing to `dcop_cli.py`). For example, the two following commands are strictly equivalent:

```
dcop_cli.py --version
pydcop --version
```

pyDCOP command line script works with an 'command' concept, similar to `git` (e.g. in `git commit`, `commit` is a 'command' for the `git` cli). Each action defines its own arguments, which must be given after the command name and are documented in their respective page. Additionally, some options apply to many commands and must be given **before** the command, for example in the following `-t` and `-v` are **global options** and `--algo` is an option of the `solve` command:

```
pydcop -t 5 -v 3 solve --algo maxsum  graph_coloring.yaml
```

pydcop supports the following global options:

```
pydcop [--version] [--timeout <timeout>] [--verbosity <level>]
       [--log <log_conf_file>]
```

**`--version`** Outputs pydcop version and exits.

**`--timeout <timeout>` / `-t <timeout>`** Set a global timeout (in seconds) for the command.

**`--output <output_file>`** Write the command's output to a file instead of std out.

**`-verbose <level>` / `-v <level>`** Set verbosity level (0-3). Defaults to level 0, which should be used when you need a parsable output as it only logs errors and the output is only the yaml formatted result of the command.

**`--log <long_conf_file>`** Log configuration file. Can be used instead of `-verbose` for precise control over log (filtering, output to several files, etc.). This file uses the standard python log configuration file format . The following sample file can be used as a starting point to build your own custom log configuration : `log.conf`.

Additionally the `--help` / `-h` option can always be used both as a global option and as a command option. Calling `pydcop --help` outputs a general help for pyDCOP command line interface, with a list of available commands. `pydcop <command> --help` outputs help for this specific command.

### 3.3.2 pyDCOP file formats

#### DCOP file format

A DCOP definition is written in yaml-formatted files and must contains definitions for:

- the objective (min or max)
- variables , and their domains
- constraints, which can be given as intentional or extensional constraints
- agents

It can also contains the following optional elements:

- agents capacity, which is used by some *computations distribution*, mechanisms.
- route and hosting costs, also used for *computations distribution*.

See this sample annotated file for a quick description of the format: `dcop_format.yml`.

---

**Note:** pyDCOP command line interface accepts DCOP given in several files, which will be concatenated before parsing the yaml (see cli the command options, for example *solve*).

This can be useful, for example when when you want to run the same of variables and constraints with different sets of agents ; you can simply write a single file for the variables and constraints and one file for each set of agents.

---

### Distribution file format

In order to run a DCOP, one must also specify which agent is responsible for which variable (or, more generally, which agent will host which computation, see. *computations distribution*). In pyDCOP, we call this mapping a **distribution** and it can be described with a yaml file.

Yaml distribution files simply contains a map, where each key is an agent name and the corresponding value a list of computations hosted on this agent:

```
distribution:
  a0: []
  a1: [v1, v2]
  a2: []
  a3: [v2, v3]
```

See this file for an sample distribution file : `dist_format.yml`.

---

**Note:** pyDCOP command line interface accepts either a yaml distribution file or the name of a distribution algorithm. In the latter case, the distribution will be automatically generated for the DCOP using the requested method.

---

### Replication file format

When using a resilient DCOP, computations are replicated on several agents. This replica distribution is generated using the *replica_dist* command and can also be given as a yaml file.

See this file for an replica distribution file : `replica_dist_format.yml`.

### Scenario file format

To run a dynamic DCOP, you must also describe the events happening to the DCOP (agents leaving or entering the system, change of value for an non-decision variable, etc.). This is done is a scenario file, in yaml.

See this file for an sample scenario file : `scenario_format.yml`.

### 3.3.3 pyDCOP API

TODO

### 3.3.4 Provisioning pyDCOP on many machines

When developing on pyDCOP, if you want to host your agent on several different machines, you generally need to deploy your changes often in order to test them. This can be a pain if you do it manually (copying pyDCOP's new development version on all machines, installing it, etc. ) and is error prone. For this reasons it is advised to automate this deployment. We provide an ansible playbook for this, along with some advices.

#### General principle

In the following sections, we use the term **controller** to name the computer that is used to initiate and drive the installation of pyDCOP on other machines, which are called **agent-machine** (as they will be used to run pyDCOP agents).

When provisioning a set of machines, the **controller** will contacts each **agent-machines** (using ssh) and make sure pyDCOP and all its dependencies are installed on it.

This guide uses ansible (tested with version 2.5.3) , thus you must install ansible on the controller, which requires python. If python is already installed on your controller machine you can simply use:

```
sudo pip install ansible
```

See The official installation guide for more details

**Warning:** Using windows for the Ansible controller is not officially supported. However you can run it under the Windows Subsystem for Linux (WSL) or with Cygwin. Another option is to use a linux virtual machine as a controller.

#### Installing without internet connection

In many cases (tutorials, lectures, demos) it is necessary to install pyDCOP (including its dependencies ) on machines that may not have internet access. For this purpose, we recommend using *cache services* on the controller, that will provide the necessary packages to agent-machines even though the internet repositories cannot be reached.

To be sure to have all required packages cached on the controller machine, you must install at least one agent machine while the controller has internet access. The controller will then cache locally all packages used during the installation and you will later be able to deploy new agent-machines with no internet access, even on the controller.

#### Local cache for apt

Some system packages (e.g. `glpk` ) need to be installed with apt-get, we use the local cache apt repository ACNG, hosted on the controller computer.

Install on controller:

```
sudo apt-get install apt-cacher-ng
```

Check that ACNG is online: http://localhost:3142/acng-report.html

See the official documentation for more details.

### Local cache for pip

We run devpi, which acts as a proxy-cache, on the controller:

```
pip install -q -U devpi-server
pip install -q -U devpi-web
devpi-server --start --init
devpi-server --stop
```

Note: the *devpi* server must be started every time we need to need to perform a deployment. Once finished you may stop it:

```
devpi-server --start --host=0.0.0.0
...
devpi-server --stop
```

You can check that devpi is online : http://127.0.0.1:3141/

### Agent-machines

You can use any computer as an agent-machine, however we recommend using linux-based machines (Macs should work and windows can work too, but this guide might require some tweaking). The main requirement for ansible is that your agent-machine must have an ssh server. The command `sudo` must be available and your user must be able to use (i.e. be in the sudoer list, simply adding it to the sudo group usually does the trick `adduser dcop sudo`)

As we generally do not have lots of machines available, agents-machines are generally implemented using Virtual Machines (VM) or cheap single-board computers like the Raspberry Pi.

### Virtual Machines

Using Virtual Machines (VM) allows you to run several agent-machines on a single server. We use VirtualBox to run our VM, but you can use any hypervisor.

Create a base linux (debian or ubuntu is recommended, as the playbook assumes packages can be installed with `apt`) and duplicate it to create as many VM as required. Do not forget to enable the ssh server on your VM !

Make sure you configure the network mode on the hypervisor in a way that allows connecting to servers running on the VMs. The details depends on your hypervisor, with VirtualBox you generally want to use the "bridged adaptater" mode (and not NAT, which the default).

### Raspberry Pis

Single board computers like the Raspberry Pi are very good candidates for agent-machines : they are cheap, run linux and are powerfull enough for pyDCOP.

You should use the standard raspbian distribution. The only modification is to make sure that the ssh server is enabled your Raspberry Pi. This can be done using the `` raspi-config`` utility or by creating a file named `ssh` in the boot partition of the SD card, which can be done easily after preparing the SD card on another computer and is the best option if you run your Pis without keyboards. See the official documentation for more details

#### Configuration

Once you have your agent-machine (and you have noted their IP address), you must edit the ansible host configuration file. An example is provided in pyDCOP's git repository (in `pyDcop/provisioning/ansible/hosts-conf.yaml`)

> **Warning:** Be careful, agent names given in the host configuration file **must match** the names given in the dcop yaml definition, and their IP address must be set to the IP address assigned to the corresponding VMs or physical machine.

TODO: explain how to use avahi to make your agent-machine automatically discoverable.

#### Deploying with ansible

Once you have properly configured your host file, you can simply run ansible-playbook to apply the operations on all your agent-machines. The playbook is in `pyDcop/provisioning/ansible/`:

```
ansible-playbook --inventory hosts-conf.yaml pydcop-playbook.yml
```

If the process fails on some machines, you can safely restart it as ansible keeps track of the progress.

You can also run the playbook on a subset of the hosts defined in your configuration file, by using the `--limit` option:

    ansible-playbook –inventory hosts-conf.yaml –limit a2 pydcop-playbook.yml

See the *tutorials* for an introduction to the most common use-cases and commands.

## 3.4 Concepts & Implementation

### 3.4.1 DCOP graph models

When solving a DCOP, The first step performed by pyDCOP is to build a graph of computations. This **computation graph** must not be confused with the constraints graph ; while they may sometime be similar. The constraints graph is strictly a graph representation of the Constraints Optimization Problem (COP) while the **computation graph** depends on the method (aka algorithm) used to solve this problem.

A computation graph is a graph where vertices represent **computations** and edges represent **communication** between these computations. Computation send and receive messages one with another, along the edges of the graph. Computations are defined as the basic unit of work needed when solving a DCOP, their exact definition depends on the algorithm used. Most algorithms define computations for the decision variables of the DCOP, but some algorithm can define computations for constraints as well, or even groups of variables.

Pydcop defines at the moment 3 kinds of computation graphs:

- Constraints hyper-graph
- Pseudo-tree (aka DFS Tree)
- Factor graph

When solving a DCOP, as each algorithm require a specific type of graph, pydcop can automatically infer the computation graph model from the algorithm you're using.

### Constraints hyper-graph

This is the most straightforward computation graph and is used with algorithms that define one computation for each variable. In a constraints graph vertices (i.e. computations) directly maps to the variables of the DCOP and edges maps to constraints. As a classical constraint graph can only represents binary constraints, pyDCOP uses an hyper-graph, where hyper-edges can represent n-ary constraints.

The kind of graph is used by algorithms like *MGM*, *DSA*, etc.

### Factor graph

A factor graph is an undirected bipartite graph in which vertices represent variables and constraints (called factors), and an edge exists between a variable and a constraint if the variable is in the scope of the constraint.

Factors graph are used for algorithms that define computations for variables and for constraints. This is typically the caase of *MaxSum* and most GDL-based algorithms.

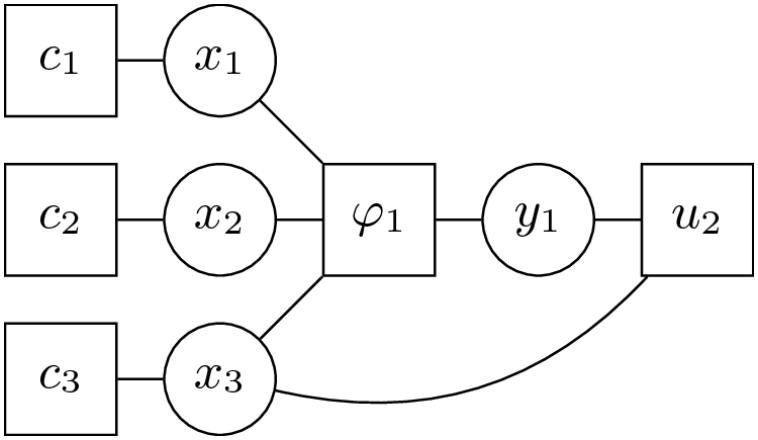


Fig. 3.3: An example of a Factor graph (from [RPR16])

### DFS tree

DFS trees are a subclass of pseudotrees, built a depth-first traversal of the constraint graph (where vertices represent variables and edges represent constraints). In addition to the parent/children edges of the tree, they contains pseudo-parent/ pseudo-children edges, which correspond to the edges (aka constraints) of the orginal graph that would othrwise not be represented in a simple tree.

The only algorithm currently implemented in pyDCOP that uses a DFS tree computation graph is *DPOP*

### Implementing a new graph model

A module for a computation graph type typically contains

- class(es) representing the nodes of the graph (i.e. the computation), extending ComputationNode
- class representing the edges (extending Link)
- a class representing the graph
- a (mandatory) method to build a computation graph from a Dcop object :

  def build_computation_graph(dcop: DCOP)-> ComputationPseudoTree:

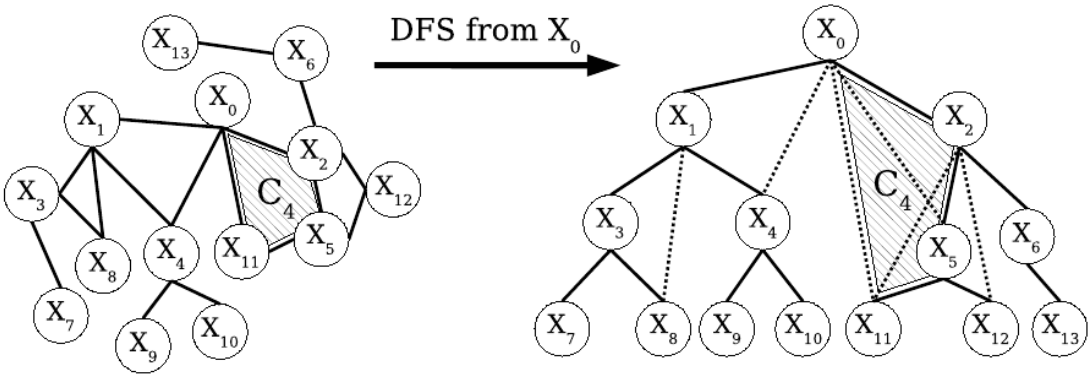


Fig. 3.4: An example of a DFS tree representing a constraint graph (from [Pet07])

### 3.4.2 DCOP computation distribution

Before running the DCOP, the computation must be deployed on agents. We name **distribution** the task of assigning each computation to one agent, which will be responsible for hosting and running the computation.

When you look at the standard DCOP definition, $\langle \mathcal{A}, \mathcal{X}, \mathcal{D}, \mathcal{C}, \mu \rangle$, pyDCOP's **distribution** can be seen as a generalization of the mapping function $\mu : \mathcal{X} \rightarrow \mathcal{A}$ used to assign variables to agents.

In classical DCOP approaches, there is exactly one agent for each variable and most DCOP algorithms define one computation for each variable. In that case, the distribution of these computations is of course trivial. The *oneagent* distribution replicates this traditional hypothesis in pyDCOP and might be enough if you do not care about distribution issues and simply want to develop or benchmark classical DCOP algorithms.

However, some DCOP algorithms (for example MaxSum) not only define computations for variable, but also for constraints. These computations then form a *factor graph* like the one on Fig. 3.5:

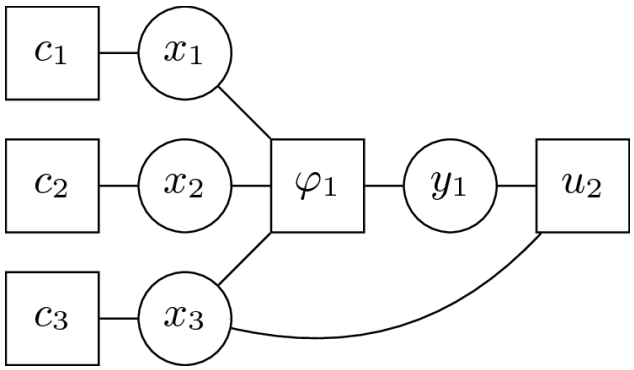


Fig. 3.5: Sample factor graph with 4 variables and 5 constraints

In this situation, one must also assign factor-computation to agent, and there is no obvious mapping. Fig. 3.6: is a possible distribution of this factor graph on 3 agents but is obvously not the only possible one.

And even with algorithms that only define computations for variables, the standard assumptions do not hold on many real world problems. Agents typically maps to physical computers or devices and the number of these devices is not necessarily equal to the number of decision variables in the DCOP. Moreover, some variables have a physical link to devices (in the sense, for example, that they model an action or decision of this particular device) while some other variables might simply be used to model an abstract concept in the problem and have no real relationship with physical devices.

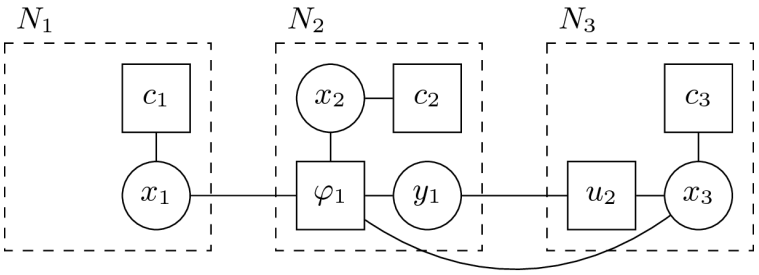


Fig. 3.6: One possible distribution of the factor graph on agents

Finally the placement of computations on the agents has an important impact on the performance characteristics of the global system: some distributions may improve response time, some other may favor communication load between nodes and some other may be better for other criteria like QoS or running cost.

Formally, once the definition of optimality has been defined for a specific problem, finding an optimal mapping is an optimization problem by itself, which can be mapped to graph partitioning and this typically falls under the category of NP-hard problems.

For all these reasons, the distribution of computations on agents is an interesting topic, which is implemented in pyDCOP through distribution methods. See [RPR16] [RPR17] and [RPR18] for background and details on the deployment and distribution of computations in a DCOP.

pyDCOP currently implements several distribution methods, you can find the full list in the *reference documentation*.

### 3.4.3 Resilient DCOP

In most articles on DCOPs, variables and computation are mapped to a static set of agents, which is supposed to persist over the resolution time of the system.

However, as for most distributed systems, DCOP are often used on problems where the environment is dynamic in nature. Agents may enter of leave the system at any time and the system needs to adapt to these changes.

This is typically the case when applying DCOP on real problems like Internet-of-Things (IoT), Ambient intelligence, sensor networks or smart-grid, where computations run on distributed, highly heterogeneous, nodes and where a central coordination might not desirable or even not possible.

In such settings, systems must be able to cope with node additions and failures: when a node stops responding, other nodes in the system must take responsibility and run the orphaned computations. Similarly, when a new node is added in the system, it might be useful to reconsider the distribution of computations in order to take advantage of the newcomer's computational capabilities, as proposed in [RPR17].

Here we use the term **computation**, which represents the basic unit of work needed when solving a DCOP, and generally maps to variables (and factors for some DCOP algorithms) in the DCOP. See the page on *DCOP graph models* for more details.

pyDCOP introduces a notion of a resilient DCOP and proposes mechanisms to cope with these changes in the infrastructure. This mechanisms are described in more details in [RPR18].

#### Initial distribution

As explained in *DCOP computation distribution*, the computations of the DCOP need to be distributed on the agent / devices at startup. In many cases, this initial distribution can computed centrally as part of a bootstrapping process.

#### Computation Replication

One pre-requisite to resilience is to still have access to the definition of every computation after a failure. One approach is to keep replicas (copies of definitions) of each computation on different agents. Provided that k replicas are placed on k different agents, no matter the subset of up to k agents that fails there will always be at least one replica left after the failure. This approach is classically found in distributed database systems

The ideal placement of these replicas is far from trivial and is an optimization problem in itself, which depends strongly on the nature of the real problem solved by the DCOP. See [RPR18] for an example.

pyDCOP currently proposes one distributed replication algorithm, called **DRPM**, which is a distributed version of iterative lengthening (uniform cost search based on path costs).

#### Distribution reparation

Given a mechanism to replicate computations, the DCOP distribution can be repaired when an agents fails or is removed from the system. This reparation process must decide which agent will host the orphaned computations (aka the computations that were hosted on the departed agent) and should ideally be decentralized.

Here again, several approach can be designed to handle such reparation, like those presented in [RPR17] and [RPR18], which are currently implemented in pyDCOP.

### 3.4.4 DCOP algorithm Implementation

**Note:** This document build upon the tutorial *Implementing a DCOP algorithm*, you should follow it before reading this.

By providing all the infrastructure, pyDCOP makes it easier to implement a new DCOP algorithm ; you only have one python module to implement, with only one mandatory class.

To implement an algorithm you must:

- create a python module
- define one or several `Message`
- implement your logic in one or several `DcopComputation` class

Optionally, you may also

- declare the algorithm's parameters
- implement some other methods used for some distribution methods

### Module

An algorithm must be defined in it's own module in *pydcop.algorithms*. For example, `dsa` is implemented in the module *pydcop.algorithms.dsa*. The name of the module is the name you will use when running your algorithm with the `pydcop` command line interface (`-a` or `--algo` parameter). For example, for a new algorithm named `my_algorithm`, you simply create a file `my_algorithm.py` in `pydcop/algorithms`. You will then be able to use this algorithm to *solve* a DCOP with the following command:

```
pydcop solve --algo my_algorithm [...]
```

The module of your algorithm **must** also have an attribute named `GRAPH_TYPE` which contains the name of the computation graph type used. Available computation graph types are `'factor_graph'`, `'pseudo_tree'` and `'constraints_hypergraph'`, other could be defined in the future.

For example, in `dsa.py`:

```
GRAPH_TYPE = 'constraints_hypergraph'
```

### Messages

DCOP algorithms are *message passing* algorithms: they work by sending messages to each other. You must define the message(s) used by your algorithm. The easiest approach is to use the *message_type()* class factory method to define your message(s).

For example, the following will define a message type `MyMessage`, with two fields `foo` and `bar`:

```
MyMessage = message_type('MyMessage', ['foo', 'bar'])
```

You can then use `MyMessage` like any class:

```
>>> msg = MyMessage(foo=42, bar=21)
>>> msg.foo
42
>>> msg.type
'MyMessage'
```

You can also subclass *pydcop.infrastructure.computations.Message*, which is more verbose but can be convenient if you want to use python's type annotations:

```
class MyMessage(Message):
    def __init__(self, foot: int, bar; float):
    super().__init__('my_message', None)
    self._foo = foo
    self._bar = bar

    @property
    def foo(self) -> int:
        return self._foo

    @property
    def bar(self) -> float:
        return self._bar
```

In any case, your messages **must** use the `pydcop.utils.simple_repr.SimpleRepr` mixin (`Message` already extends it) for your message to be serializable. When subclassing *Message* or using *message_type()* this is done automatically. This is necessary when running the agents in different processes, as messages will be sent over the network.

### Computation

An algorithms consists in one or several `DcopComputation` class. Most algorithms have one single type of computation, which is responsible for selecting the value for a single variable. In this case you must subclass `VariableComputation`, which provides some convenient methods for value selection.

For more complex algorithm, you can define several computations (with pyDCOP, your algorithm can have as many kind of computation as you want), look at MaxSum's implementation for an example (*MaxSum* has two kind of computations, for *Factor* and *Variable*).

### Receiving messages

At runtime, an instance of a computation is deployed on an agent, which notifies it when receiving a message. The computation then processes the message and, if necessary, emits new messages for other computations.

For each message type, you must declare a handler method using the `register()` decorator:

```
@register("my_message_type")
def _on_my_message(self, sender_name, msg, t):
    # handle message of type 'my_message'
    # sender_name is the name of the computation that sent the message
    # t is the time the message was received by the agent.
```

### Sending messages

When sending messages, a computation never needs to care about the agent hosting the target computations : all message routing and delivery is taken care of by the agent and communication infrastructure. Messages are sent by calling `self.post_msg`:

```
self.post_msg(target_computation_name, message_object)
```

You can also send a message to all neighbors by using `self.post_to_all_neighbors`.

### Selecting a value

In your computation, when selecting a value for a variable, you **must** call `self.value_selection` with the value and the associated local cost. This is allows pyDcop to monitor value selection on each agent and extract the final assignment:

```
self.value_selection(self._v.initial_value, local_cost)
```

The `local_cost` is the cost as seen from this variable.

### Cycles

Each your algorithm has a concept of cycle (i.e. it works in sycnhronized steps), you should call `self.new_cycle()` when you start a new cycle.

### Terminating the algorithm

TODO

### Argument Parameters

If the algorithm supports parameters, you must give a definition of these parameters in your module, by defining a variable named `algo_params` that contains a list of `AlgoParameterDef`.

See for example in mgm implementation:

```
algo_params = [
    AlgoParameterDef('break_mode', 'str', ['lexic', 'random'], 'lexic'),
    AlgoParameterDef('stop_cycle', 'int', None, None),
]
```

These definitions will be automatically used (with `pydcop.algorithms.prepare_algo_params()`) to check parameters for validity and add default values.

An `AlgoritmDef` instance populated with the parsed parameters will be passed to your `__init__` method, you can then use it to pass these parameters to the computation instance.

### Builder method

TODO

### Distribution and deployment

Your module must also provide a a few predefined utility methods, used to build and deploy your algorithm, and may define some optional methods, used for deployment and distribution.

Most distribution methods require the following two methods. These methods are generally required for a correct distribution of the computations on agents, but if you only want to use *oneagent* distribution (or simply during development) you can simply return 0:

```
def computation_memory(computation: ComputationNode, links):
"""
  This method must return the memory footprint for the given computation
  from the graph.
"""
```

```
def communication_load(link: Link):
"""
This method must return the communication load for this link in the
computation graph.
"""
```

When deploying the computation, concrete `MessagePassingComputation` objects must be instantiated on their assigned agent. For this, an algorithm module **must** also provide a factory method to build computation object:

```
def build_computation(node: ComputationNode, links: Iterable[Link], algo:␣
↪AlgorithmDef)-> MessagePassingComputation:
"""
Build a computation instance for a given algorithm (and parameters)
"""
```

#### Computations's footprint

TODO

#### Communication load

TODO

### 3.4.5 Reference

#### pydcop.infrastructure.computations

The `computations` module contains base classes for computation and message.

You generally need to sub-class classes from this module when implementing your own DCOP algorithm.

**class Message**(*msg_type*, *content=None*)

Base class for messages.

you generally sub-class `Message` to define the message type for a DCOP algorithm. Alternatively you can use *`message_type()`* to create your own message type.

**Parameters**

- **msg_type** (*str*) – the message type ; this will be used to select the correct handler for a message in a DcopComputation instance.
- **content** (*Any*) – optional, usually you sub-class Message and add your own content attributes.

**property size**

Returns the size of the message.

You should overwrite this methods in subclasses, will be used when computing the communication load of an algorithm and by some distribution methods that optimize the distribution of computation for communication load.

**Returns** size

**Return type** int

**property type**

The type of the message.

**Returns** message_type

**Return type** str

**message_type**(*msg_type: str*, *fields: List[str]*)
Class factory method for Messages

This utility method can be used to easily define new Message type without subclassing explicitly (and manually) the Message class. Tt output a class object which subclass Message.

Message instance can be created from the return class type using either keywords arguments or positional arguments (but not both at the same time).

Instances from Message classes created with *message_type* support equality, simple_repr and have a meaningful str representation.

**Parameters**

- **msg_type** (*str*) – The type of the message, this will be return by *msg.type* (see example)
- **fields** (*List[str]*) – The fields in the message

**Returns**

**Return type** A class type that can be used as a message type.

### Example

```
>>> MyMessage = message_type('MyMessage', ['foo', 'bar'])
>>> msg1 = MyMessage(foo=42, bar=21)
>>> msg = MyMessage(42, 21)
>>> msg.foo
42
>>> msg.type
'MyMessage'
>>> msg.size
0
```

**register**(*msg_type: str*)
Decorator for registering message handles in computations.

This decorator is meant to be used to register message handlers in computation class (subclasses of MessagePassingComputation).

A method decorated with *@register('foo')* will be called automatically when the computation receives a message of type *'foo'*.

Message handler must accept 3 parameters : (sender_name, message, time). * sender_name is the name of the computation that sent the message * message is the message it-self, which is an instance of a subclass of *Message* * time is the time the message was received.

**Parameters** **msg_type** (*str*) – the type of message

### Example

```
> class C(MessagePassingComputation):
>     ...
>     @register('msg_on_c')
>     def handler_c(self, s, m, t):
>         print("received messages", m, "from", s)
```

See DsaTuto sample implementation for a complete example.

**class MessagePassingComputation**(*name: str*, **args*, ***kwargs*)

*MessagePassingComputation* is the base class for all computations. It defines the computation lifecycle (*start*, *pause*, *stop*) and can send and receive messages.

When subclassing *MessagePassingComputation*, you can use the *@register* decorator on your subclass methods to register then as handler for a specific kind of message. For example:

```
> class MyComputation(MessagePassingComputation):
>     ...
>     @register('msg_on_c')
>     def handler_c(self, s, m, t):
>         print("received messages", m, "from", s)
```

---

**Note:** A computation is always be hosted and run on an agent, which works with a single thread. This means that its methods do not need to be thread safe as they will always be called sequentially in the same single thread.

---

**Parameters** **name** (*str*) – The name of the computation.

**add_periodic_action**(*period: float*, *cb: Callable*)

Add an action that will be called every *period* seconds.

**Parameters**

- **period** (*float*) – the period, in second
- **cb** (*Callable*) – A callable, with no parameters

**Returns**

**Return type** the callable

**on_pause**(*paused: bool*)

Called when pausing or resuming the computation.

This method is meant to be overwritten in subclasses.

**Parameters** **paused** (*boolean*) – the new pause status. This method is only called is the status has changed

**on_start**()

Called when starting the computation.

This method is meant to be overwritten in subclasses.

**on_stop**()

Called when stopping the computation

This method is meant to be overwritten in subclasses.

**post_msg**(*target: str*, *msg*, *prio: int = None*, *on_error=None*)

Post a message.

### Notes

This method should always be used when sending a message. The computation should never use the *_msg_sender* directly has this would bypass the pause state.

**Parameters**

- **target** (*str*) – the target computation for the message
- **msg** (*an instance of Message*) – the message to send
- **prio** (*int*) – priority level
- **on_error** (*error handling method*) – passed to the messaging component.

**class DcopComputation**(*name*, *comp_def: pydcop.algorithms.ComputationDef*)

Computation representing a DCOP algorithm.

DCOP algorithm computation are like base computation : they work by exchanging messages with each other. The only difference is that a DCOP computation has additional metadata representing the part of the DCOP it as been defined for.

**Parameters**

- **name** (*string*) – the name of the computation
- **comp_def** (*ComputationDef*) – the ComputationDef object contains the information about the dcop computation : algorithm used, neighbors, etc.

### Notes

When subclassing DcopComputation, you **must** declare in __init__ the message handlers for the message's type(s) of your algorithm. For example with the following `_on_my_msg` will be called for each incoming `my-msg` message.

self._msg_handlers['my_msg'] = self._on_my_msg

**footprint**()

The footprint of the computation.

A DCOP computation has a footprint, which represents the amount of memory this computation requires to run. This depends on the algorithm used and thus must be overwritten in subclasses. This footprint is used when distributing computation on agents, to ensure that agents only host computation they can run.

**Returns** the footprint

**Return type** float

**property neighbors**

The neighbors of this computation.

### Notes

This is just a convenience shorthand to the neighbors of the node in the computation node :

my_dcop_computation.computation_def.node.neighbors

**Returns** a list containing the names of the neighbor computations.

**Return type** list of string

**new_cycle**()
For algorithms that have a concept of cycle, you must call this method (in the subclass) at the start of every new cycle. This is used to generate statistics by cycles.

### Notes

You can just ignore this if you do not care about cycles.

**post_to_all_neighbors**(*msg*, *prio: int = None*, *on_error=None*)
Post a message to all neighbors of the computation.

**Parameters**

- **msg** (*an instance of Message*) – the message to send
- **prio** (*int*) – priority level
- **on_error** (*error handling method*) – passed to the messaging component.

**class VariableComputation**(*variable:* pydcop.dcop.objects.Variable, *comp_def:* *pydcop.algorithms.ComputationDef*)
A VariableComputation is a dcop computation that is responsible for selecting the value of a variable.

**See also:**

*DcopComputation*

**property current_value**
Return the value currently selected by the algorithm.

If a computation algorithm does no select a value, override this method to raise an exception. :return:

**random_value_selection**()
Select a random value from the domain of the variable of the VariableComputation.

**value_selection**(*val*, *cost=0*)
When the computation selects a value, it MUST be done by calling this method. This is necessary to be able to automatically monitor value changes. :param val: :param cost:

**property variable**
Variable: The variable this algorithm is responsible for, if any.

### pydcop.algorithms

The `pydcop.algorithms` module contains the implementation of all DCOP algorithm supported by pyDCOP.

### pydcop.algorithms.dba

### DBA : Distributed Breakout Algorithm

The Distributed Breakout algorithm [YH96] is a local-search algorithm, built as a a distributed version of the Breakout algorithm for CSP [Mor93]. It is meant to solve distributed constraints satisfaction problems (and not optimization problems).

See also [WZ03] on using DBA for optimization problems.

### Algorithm Parameters

Our DBA implementation supports two parameters:

- **infinity**: the value used as 'infinity', returned as the cost of a violated constraint (it must map the value used in your dcop definition). Defaults to 10 000
- **max_distance** : an upper bound for the maximum distance between two agents in the graph. Ideally you would use the graph diameter or simply the number of variables in the problem. It is used for termination detection (which in DBA only works is there is a solution to the problem). Defaults to 50

### Example

```
pydcop -t 2 solve -a dba -p infinity:10000 max_distance:3            -d adhoc graph_
↪coloring_csp.yaml
{
  "assignment": {
    "v1": "G",
    "v2": "R",
    "v3": "G"
  },
  "cost": 0,
  "duration": 1.9932785034179688,
  "status": "TIMEOUT"
}
```

### Messages

**class DbaOkMessage**(*value*)

**property size**

Returns the size of the message.

You should overwrite this methods in subclasses, will be used when computing the communication load of an algorithm and by some distribution methods that optimize the distribution of computation for communication load.

**Returns size**

**Return type** int

**class DbaImproveMessage**(*improve*, *current_eval*, *termination_counter*)

**property size**
Returns the size of the message.

You should overwrite this methods in subclasses, will be used when computing the communication load of an algorithm and by some distribution methods that optimize the distribution of computation for communication load.

**Returns** size

**Return type** int

**class DbaEndMessage**

**property size**
Returns the size of the message.

You should overwrite this methods in subclasses, will be used when computing the communication load of an algorithm and by some distribution methods that optimize the distribution of computation for communication load.

**Returns** size

**Return type** int

### Computation

**class DbaComputation**(*variable:* pydcop.dcop.objects.Variable, *constraints: Iterable[pydcop.dcop.relations.RelationProtocol]*, *msg_sender=None*, *mode='min'*, *infinity=10000*, *max_distance=50*, *comp_def=None*)
DBAComputation implements DBA.

See. the following article for a description of original DBA: 'Distributed Breakout Algorithm for Solving Distributed Constraint Satisfaction Problems' (Makoto Yokoo, Katsutoshi Hirayama, 1996)

**compute_eval_value**(*val*, *relations*)
This function compute the evaluation value (the number of violated constraints) regarding the current assignment.

**Parameters**

- **val** (*Any*) – A value for the variable of this object. You can choose any of the definition domain, according to the context in which you use the function.
- **relations** (*list of constraints objects*) – The list of constraints involving the variable of this computation, with the values of other variables set to the values sent by the neighbors

**Returns**

- *The evaluation value for the given assignment and the list*
- *of indices of the violated constraints for this value*

**property neighbors**
The neighbors of this computation.

#### Notes

This is just a convenience shorthand to the neighbors of the node in the computation node :

my_dcop_computation.computation_def.node.neighbors

**Returns** a list containing the names of the neighbor computations.

**Return type** list of string

**on_start**()
Called when starting the computation.

This method is meant to be overwritten in subclasses.

### pydcop.algorithms.dsa

#### DSA : Distributed Stochastic Algorithm

Distributed Stochastic Algorithm [ZWXW05] is a synchronous, stochastic, local search DCOP algorithm.

This is the classical synchronous version of DSA ; at each cycle each variable waits for the value of all its neighbors before computing the potential gain and making a decision. This means that this implementation is not robust to message loss.

#### Algorithm Parameters

**variant** 'A', 'B' or 'C' ; the variant of the algorithm, as defined in [ZWXW05] . Defaults to B

**probability** probability of changing a value. Defaults to 0.7

**stop_cycle** The number of cycle after which the algorithm stops, defaults to *0* If not defined (of equals to *0*), the computation never stops.

#### Example

```
pydcop -t 3 solve -algo dsa  \
  --algo_param stop_cycle:30 \
  --algo_param variant:C \
  --algo_param probability:0.5 \
 -d adhoc graph_coloring_csp.yaml

{
  "assignment": {
    "v1": "G",
    "v2": "R",
    "v3": "G"
  },
  "cost": 0,
  "duration": 1.9972785034179688,
  "status": "TIMEOUT"
}
```

**See also:**

*DSA-tuto*: for a very simple implementation of DSA, made for tutorials.

*A-DSA*: for an asynchronous implementation of DSA.

### pydcop.algorithms.adsa

### A-DSA : Asynchronous Distributed Stochastic Algorithm

ADSA [FM03] is an asynchronous version of DSA, the Distributed Stochastic Algorithm [ZWXW05] ( stochastic, local search DCOP algorithm.) Instead of waiting for its neighbors, each variables periodically determines if it should select a new values, based on the values received from its neighbors.

### Algorithm Parameters

**variant** 'A', 'B' or 'C' ; the variant of the algorithm, as defined in [ZWXW05] . Defaults to B.

**probability** probability of changing a value. Defaults to 0.7.

**period** The period between variables activation, in second. Defaults to 0.5.

### Example

```
pydcop -t 10 solve --algo adsa \
  --algo_param variant:B  \
  --algo_param probability:0.5 \
  --algo_param period:0.2
  graph_coloring_50.yaml
```

**See also:**

*DSA-tuto*: for a very simple implementation of DSA, made for tutorials.

*A-DSA*: for an asynchronous implementation of DSA.

### pydcop.algorithms.dsatuto

This module contains a very simple implementation of DSA, for demonstration purpose.

To keep things as simple as possible, we implemented the bare minimum, and avoided some details you would generally care about:

- no algorithm parameters (threshold, variants, etc.)
- no computation footprint nor message size

### pydcop.algorithms.mgm

### MGM : Maximum Gain Message

### Algorithm Parameters

MGM supports two parameters:

- break_mode
- stop_cycle

### Example

TODO

### pydcop.algorithms.mgm2

### MGM2 : a 2-coordinated DCOP algorithm

Mgm2 algorithm as described in 'Distributed Algorithms for DCOP: A Graphical-Game-Base Approach' (R. Maheswaran, J. Pearce, M. Tambe, 2004)

### pydcop.algorithms.gdba

### GDBA Algorithm

### pydcop.algorithms.maxsum

### MaxSum: Belief-propagation DCOP algorithm

Max-Sum [FRPJ08] is an incomplete inference-based DCOP algorithm.

This is a **synchronous implementation** of Max-Sum, where messages are sent and received in cycles. For an asynchronous implementation, see. *A-Max-Sum*

### Algorithm Parameters

**damping** amount of dumping [0-1]

**damping_nodes** nodes that apply damping to messages: "vars", "factors", "both" or "none"

**stability** stability detection coefficient

**noise** noise level for variable

**start_messages** nodes that initiate messages : "leafs", "leafs_vars", "all"

FIXME: add support for stop_cycle

### Example

```
pydcop solve -algo maxsum  \
  --algo_param stop_cycle:30 \
 -d adhoc graph_coloring_csp.yaml
```

FIXME: add results

**See also:**

*A-Max-Sum*: an asynchronous implementation of Max-Sum.

## pydcop.algorithms.amaxsum

### MaxSum: Belief-propagation DCOP algorithm

Max-Sum [FRPJ08] is an incomplete inference-based DCOP algorithm.

This is a **asynchronous implementation** of Max-Sum, where factors and variable send messages every time they receive a message. For an synchronous implementation, see. *Max-Sum*

### Algorithm Parameters

### Example

```
pydcop solve -algo amaxsum  \
 -d adhoc graph_coloring_csp.yaml
```

FIXME: add results

**See also:**

*Max-Sum*: an synchronous implementation of Max-Sum.

## pydcop.algorithms.dpop

### DPOP: Dynamic Programming Optimization Protocol

Dynamic Programming Optimization Protocol is an optimal, inference-based, dcop algorithm implementing a dynamic programming procedure in a distributed way [PF04].

DPOP works on a Pseudo-tree, which can be built using the *distribute* command (and is automatically built when using the *solve* command).

This algorithm has no parameter.

### Example

```
pydcop -algo dpop graph_coloring_eq.yaml
```

## pydcop.algorithms.syncbb

### SyncBB: Synchronous Branch and Bound

Synchronous Branch and Bound (SBB or SyncBB) [HY97] is a complete search DCOP algorithm that simply simulates Branch and Bound in a distributed environment. It was initially defined in [HY97] for distributed constraints satisfaction problems but is easily extended to optimization problems, as it is done here.

During execution of SyncBB, a *Current Partial Assignment* (CPA) is exchanged as a token among the agents according to the ordering until a solution is found. The algorithm is synchronous and sequential, only the agent holding the CPA can work (all other agents are idle), which make SyncBB slow.

In SyncBB, the CPA is represented as a path, whose elements consist of a variable, the value for that variable and the cost incurred by that value. For example in the path `[ (A, 2, 0), (B, 3, 4) ]`, the variable `A` takes the value 2 and `B` the value 3, which causes a cost of 4. Along with this path, agents also send the currently known upper-bound (notice that some article consider the upper bound to be broadcasted, but that is not necessary as it can be passed in the messages with the path.)

At start, the first agent in the ordering assigns it first value and send the resulting path to the next agent. Each agent extends the path it by adding its value assignment to it, as well as the cost it incurred because of constraints with other assignments appearing in the received path. Whenever reaching the last agent of the ordering, the path contains a full assignment, whose cost can be evaluated and might be used as a new upper-bound.

When the domain of the first agent is exhausted, the last discovered full assignment is reported as the solution (this requires remembering what that assignment was).

Variables and value ordering are given in advance:

- variables are ordered as a simple chain (list)
- values in a variable's domain are also ordered

### Example

```
pydcop solve -a syncbb graph_coloring_tuto.yaml
{
  ...
  "assignment": {
    "v1": "G",
    "v2": "G",
    "v3": "G",
    "v4": "G"
  },
  "cost": 12,
  "cycle": 0,
  "msg_count": 23,
  "msg_size": 0,
  "status": "FINISHED",
  "time": 0.022340279014315456,
  "violation": 0
}
```

### Implementation

**Features**

- Only supports **binary constraints** (it could easily be extended to n-ary conatraints, pull requests are welcome !)
- cycles reporting
- no parameters
- complete: terminates automatically (no need for `timeout` `stop_cycle` parameters.

**Notes**

- The fixed ordering of agents is based on a *ordered_graph* computation graph, which is simply a classical constraints graph with a lexical order on variables.
- The ordering of the values is simply the order used when building the domain of the variables.
- We introduce an extra message to terminate the algorithm at each agent. This messages is sent from the first agent (which is the only agent that can decide termination) and is propagated according to the ordering.
- Agent select their value when they receive a backward message that contains a better bound that the one they currently know. This means that an agent may select different value during execution but is guaranteed to have the value from the solution assignment at termination.
- Although SyncBB is a synchronous algorithm, we do not use the `SynchronousComputationMixin` which would make things more complicated with no benefits. As only one single computation is active at any given time, we can simple call `new_cycle()` everytime a computation handles a message.

### Messages

**SyncBBForwardMessage**
alias of `pydcop.infrastructure.computations.forward`

**SyncBBBackwardMessage**
alias of `pydcop.infrastructure.computations.backward`

**SyncBBTerminateMessage**
alias of `pydcop.infrastructure.computations.terminate`

### Computation

**class SyncBBComputation**(*computation_definition: pydcop.algorithms.ComputationDef*)
Computation for the SyncBB algorithm.

**on_backward_msg**(*sender*, *recv_msg*, *t*) → None
Message handler for backward messages.

**Parameters**

- **sender** (*computation*) – name of the computation that sent the message
- **recv_msg** (*message*) –
- **t** (*timestamp*) –

**on_forward_message**(*sender*, *recv_msg*, *t*) → None
Message handler for forward messages.

**Parameters**

- **sender** (*computation*) – name of the computation that sent the message
- **recv_msg** (*message*) –
- **t** (*timestamp*) –

**on_start**() → None
SyncBB computation startup handler.

At startup, only the first computation in the ordering assigns its first value and send the path to the next computation.

**on_terminate_message**(*sender*, *_*, *t*) → None
Message handler for forward messages.

**Parameters**

- **sender** (*computation*) – name of the computation that sent the message
- **_** (*terminate message*) –
- **t** (*timestamp*) –

For documentation on how to develop new algorithms, look at the this *tutorial* and this *documentation*.

`pydcop.algorithms` also defines objects and functions that are used to describe and define DCOP algorithms' computations.

### Classes

| | |
|---|---|
| `ComputationDef`(node, algo) | Full definition of a Computation. |
| `AlgorithmDef`(algo, params[, mode]) | Full definition of an algorithm's instance. |
| `AlgoParameterDef` | Definition of an algorithm's parameter. |

### Functions

| | |
|---|---|
| `list_available_algorithms`() | The list of available DCOP algorithms. |
| `load_algorithm_module`(algo_name) | Dynamically load an algorithm module. |
| `prepare_algo_params`(params, ...) | Ensure algorithm's parameters are valid. |
| `check_param_value`(param_val, param_def) | Check if `param_val` is a valid value for a `AlgoParameterDef` |

## pydcop.distribution

A **distribution** method is used to decide which agent hosts each computation.

Distribution methods are implemented in `pydcop.distribution`. `object.py` defines objects that are used by all distribution methods (`Distribution` and `DistributionHints`). A distribution method computes the allocation of a set computations to a set of agents.

See *DCOP computation distribution* for more details on the distribution concept.

pyDCOP currently provides the following distribution methods :

### pydcop.distribution.oneagent

The `oneagent` distribution algorithm assigns exactly one computation to each agent in the system.

It is the most simple distribution and, when used with many DCOP algorithms, it replicates the traditional hypothesis used in the DCOP literature where each agent is responsible for exactly one variable.

Note that this applies to algorithms using a computation-hyper graph model, like DSA, MGM, etc.

The `oneagent` distribution does not define any notion of distribution cost.

#### Functions

**distribute**(*computation_graph: pydcop.computations_graph.objects.ComputationGraph*, *agentsdef: Iterable[*pydcop.dcop.objects.AgentDef*]*, *hints: pydcop.distribution.objects.DistributionHints = None*, *computation_memory=None*, *communication_load=None*, *timeout=None*) → pydcop.distribution.objects.Distribution

Simplistic distribution method: each computation is hosted on agent agent and each agent host a single computation. Agent capacity is not considered.

Raises an ImpossibleDistributionException

**Parameters**

- **computation_graph** (*a ComputationGraph*) – the computation graph containing the computation that must be distributed
- **agentsdef** (*iterable of AgentDef objects*) – The definition of the agents the computation will be assigned to. There **must** be at least as many agents as computations.
- **hints** – Not used by the `oneagent` distribution method.
- **computation_memory** – Not used by the `oneagent` distribution method.
- **computation_memory** – Not used by the `oneagent` distribution method.

**Returns** **distribution** – A Distribution object containing the mapping form agents to computations.

**Return type** *Distribution*

**distribution_cost**(*distribution: pydcop.distribution.objects.Distribution*, *computation_graph: pydcop.computations_graph.objects.ComputationGraph*, *agentsdef: Iterable[*pydcop.dcop.objects.AgentDef*]*, *computation_memory: Callable[pydcop.computations_graph.objects.ComputationNode, float]*, *communication_load: Callable[[pydcop.computations_graph.objects.ComputationNode, str], float]*) → float

As the `oneagent` distribution does not define any notion of distribution cost, this function always returns 0.

**Parameters**

- **distribution** –
- **computation_graph** –
- **agentsdef** –
- **computation_memory** –
- **communication_load** –

**Returns** 0

**Return type** distribution cost

### pydcop.distribution.ilp_fgdp

### pydcop.distribution.ilp_compref

### pydcop.distribution.heur_comhost

### Implementing a distribution method

To implement a new distribution method, one must:

- create a new module in `pydcop.distribution`, named after the distribution method
- define the following methods in this file:
  - `distribute`, which returns a Distribution object
  - `distribution_cost`, which return the cost of a distribution. If your distribution algorithm does define a notion of cost, you can simply return 0 (like the `oneagent` distribution)
- additionally, for dynamic distribution, you can also provide these methods:
  - `distribute_remove`
  - `distribute_add`

### Base classes

**class Distribution**(*mapping: Dict[str, List[str]]*)

This object is a convenient representation of a distribution with methods for querying it (has_computation, agent_for, etc.)

**Parameters** **mapping** (*mapping: Dict[str, List[str]]*) – A dict agent name: [computation names]. Basic validity checks are performed to ensure that the same computation is not hosted on several agents.

**agent_for**(*computation: str*) → str

Agent hosting one given computation :param computation: a computation's name :type computation: str

**Returns** the name of the agent hosting this computation.

**Return type** str

**property agents**

Agents used in this distribution

**Returns** The list of the names of agents used in this distribution.

**Return type** agents list

**property computations**

Distributed computations

**Returns** A list containing the names of the computations distributed in this distribution.

**Return type** computations list

**computations_hosted**(*agent: str*) → List[str]

Computations hosted on an agent.

If the agent is not used in the distribution (its name is not known), returns an empty list.

**Parameters** **agent** (*str*) – the name of the agent

**Returns** The list of computations hosted by this agent.

**Return type** List[str]

**has_computation**(*computation: str*)

**Parameters** **computation** (*str*) – computation name

**Returns** True if this computation is part of the distribution

**Return type** Boolean

**host_on_agent**(*agent: str*, *computations: List[str]*)
Host several computations on an agent.

Modify the distribution by adding computations to be hosted on agent. If this agent name is unknown, it is added, otherwise the list of computations is added to the computations already hosted by this agent.

**Parameters**

- **agent** (*str*) – an agent name
- **computations** (*List[str]*) – A list of computation names

**is_hosted**(*computations: Union[str, Iterable[str]]*)
Indicates if some computations are hosted.

This methods does not care on which agent the computations are hosted.

**Parameters** **computations** (*List[str]*) – A list of computation names

**Returns** True if all computations are hosted.

**Return type** Boolean

**mapping**() → Dict[str, List[str]]
The distribution represented as a dict.

**Returns** A dict associating a list of computation names to each agent name.

**Return type** Dict[str, List[str]]

### pydcop.dcop

Objects used for representing DCOP are in the package `pydcop.dcop`

**class Domain**(*name: str*, *domain_type: str*, *values: Iterable*)
A VariableDomain indicates which are the valid values for variables with this domain. It also indicates the type of environment state represented by there variable : 'luminosity', humidity', etc.

A domain object can be used like a list of value as it support basic list-like operations : 'in', 'len', iterable…

**index**(*val*)
Find the position of a value in the domain

**Parameters** **val** – a value to look for in the domain

**Returns**

**Return type** the index of this value in the domain.

### Examples

```
>>> d = Domain('d', 'd', [1, 2, 3])
>>> d.index(2)
1
```

**to_domain_value**(*val: str*)

Find a domain value with the same str representation

This is useful when reading value from a file.

**Parameters** **val** (*str*) – a string that should match a value in the domain (which may contains non-string values, eg int)

**Returns**

- *a pair (index, value) where index is the position of the value in the*
- *domain and value the actual value that matches val.*

### Examples

```
>>> d = Domain('d', 'd', [1, 2, 3])
>>> d.to_domain_value('2')
(1, 2)
```

**class Variable**(*name: str*, *domain: Union[*pydcop.dcop.objects.Domain*, Iterable[Any]]*, *initial_value=None*)

A DCOP variable.

This class represents the definition of a variable : a name, a domain where the variable can take it's value and an optional initial value. It is not used to keep track of the current value assigned to the variable.

**Parameters**

- **name** (*str*) – Name of the variable. You must use a valid python identifier if you want to use python expression (given as string) to define constraints using this variable.
- **domain** (*Domain or Iterable*) – The domain where this variable can take its value. If an iterable is given a Domain object is automatically created (named after the variable name: *d_<var_name>*.
- **initial_value** (*Any*) – The initial value assigned to the variable.

**class AgentDef**(*name: str*, *default_route: float = 1*, *routes: Dict[str, float] = None*, *default_hosting_cost: float = 0*, *hosting_costs: Dict[str, float] = None*, ***kwargs: Union[str, int, float]*)

Definition of an agent.

AgentDef objects are used when only the definition of the agent is needed, and not the actual running agents. This is for example the case when computing the computations' distribution, or when instanciating concrete agents.

### Notes

Route cost default to 1 because they are typically used as a multiplier for message cost when calculating communication cost. On the other hand, hosting cost default to 0 because they are used in a sum. In order to allow using problem-specific attribute on agents, any named argument passed when creating an AgentDef is available as an attribute

### Examples

```
>>> a1 = AgentDef('a1', foo='bar')
>>> a1.name
'a1'
>>> a1.foo
'bar'
```

**Parameters**

- **name** (*str*) – the name of the agent
- **default_route** (*float*) – the default cost of a route when not specified in routes.
- **routes** (*dictionary of agents name, as string, to float*) – attribute a specific route cost between this agent and the agents whose names are used as key in the dictionary
- **default_hosting_cost** – the default hosting for a computation when not specified in hosting_costs.
- **hosting_costs** (*dictionary of computation name, as string, to float*) – attribute a specific cost for hosting the computations whose names are used as key in the dictionary.
- **kwargs** (*dictionary string -> any*) – any extra attribute that should be available on this AgentDef object.

**extra_attr**() → Dict[str, Any]
Extra attributes for this agent definition.

These extra attributes are the *kwargs* passed to the constructor. They are typically used to defined extra properties on an agent, like the capacity.

**Returns**

**Return type** Dictionary of strings to values,

**hosting_cost**(*computation: str*) → float
The cost for hosting a computation.

**Parameters** **computation** (*str*) – the name of the computation

**Returns** the cost for hosting a computation

**Return type** float

#### Examples

```
>>> agt = AgentDef('a1', default_hosting_cost=3)
>>> agt.hosting_cost('c2')
3
>>> agt.hosting_cost('c3')
3
```

```
>>> agt = AgentDef('a1', hosting_costs={'c2': 6})
>>> agt.hosting_cost('c2')
6
>>> agt.hosting_cost('c3')
0
```

**route**(*other_agt: str*) → float

The route cost between this agent and other_agent.

**Parameters** **other_agt** (*str*) – the name of the other agent

**Returns** the cost of the route

**Return type** float

#### Examples

```
>>> agt = AgentDef('a1', default_route=5)
>>> agt.route('a2')
5
>>> agt.route('a1')
0
```

```
>>> agt = AgentDef('a1', routes={'a2':8})
>>> agt.route('a2')
8
>>> agt.route('a3')
1
```

**TODO: add documentation for `Constraint` object and utility functions** (`relation_from_str(espression, variables)`, etc. )

### Agents

Agents can be responsible for several variables and can host one or several computations. Each agent has it's own thread and message queue and manage communication with other agents. An agent delivers messages to the computations it hosts and send messages on their behalf.

When running a dcop, all agents can run in the same process or in different processes. In the first case communication is implemented with direct object exchanges between the agent's threads. In the second case, agents may run on different computers and communication uses the network. Current implementation is based on HTTP+JSON but other network communication mecanism (zeromq, CoAP, BSON, etc.) could easily be implemented by subclassing `CommunicationLayer`

The Orchestrator is a special agent that is not part of the DCOP : it's role is to bootstrap the solving process by distributing the computations on the agents. It also collects metrics for benchmark purpose. Once the system is started (and if no metric is collected), the orchestrator could be removed.

Agents objects are implemented in 3 different modules:

- pydcop.dcop.infrastructure.agents
- pydcop.dcop.infrastructure.orchestratedagents
- pydcop.dcop.infrastructure.orchestrator

| | |
|---|---|
| *agents* | Base 'Agent' classes. |
| *orchestratedagents* | |
| *orchestrator* | |

### pydcop.infrastructure.agents

Base 'Agent' classes.

An Agent instance is a stand-alone autonomous object. It hosts computations, which send messages to each other. Each agent has its own thread, which is used to handle messages as they are dispatched to computations hosted on this agent.

#### Functions

| | |
|---|---|
| notify_wrap(f, cb) | |

#### Classes

| | |
|---|---|
| Agent(name, comm[, agent_def, ui_port, ...]) | Object representing an agent. |
| RepairComputation(agent) | |
| RepairComputationRegistration(computation, ...) | |
| ResilientAgent(name, comm, agent_def, ...[, ...]) | An agent that supports resiliency by replicating it's computations. |

#### Exceptions

| | |
|---|---|
| AgentException | |

### pydcop.infrastructure.orchestratedagents

#### Classes

| | |
|---|---|
| OrchestratedAgent(agt_def, comm, ...[, ...]) | An *OrchestratedAgent* is an agent that can be controlled through messages. |
| OrchestrationComputation(agent) | The OrchestrationComputation is used by OrchestratedAgents to answer to all messages from the orchestrator. |

#### pydcop.infrastructure.orchestrator

#### Classes

| | |
|---|---|
| `AgentsMgt`(algo, cg, agent_mapping, dcop, ...) | Computation for the orchestrator. |
| `Orchestrator`(algo, cg, agent_mapping, comm, dcop) | Centralized organisation of the set of agents used to solve a dcop. |
| `RepairReadyMessage`(agent, computations) | Sent by a resilient orchestrated agent to the orchestrator, when the computations for the repair dcop are ready. |
| `RepairRunMessage`() | Sent by the orchestrator to resilient orchestrated agents to start the computations for the repair dcop are ready. |
| `SetupRepairMessage`(repair_info) | Sent to an agent when the distribution must be repaired. |

## 3.5 Algorithms

pyDCOP currently implements the following algorithms:

We welcome contributions, especially the implementation of DCOP algorithms (novel or well-known) ! Join us on GitHub.

## 3.6 Bibliography

CHAPTER

FOUR

# CONTRIBUTING

We welcome contributions, especially the implementation of DCOP algorithms (novel or well-known). Join us on GitHub.

CHAPTER

FIVE

# LICENCE

pyDCOP is license under the BSD-3-clause license.

CHAPTER

SIX

# INDICES AND TABLES

- genindex
- modindex
- search

# BIBLIOGRAPHY


[CPRA15] Jesus Cerquides, Gauthier Picard, and juan Rodriguez-Aguilar. Designing a Marketplace for the Trading and Distribution of Energy in the Smart Grid. In *14th International Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, 1285–1293. Istanbul, Turkey, 2015. URL: https://hal.archives-ouvertes.fr/hal-01122020.

[CEPRA18] Jesús Cerquides, Rémi Emonet, Gauthier Picard, and Juan A Rodríguez-Aguilar. Improving max-sum through decimation to solve loopy distributed constraint optimization problems. In *OPTMAS proceedings*, 13. 2018.

[FRPJ08] Farinelli, Rogers, Petcu, and Jennings. Decentralised Coordination of Low-power Embedded Devices Using the Max-sum Algorithm. In *Proceedings of the 7th international joint conference on Autonomous agents and multiagent systems - Volume 2*, volume 2. Richland, SC, 2008. International Foundation for Autonomous Agents and Multiagent Systems. OCLC: 881353656. URL: http://dl.acm.org/citation.cfm?id=1402298.1402313.

[FYP17] Ferdinando Fioretto, William Yeoh, and Enrico Pontelli. A Multiagent System Approach to Scheduling Devices in Smart Homes. In *Proceedings of the 16th Conference on Autonomous Agents and MultiAgent Systems*, 9. São Paulo, Brazil, 2017.

[FM03] Stephen Fitzpatrick and Lambert Meertens. Distributed coordination through anarchic optimization. In Victor Lesser, Charles L. Ortiz, and Milind Tambe, editors, *Distributed Sensor Networks*, volume 9, pages 257–295. Springer US, 2003. URL: http://link.springer.com/10.1007/978-1-4615-0363-7_11, doi:10.1007/978-1-4615-0363-7_11.

[HY97] Katsutoshi Hirayama and Makoto Yokoo. Distributed partial constraint satisfaction problem. In *Proceedings of the International Conference on Principles and Practice of Constraint Programming*, volume 1330, 222–236. Springer Berlin Heidelberg, 1997. URL: http://link.springer.com/10.1007/BFb0017442, doi:10.1007/BFb0017442.

[KSL+08] Joseph B. Kopena, Evan A. Sultanik, Robert N. Lass, Duc N. Nguyen, Christopher J. Dugan, Pragnesh J. Modi, and William C. Regli. Distributed Coordination of First Responders. *IEEE Internet Computing*, 12(1):45–47, January 2008. URL: http://ieeexplore.ieee.org/document/4428334/, doi:10.1109/MIC.2008.9.

[LdSFB08] Fabiana Lorenzi, Fernando dos Santos, Paulo R. Ferreira, and Ana L. C. Bazzan. Optimizing Preferences within Groups: A Case Study on Travel Recommendation. In Gerson Zaverucha and Augusto Loureiro da Costa, editors, *Advances in Artificial Intelligence - SBIA 2008*, 103–112. Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[MTB+04] Rajiv T. Maheswaran, Milind Tambe, Emma Bowring, Jonathan P. Pearce, and Pradeep Varakantham. Taking DCOP to the real world: efficient complete solutions for distributed multi-event scheduling. In *in AAMAS*, 310–317. 2004.

[MPP+12] Tânia L. Monteiro, Marcelo E. Pellenz, Manoel C. Penna, Fabrício Enembreck, Richard Demo Souza, and Guy Pujolle. Channel allocation algorithms for WLANs using distributed optimization. *AEU - International Journal of Electronics and Communications*, 66(6):480–490, June 2012. URL: http://linkinghub.elsevier.com/retrieve/pii/S1434841111002640, doi:10.1016/j.aeue.2011.10.012.

[Mor93] Paul Morris. The Breakout Method for Escaping from Local Minima. In *Proceedings of the Eleventh National Conference on Artificial Intelligence (AAAI'93)*, 40–45. 1993. URL: http://www.aaai.org/Papers/AAAI/1993/AAAI93-007.pdf.

[Pet07] Adrian Petcu. *A Class of Algorithms for Distributed Constraint Optimization*. PhD thesis, École Polytechnique Fédérale de Lausanne, 2007.

[PF04] Adrian Petcu and Boi Faltings. A distributed, complete method for multi-agent constraint optimization. In *CP 2004 - Fifth International Workshop on Distributed Constraint Reasoning (DCR2004)*, 15. 2004. URL: http://infoscience.epfl.ch/record/52657.

[RPR16] Pierre Rust, Gauthier Picard, and Fano Ramparany. Using Message-passing DCOP Algorithms to Solve Energy-efficient Smart Environment Configuration Problems. In *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence*, IJCAI'16, 468–474. New York, New York, USA, 2016. AAAI Press. URL: http://dl.acm.org/citation.cfm?id=3060621.3060687.

[RPR17] Pierre Rust, Gauthier Picard, and Fano Ramparany. On the Deployment of Factor Graph Elements to Operate Max-Sum in Dynamic Ambient Environments. In *Autonomous Agents and Multiagent Systems*, volume 10642, 116–137. Cham, 2017. Springer International Publishing. URL: http://link.springer.com/10.1007/978-3-319-71682-4_8, doi:10.1007/978-3-319-71682-4_8.

[RPR18] Pierre Rust, Gauthier Picard, and Fano Ramparany. Self-Organized and Resilient Distribution of Decisions over Dynamic Multi-Agent Systems. In *OPTMAS 2018*. 2018.

[WZ03] Lars Wittenburg and Weixiong Zhang. Distributed breakout algorithm for distributed constraint optimization problems – DBArelax. In *AAMAS '03 Proceedings of the second international joint conference on Autonomous agents and multiagent systems*, 1158. Melbourne, Australia, 2003. ACM Press. URL: http://portal.acm.org/citation.cfm?doid=860575.860844, doi:10.1145/860575.860844.

[YH96] Makoto Yokoo and Katsutoshi Hirayama. Distributed Breakout Algorithm for Solving Distributed Constraint Satisfaction Problems. *AAAI*, 1996. URL: http://citeseer.ist.psu.edu/viewdoc/summary?doi=10.1.1.21.65.

[ZWXW05] Weixiong Zhang, Guandong Wang, Zhao Xing, and Lars Wittenburg. Distributed stochastic search and distributed breakout: properties, comparison and applications to constraint optimization problems in sensor networks. *Artificial Intelligence*, 161(1-2):55–87, January 2005. URL: http://linkinghub.elsevier.com/retrieve/pii/S0004370204001481, doi:10.1016/j.artint.2004.10.004.

[ZXWW03] Weixiong Zhang, Zhao Xing, Guandong Wang, and Lars Wittenburg. An Analysis and Application of Distributed Constraint Satisfaction and Optimization Algorithms in Sensor Networks. In *Proceedings of the Second International Joint Conference on Autonomous Agents and Multiagent Systems*, AAMAS '03, 185–192. New York, NY, USA, 2003. ACM. URL: http://doi.acm.org/10.1145/860575.860605, doi:10.1145/860575.860605.

# PYTHON MODULE INDEX

## p

# INDEX

## N



## O



## P



## R



## S



## T

## V