

UNIVERSITÉ DE LILLE

CRIS<sub>t</sub>AL

SMAC TEAM

---

# Individual-based Approach to Distributed Constraint Optimization Problem

---

*Author:*  
Alex VIGNERON

*Supervisors :*  
Anne-Cécile CARON  
Maxime MORGE  
Jean-Christophe ROUTIER

November 13, 2020



### **Abstract**

This paper presents a short overview of DCOPs and details MGM and MGM-2 algorithms. Both algorithms are presented with corresponding detailed automata and two afferent examples of execution.

# Contents

<b>1</b>	<b>Introduction</b>	<b>4</b>
<b>2</b>	<b>Distributed Constraint Optimization Problems</b>	<b>5</b>
2.1	Origins . . . . .	5
2.2	Classification . . . . .	7
<b>3</b>	<b>State of the art</b>	<b>9</b>
3.1	General overview . . . . .	9
3.2	MGM & MGM-2 . . . . .	9
3.2.1	MGM-2 overview . . . . .	10
3.2.2	MGM-2 prnciple . . . . .	10
<b>4</b>	<b>Algorithms</b>	<b>12</b>
4.1	MGM . . . . .	12
4.2	MGM-2 . . . . .	16
<b>5</b>	<b>FSM modeling</b>	<b>19</b>
5.1	Outline . . . . .	19
5.2	Agent States . . . . .	19
5.2.1	General overview . . . . .	19
5.2.2	Init . . . . .	20
5.2.3	Continue . . . . .	21
5.2.4	waitingForRole . . . . .	22
5.2.5	OffererWaitingValues . . . . .	22
5.2.6	OffererMakingOffer . . . . .	22
5.2.7	ReceiverWaitingValues . . . . .	24
5.2.8	ReceiverWaitingOffers . . . . .	25
5.2.9	ReceiverAllOffersReceived . . . . .	25
5.2.10	Comitted . . . . .	27
5.2.11	Uncomitted . . . . .	27
5.2.12	ActSolo . . . . .	28
5.2.13	GivingPartnerGoNogo . . . . .	28
5.2.14	HandlingPartnersGoNogo . . . . .	28
5.2.15	Final . . . . .	28
5.3	Agent's Mind . . . . .	29
5.3.1	Personal beliefs . . . . .	29
5.4	Messages . . . . .	29
5.4.1	Standard agent messages . . . . .	29
5.5	Outline . . . . .	30
5.6	Supervisor states . . . . .	31
5.6.1	General overview . . . . .	31
5.6.2	Start . . . . .	31
5.6.3	WaitingForAgentValues . . . . .	31
5.6.4	DecidingToStopOrContinue . . . . .	31

5.6.5	Finish . . . . .	31
5.7	Supervisor's mental state . . . . .	31
5.8	Supervisor messages . . . . .	32
<b>6</b>	<b>Example 1: An audience with lady Galadriel</b>	<b>33</b>
6.1	Introduction . . . . .	33
6.2	MGM approach . . . . .	35
6.3	MGM-2 approach . . . . .	36
6.4	Modelling of the problem with SCADCOP API . . . . .	39
<b>7</b>	<b>Example 2: Dalek's surveillance system</b>	<b>41</b>
7.1	Introduction . . . . .	41
7.2	MGM resolution . . . . .	42
7.3	MGM-2 resolution . . . . .	44
7.4	Modelling of the problem with SCADCOP API . . . . .	44
<b>8</b>	<b>Glossary of technical terms</b>	<b>50</b>
<b>9</b>	<b>Conclusion</b>	<b>52</b>
<b>10</b>	<b>Annex</b>	<b>53</b>

# 1 Introduction

Among existing DCOP algorithms, Maximum-Gain-Message (MGM) and its 2-coordinated version MGM-2 are popular. Other similar algorithms such as DSA which is considered as a stochastic variant of MGM Fioretto et al. (2018) and BE-Rebid Taylor et al. (2010) which extends MGM by calculating and communicating expected gain Fioretto et al. (2018) also exist. Several open-source implementations of MGM exist in Python with the pyDCOP framework developed by Rust et al. (2019) and Java with the Frodo project offered by Léauté et al. (2009). Though not necessarily always the best performers, MGM and its variants are deemed robust and efficient algorithms on average, making them suitable benchmark options Fioretto et al. (2018). We hereby present the MGM and MGM-2 algorithms and detail two toy-examples to facilitate their analysis.

We begin by giving a short overview of what DCOPs are and a general picture of the state of the art algorithms used to solve them. We then move on to a brief review of the existing algorithms to address DCOPs and detail two of these, namely MGM and MGM-2. We proceed to the description of the MGM-2 automaton and finally move on to two examples of execution of said algorithms.

## 2 Distributed Constraint Optimization Problems

A Distributed Constraint Optimization Problem (DCOP) framework is the distributed version of constraint optimization problems. In this multi-agent paradigm, agents representing variables in the problem communicate so that each of them can gradually update the value of the variable it controls to improve the global utility function.

There are at least as many variables as agents and possibly more variables than agents, implying that a single agent might control several variables. However in most DCOPs, control of a single variable by a single agent is assumed. The optimal solution is the minimum/maximum of the global objective function. Each variable has a domain of values it can take, this domain is only known to the agent in control of the said variable. However the value of a variable is known to an agent's neighbour. The notion of neighbourhood is key because DCOPs rely on locality. Therefore each agent can only communicate with its neighbours and only knows about cost functions which involve at least one of the variables it controls.

### 2.1 Origins

In order to better understand DCOPs, we give here a brief overview of the types of problems they come from.

A CSP is a problem where the goal is to determine whether a set of variables spanning over determined domains can satisfy constraints. These are typically combinatorial problems Fioretto et al. (2018)

**Definition 1 (CSP)** *A Constraint Satisfaction Problem is a tuple  $\langle X, D, C \rangle$  where*

- $X = \{x_1, \dots, x_n\}$  *is the set of variables*
- $D = \{d_1, \dots, d_n\}$  *is the set of corresponding non-empty domains*
- $C = \{c_1, \dots, c_m\}$  *is the set of constraints over the variables.*

COPs -sometimes called Weighted Constraint Satisfaction Problems- take a step further when compared with CSPs. Here the solution is not binary simply stating whether constraint can be satisfied or not but values are quantifiable. COPs are akin to CSPs with an additional particular objective function, either a maximisation or a minimisation. In this setting, constraints can either be hard or soft depending on whether respecting them is vital or preferable. There are two types of objective functions, maximisations with their respective gain and minimisations with their respective cost. Typically, a constraint which *needs* to be satisfied will be coined *hard* while a constraint which *should* be satisfied will be coined *soft*.

**Definition 2 (COP)** *A Constraint Optimization Problem is a tuple  $\langle X, D, F, \alpha \rangle$  where:*

- $X = \{x_1, \dots, x_n\}$  is a set of  $n$  variables
- $D = \{D_1, \dots, D_n\}$  is the set of finite domains for the variables in  $X$ , with  $D_i$  being the set of possible values for the variable  $x_i$
- $F$  is a set of constraints. A constraint  $f_i \in F$  is a function  $f_i : \prod_{x_j \in \mathbf{x}^i} D_j \rightarrow \mathbb{R}^+ \cup \{\perp\}$ , where the set of variables  $\mathbf{x}^i = \{x_{i_1}, \dots, x_{i_k}\} \subseteq X$  is the scope of  $f_i$  and  $\perp$  is a special element used to denote that a given combination of values for the variables in  $\mathbf{x}^i$  is not allowed, and it has the property that  $a + \perp = \perp + a = \perp, \forall a \in \mathbb{R}$ .

A distributed version of CSPs also exists in the form of DisCSPs. DCOPs are decentralised versions of COPs which are in turn extended CSPs.

**Definition 3 (DCOP)** A Distributed Constraint Optimization Problem (DCOP) is a tuple  $\langle A, X, D, F, \alpha \rangle$  where:

- $A = \{a_1, \dots, a_m\}$  is a set of  $m$  autonomous agents
- $X = \{x_1, \dots, x_n\}$  is a set of  $n$  variables
- $D = \{D_1, \dots, D_n\}$  is the set of finite domains for the variables in  $X$ , with  $D_i$  being the set of possible values for the variable  $x_i$
- $F$  is a set of constraints. A constraint  $f_i \in F$  is a function  $f_i : \prod_{x_j \in \mathbf{x}^i} D_j \rightarrow \mathbb{R}^+ \cup \{\perp\}$ , where the set of variables  $\mathbf{x}^i = \{x_{i_1}, \dots, x_{i_k}\} \subseteq X$  is the scope of  $f_i$  and  $\perp$  is a special element used to denote that a given combination of values for the variables in  $\mathbf{x}^i$  is not allowed, and it has the property that  $a + \perp = \perp + a = \perp, \forall a \in \mathbb{R}$
- $\alpha : X \rightarrow A$  is a surjective function, from variables to agents, which assigns the control of each variable  $x \in X$  to an agent  $\alpha(x)$ .

A partial assignment is a value assignment for a proper subset of variables of  $X$ . An assignment is complete if it assigns a value to each variable in  $X$ . For a given complete assignment  $\sigma$ , we say that a constraint (also called cost function)  $f_i$  is satisfied by  $\sigma$  if  $f_i(\sigma(x_i)) \neq \perp$ . A complete assignment is a solution of a DCOP if it satisfies all its constraints.

In the case of anytime algorithms, each assignment is complete and gradually improves over time, which makes it possible to stop the algorithm at any point.

The optimization objective is represented by the function  $\bar{F}$ , which can be of different nature, either a minimisation or a maximisation. A solution is a complete assignment with cost different from  $\perp$ , and an optimal solution is a solution with minimal cost (resp. maximal utility). In general, this function is a sum of cost (resp. utility) constraints:  $F = \sum_i f_i$ ; but some approaches can use other kind of aggregation.

Additionally, in the following report, we consider that:

- $n = m$ , i.e. each agent controls only one variable. This restriction is often considered in the literature;

- constraints are binary constraints. More precisely,  $F$  contains at most one function  $f_{ij}$  per pair  $i, j$ ;
- there can be as many constraints as needed for each variable.

With these assumptions, a DCOP can be easily represented as a graph where vertices are agents (each agent owns one variable) and edges are binary constraints. Note that since constraints are binary and can only be over a given pair once, the resulting graph is never a multigraph.

**Definition 4** *Let  $\langle A, X, D, F, \alpha \rangle$  be a DCOP, such that*

*We define from  $F$  the global cost function and the local cost function of an agent depending on its neighborhood :*

- *the global cost function  $\bar{F} = \sum_{f_{ij} \in F} f_{ij}(x_i, x_j)$  ;*
- *the neighborhood  $N_i$  of agent  $a_i$  is  $\{a_j \in A \mid f_{ij} \in F\}$  ;*
- *the local cost function of agent  $a_i$  is  $c(i) = \sum_{a_j \in N_i} f_{ij}(x_i, x_j)$  ;*
- *given a valuation  $\sigma$  of variables in  $X$ , the contribution of agent  $a_i$  is the result of its local cost function for this valuation, i.e.  $\sum_{a_j \in N_i} f_{ij}(\sigma(x_i), \sigma(x_j))$ .*

We consider here cost functions, so the objective is to minimize the global cost function. Symmetrically, one can use utility functions and the objective becomes to maximize the global utility function.

## 2.2 Classification

In the multi-agent domain, DCOPs are classified according to their set of characteristics, namely:

- **Action effects** : stochastic or deterministic. Refers to the results obtained by actions, in a deterministic setting one action is considered to have one specific result in a particular context, and if this action had to be repeated in a similar context, the result ought to be identical. Whereas in a stochastic one, the same action performed in the same environment can lead to a different result.
- **Knowledge** : total or incomplete Refers to the extent of an agent's awareness of the other agent's variables. With incomplete knowledge, locality is often considered as the landmark, while with total knowledge any agent possesses information about any other agent.
- **Group behaviour** : cooperative or competitive Depends on whether agents consider their own welfare first or the general well-being of the system.
- **Environment type** : deterministic or stochastic Refers to how an environment evolves, whether events follow a certain pattern or not.



- Environment's dynamics : static or dynamic Depends on whether an environment remains identical from beginning to end of a process or evolves.

Generally speaking, most DCOP algorithms rely on deterministic action effects, a cooperative group-behaviour and total knowledge. MGM and MGM-2 are deterministic both in terms of actions and environment, with cooperative group behaviour sharing incomplete (local) knowledge.

DCOP algorithms are used for several applications, for instance:

- Disaster management
- Radio frequency allocation problems
- Recommendation Systems
- Scheduling
- Sensor networks
- Service-oriented management (cloud, server, power supply)
- Supply chain management

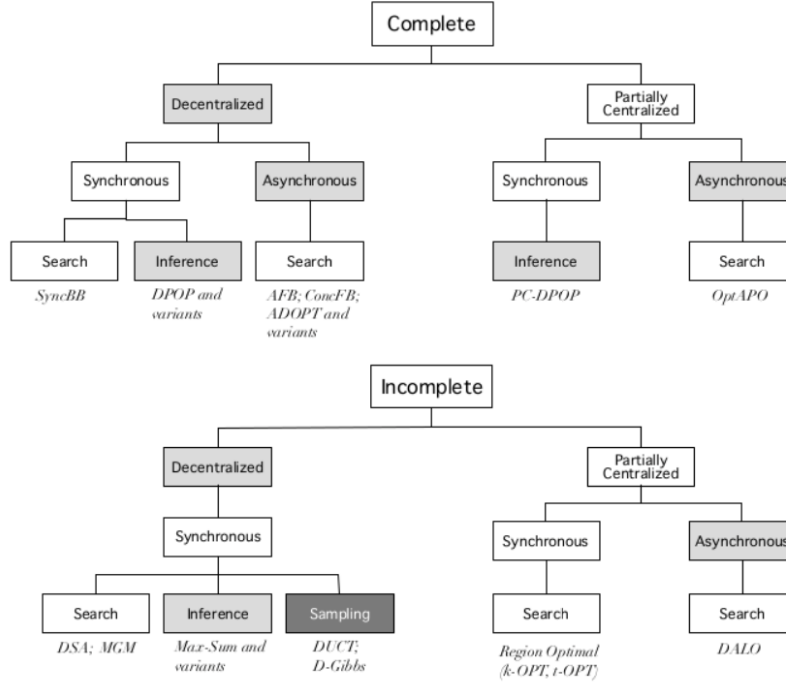


Figure 1: Fioretto’s taxonomy

### 3 State of the art

#### 3.1 General overview

DCOPs can be addressed by algorithms described in Fioretto et al. (2018) and presented in Figure 1. The first distinction made is in terms of solution optimality with complete algorithms guaranteeing the optimal solution to be found while incomplete ones offer no guarantee but have shorter execution times. Then, they are classified according to their (lack or presence of) centralisation and synchronicity. Finally, they are divided based on their exploration process, revolving around three main frames, namely search, inference and sampling. In our case, we focus on MGM and its coordinated variant MGM-2 which are incomplete, decentralised, semi-synchronous search algorithms.

#### 3.2 MGM & MGM-2

Both MGM and MGM-2 are extensively described by Maheswaran et al. (2004). In this paper the 2-coordinated algorithm is detailed and hints are given at k-coordinated versions.

Historically, MGM evolved from DBA with the difference that there is no

change on constraint costs to exit local minima issues and no need for DBA's global knowledge of solution quality. Another algorithm MGM is often compared with is DSA. The difference lies in the guarantees MGM offers when compared with DSA. MGM's gain never drops below 0 while DSA might. In terms of solution quality, both MGM and MGM-2 are proved to be monotonous, intuitively because since the utility function is the sum of all neighbours' utilities, if one increases, all neighbours' utility consequently increases too.

### 3.2.1 MGM-2 overview

Their main focus of interest is the application of DCOPs to large-scale problems where the limitation of fully-connected networks of agents (complete graphs) is high. Computational costs caused by such topologies are prohibitive and a possible solution to this is the local knowledge approach of distributed algorithms.

The principle is one of coordinated negotiation and distributed control of variables. "*The optimal solution of a DCOP is a Nash Equilibrium in an appropriate game*". Coordination and cooperation are key since selfish behaviours can result in unstable dynamics, which means a structure needs to be imposed on how values get updated. Agents can perform either unilateral or bilateral (2-coordinated) moves whereby they update their values according to the computed improvement in global utility. As opposed with coalition scenarios where a manager handles agent's decisions, therefore merging the behaviour into a centralised one, MGM-2 aims at allowing coordination while maintaining a distributed decision-making process. A notion of solidarity is necessary here, hinting at a cooperative environment but could be replaced by compensations between agents in a competitive environment. In MGM-2, coordinated pairs consider the overall gain they and their partner can achieve, irrespective of whether their own gain is better or not. This is possible because we base the interaction on a cooperative framework. Were it competitive, a joint action is considered useful if the sum of the 2 partnered agents utility functions increases, even if one of them diminishes.

### 3.2.2 MGM-2 principle

Globally speaking, the process of MGM could be summed-up as follows: at the beginning of each round, agents inform their neighbours of their current value. Thanks to this information received from each of their neighbours, each agent is then capable of computing the changes it can make to its own value to change its utility taking into account each neighbour's value. Once this is done, each agent selects the best move it can do and the corresponding gain and informs each of its neighbours of this. Among each neighbourhood, a single agent will be allowed to act, the one having made the best offer of move. The agent thus selected updates its value and another round can begin.

In MGM-2, the process is slightly more complex since coordinated moves come into play. The difference starts from the beginning of the round where agents are randomly split into 2 sets, *offerers* and *receivers*. Each set will have a

very different behaviour. Offerers select a neighbour at random, make an offer to said neighbour and wait for their neighbour's response. If the neighbour declines the offer, they switch back to an MGM-like behaviour where they compute their best solo move (as in MGM) and so on. If their neighbour accepts, they are from then on *committed*, just like their neighbour. Receivers merely wait for potential offers, they might receive none and therefore go for an MGM-like behaviour of solo move, or receive offers and choose among them. If they do get offers, they will chose the best one among the acceptable ones. If no offer is acceptable, they act as if they had not received any and go back to MGM behaviour. Once they have selected the best offer, they send the selected partner an *accept* message and are from now onwards *committed*. Committed agents, both receivers and offerers, converge again in their behaviour after the accept/reject messages. They both enter the phase where they inform each of their neighbours of their potential joint gain. This phase is akin to MGM since in each neighbourhood, only one agent will be the winner of that round. If the winner is an uncommitted agent, then the round continues as in MGM. However, if the winner is a committed agent, then more communication occurs between both partners. Each partner will send its partner a *Go* or *No-Go* message, depending on whether they have won or lost their neighbourhood's round. A No-Go means neither of them will make a move this round and so they move on to the next.

Since variables are in fact agents in a DCOP game, the optimal solution corresponds to a Nash Equilibrium in the specified game. The notion of vicinity is crucial and should be considered fixed once and for all, the agent's neighbours do not vary during the game. A round starts by all agents broadcasting the current value to their vicinity. This means each agent sends one message and receives  $|vicinity|$  messages, each containing one particular neighbours's current value. At this stage, all agents are now aware of their own value as well as of of their vicinity values. Now the stake is to select which agents will be allowed to act, aka modify their value, the set of them will be called  $M$ . To select said agents, each of them broadcasts a gain message, stating the  $\epsilon$  by which it can improve its current local utility value *if* allowed to act. At this stage, each agent knows the  $\epsilon$  by which it can improve but also all its vicinity's  $\epsilon$ s. The winner is simply the one which yields the highest  $\epsilon$  (potential improvement). Implementation should take into account possible ties and how to break them. In the case of MGM-2, it is a pair of agents which gain is highest which are allowed to act. The question of how to determine the highest pair gain for MGM-2 relies on which actions are acceptable or not by agents, for instance in cooperative environments, a joint gain can be considered acceptable even if one of the agents actually loses, in other settings this might not be the case and acceptable actions would in this case be limited to coordinated actions which improve both agent's value. The winner (winners in the case of MGM-2) act(s) and update its/their value(s) consequently.

Both processes go on until the algorithm is stopped, either by having reached the predefined number of cycles or by reaching a Nash equilibrium.

## 4 Algorithms

Both MGM and MGM-2 algorithms are described here.

### 4.1 MGM

The algorithm is described in Algorithms 1, 2, 3, 4, 6, 7, 8, 9, 10, 11,

We define  $N(a_i)$  the set of neighbouring agents for  $a_i$ :

$$N : A \rightarrow 2^A$$

$$a_i \mapsto N_i = N(a_i) =$$

From an agent's perspective, the MGM algorithm is performed as follows:

---

#### Algorithm 1 : Agent-centered view of MGM

---

**Input** :  $x_i \in X, d_i \in D, F_i \in F, b$   
**Output** :  $v \in d_i$ , the value of  $x_i$  variable such that  $v_{t+1}$  is more optimal than  $v_t$

```

1 while not terminated do
2    $\forall N(\text{self}) ! \text{informValue}(\text{self.val})$  //one cycle
3    $\text{currLocContext} \leftarrow \forall N(\text{self}) : \text{informValue}(\text{neighbourVal})$  //sync
   point
4    $(bVal, b\delta) \leftarrow \text{evaluate}(\text{currLocContext}, \text{currentValue}, d_x, b)$ 
5    $\forall N(\text{self}) ! \text{informDelta}(N(a), b\delta)$  //one cycle
6    $\text{currOffers} \leftarrow \text{getAllOffers}(N(a))$  //sync point
7    $\text{shouldAct} \leftarrow \text{decide}(\text{currOffers}, b\delta, b)$ 
8   if shouldAct is true then
9      $\text{currentValue} \leftarrow bVal$  // $a_i$  updates  $x_i$ 
10  end
11 end
12 supervisor !  $\text{informValue}(\text{self.val})$ 
```

---

While from a system-centered perspective, it goes as such:

---

#### Algorithm 2 : System-centered view of MGM

---

**Input** : A DCOP  $\langle A; X; D; F; \alpha \rangle$  such that  $|A| = |X|$  and  $\alpha(x_i) = a_i$ ;  
a boolean  $b$  set to true if the goal is miniMax, false if maxiMin  
**Output** : A solution to the DCOP

```

1  $\text{currentValue}(x_i) \leftarrow \text{rand}(d_i)$  //initialise currentValue
2 for  $a_i \in A$  //In parallel do
3    $\text{doAgentMGM}(x_i, d_i, F_i, b)$ 
4 end
5  $\text{currentGlobalContext} \leftarrow \text{currLocContext} \forall a \in A$ 
Return :  $\text{currentGlobalContext}$ 
```

---

---

**Algorithmme 3 :** evaluate(currentLocalContext, currentValue, domainX, b)

---

**Input :** The local context corresponding to current values of  $x_j \forall n \in N_i$ ; current value of the agent's variable  $x$ ; the domain  $d$  of the agent's variable; a boolean  $b$  set to true if the goal is miniMax, false if maxiMin

**Output :** A couple with the best value  $x$  can take given the current context and the  $\delta$  of change it can produce by doing so

```

1 results  $\leftarrow \emptyset$ 
2 for  $v \in d$  do
3    $x \leftarrow v$ 
4   results[v]  $\leftarrow \sum_{x_j \in N(a)} f(x, x_j)$  //sum over all arcs
5 end
6 if  $b$  is true then
7   (bestVal, best- $\delta$ )  $\leftarrow \min(\text{results})$  //min over  $\delta$  and
   corresponding  $v$ 
8 else
9   (bestVal, best- $\delta$ )  $\leftarrow \max(\text{results})$  //max over  $\delta$  and
   corresponding  $v$ 
10 end
    Return : (bestVal, best- $\delta$ )

```

---



---

**Algorithmme 4 :** decide(currentOffer, bestDelta, b)

---

**Input :**  $\Delta$  the set of  $\delta$  improvements offered by  $a$ 's neighbours;  $a$ 's  $\delta$ ; a boolean  $b$  set to true if the goal is miniMax, false if maxiMin

**Output :** A boolean, true if the agent  $a$  controlling  $x$  should act, false otherwise

```

1 if  $b$  is true then
2   if  $\delta_a \leq \min(\Delta)$  then
3     advice  $\leftarrow$  true
4   end
5 else if  $\delta_a \geq \max(\Delta)$  then
6   advice  $\leftarrow$  true
7 else
8   advice  $\leftarrow$  false
9 end
    Return : advice

```

---

---

**Algorithm 5 : communicateValuesToAll**

---

1 Symmetrical to getAllValues  
  **Input** :  $N(a), a, v \in D_x$   
  **Output** :  $(v, a)$   
2 **for**  $a_i \in N(a)$  **do**  
3   |  $\text{msg} \leftarrow \text{communicateValue}(a_i, v_x)$   
4 **end**  
  **Return** : msg

---

---

**Algorithm 6 : communicateValue**

---

1 Symmetrical to getValue  
  **Input** :  $a_i, a, v \in D_x$   
  **Output** :  $(v, a)$   
2  $\text{msg} \leftarrow \text{send}(v, a_i)$   
  **Return** : msg

---

---

**Algorithm 7 : getValue**

---

1 Symmetrical to communicateValue  
  **Input** :  $N(a)$   
  **Output** :  $(v, a)$   
2  $\text{msg} \leftarrow \text{receive}(\text{communicateValue}(a_i, a, v \in D_x))$   
  **Return** : msg

---

---

**Algorithm 8 : GetAllValues**

---

1 Symmetrical to communicateValuesToAll  
  **Input** :  $N(a)$   
  **Output** :  $\{(v_{ai}, a_i)\}$   
2  $\text{currLocContext} \leftarrow \emptyset$  //fresh start  
3 **while**  $|\text{currLocContext}| < |N(a)|$  **do**  
4   |  $\text{currLocContext} \leftarrow \text{currLocContext} \cup \text{getValue}(N(a))$   
5 **end**  
  **Return** : currLocContext

---

---

**Algorithm 9 : makeAllOffers**

---

1 Symmetrical to getAllOffers  
  **Input** :  $N(a), \delta_a$   
  **Output** :  $(\delta_a, a)$   
2 **for**  $a_i \in N(a)$  **do**  
3   |  $\text{makeOffer}(a_i, \delta_a)$   
4 **end**  
  **Return** :  $(\delta_a, a)$

---

---

**Algorithme 10 : makeOffer**

---

1 Symmetrical to getOffer

**Input :**  $a_i, \delta_a$

**Output :**  $(\delta_a, a)$

2 offer  $\leftarrow$  send( $\delta_a, a_i$ )

**Return :** offer

---

---

**Algorithme 11 : getOffer**

---

1 Symmetrical to makeOffer

**Input :**  $N(a)$

2 offer  $\leftarrow$  receive(makeOffer( $\delta_a, a$ )))

**Return :** offer

---

---

**Algorithme 12 : GetAllOffers**

---

1 Symmetrical to makeAllOffers

**Input :**  $N(a)$

2 currOffers  $\leftarrow \emptyset$  //fresh start

3 **while**  $|currOffers| < |N(a)|$  **do**

4     currOffers  $\leftarrow$  currOffers  $\cup$  getOffer( $N(a)$ )

5 **end**

**Return :**  $\{(\delta_{ai}, a_i)\}$

---



## 4.2 MGM-2

Algorithm 13 and its sub-algorithms 14, 15, 16, 17 and 18 present MGM-2.

---

**Algorithm 13 : MGM-2**

---

**Input :** A DCOP formalised as: A a set of autonomous agents, X a set of variables, D a set of finite domains for the  $x_i$  variables, F a set of utility/cost functions

**Output :** A solution to the DCOP

```
1 while not terminated do
2   Inform
3   Determine offerers/receivers as resp. O and R
4   Offer
5   Evaluate
6   Inform2
7   Go/No-Go
8   The agent with  $\max(bm_x)$  wins the round and is allowed to act
9 end
```

---

---

**Algorithm 14 : MGM-2 Inform**

---

```
1 for each agent  $a_i$  in A do
2   for each agent  $a_j$  in A do
3     if  $a_i$  and  $a_j$  share a function  $f_m$  then
4        $a_i$  informs  $a_j$  of the  $x_i$  value it controls in  $f_m$ 
5     end
6   end
7 end
```

---

---

**Algorithm 15 : MGM-2 Offer**

---

```
1 for each agent  $a_i$  in  $O$  do
2   offerer sends  $(a_i, a_j, gain^O(a_i, a_j))$  where  $gain^O(a_i, a_j)$  is offerer's
   local utility gain for suggested values  $a_i$  and  $a_j$ . Chose at
   random an agent  $a_j$  in  $R$  sharing a function  $f_m$  with self Send offer
   message containing all coordinated moves for the pair  $(a_i, a_j)$ 
   which improve current  $f_m$ 
3 end
```

---

---

**Algorithm 16 : MGM-2 Evaluate**

---

```
1 for each agent  $a_i$  in  $R$  do
2   Compute improvement  $\delta$  achievable, taking into account  $f_m$  offer
   and its own function  $f_n$ 
3   If  $gain^R(a_i, a_j)$  is receiver's local utility gain for suggested values
    $a_i$  and  $a_j$ 
4   for each  $(a_i, a_j)$ , compute
      $globalGain(a_i, a_j) = gain^R(a_i, a_j) + gain^O(a_i, a_j) - \Delta(a_i, a_j)$ 
     where  $\Delta(a_i, a_j)$  = how much "link between both" changes using
      $(a_i, a_j)$  ( not very clear in Mashewaran (fin page 4)).)
5   if  $max(globalGain(a_i, a_j)) > 0$  then
6     Commit both offerer and receiver by sending accept message
7     Offerer and receiver are now committed
8   end
9 end
```

---

---

**Algorithm 17 : MGM-2 Inform2**

---

```
1 Foreach agent in  $A$  Send a gain message to all neighbours
2 if agent is committed then
3   Gain message is for coordinated move
4 end
5 else
6   Gain message is best  $bm$  for unilateral move
7 end
```

---

---

**Algorithme 18 : MGM-2 Go/No-Go**

---

```
1 for each agent  $a_i$  in  $A$  do
2   if  $a_i$  is in  $C$  then
3     if  $a_i$ 's offer was the best of the neighbourhood then
4        $a_i$  send its committed partner  $a_j$  a go message
5     end
6     else
7        $a_i$  send  $a_j$  a no-go message
8     end
9   end
10  else
11    Uncommitted agents act if their offer was the best in their
12    neighbourhood
13  end
14 end
```

---

## 5 FSM modeling

### 5.1 Outline

According to the multi-agent approach, values of variables are discussed among agents who come to local agreements and iteratively repeat the process to finally yield a satisfying global assignment. Agents systematically base their decision on their local knowledge.

The process is triggered by a supervisor agent which invites other agents to begin their rounds of interactions. Each agent  $a_i$  is sent a **Trigger** message by the supervisor and initiates its variable's value at random in the variable's domain. The behaviour of each agent is defined by a finite state automata (see Figure 20 in annex for the big picture and Figures 2, 3, 4, 5 and 6 for focused extracts ).

Communication of potential moves and their corresponding potential gains helps agents determine who should act. A single move is an action whereby an agent changes the current value of its variable to another possible value belonging to the variable's domain. A coordinated move is similar but involves two agents changing the values of their variables. Note that a move can also be static if the change to *another* value of the domain happens to be the current value of the variable.

### 5.2 Agent States

#### 5.2.1 General overview

- **Init** : initial state in which the agent can be triggered by the supervisor.
- **Continue**: central state which sparks each round of the algorithm, agent's behaviour depends here on the message it receives from supervisor.
- **waitingForRole**: agent awaits for the role assignment which will tell it whether it will be an Offerer or a Receiver for this round.
- The Offerer's side:
  - **OffererWaitingValues**: the offerer agent collects all of its neighbours values before it can choose a neighbour at random and select a joint offer for it.
  - **OffererMakingOffer**: the Offerer selects one neighbour and selects an offer with that neighbour's value which is then sent. No matter the response given by the selected partner, only one offer can be made, hence if rejected the offerer will move on to act on its own.
- The Receiver's side:
  - **ReceiverWaitingOffers**: the receiver agent gathers all offers he receives

- **ReceiverAllOffersReceived**: the Receiver has received all offers from its neighbourhood and now chooses the best offer among those.
- **Uncommitted**: receivers who have not received any acceptable offer (or no offer at all) and Offerer's whose offer has been rejected join back in this state and wait for their neighbours deltas while computing their own deltas and informing their neighbourhood.
- **Comitted**: offerers which offers have been accepted and Receiver which have received at least one acceptable offer become committed with their respective partners.
- **ActSolo**: uncommitted agents evaluate their neighbourhood and decide whether they should act or not depending on the neighbourhood's winner.
- **GivingPartnerGoNogo**: committed agents evaluate their neighbourhood and decide whether they should act or not depending on the neighbourhood's winner.
- **HandlingPartnersGoNogo**: committed agents which can act check whether their partner can act too.
- **Final**: the last state an agent can be into, after having received the **StopAlgo** message from the supervisor.

### 5.2.2 Init

This is the initial state agents wait in before starting the algorithm. Messages received in this state are:

- **InformValue(someVal)**: if the agent receives a **InformValue(someVal)** message, it stashes it to deal with it when leaving **waitingForRole**.
- **Trigger**: upon receiving a **Trigger**, sent by the supervisor, agent performs a **mind.choose(variable)(mind.value.domain)** operation, this method is called only once per algorithm execution for each agent. The agent unstashes all previous **InformValue()** stashed messages and sends a **KickStartMe** message to the Supervisor. The agent will then transition to **Continue**.

During **mind.choose(variable)(mind.value.domain)**, the agent's variable  $x_i$  is initialised at a random value  $v_i \in d_i$ . When the Supervisor received **KickStartMe**, it will send back a **ContinueAlgo** message needed to get going. This artificial trigger of **ContinueAlgo** is used only once, at the beginning of the algorithm.

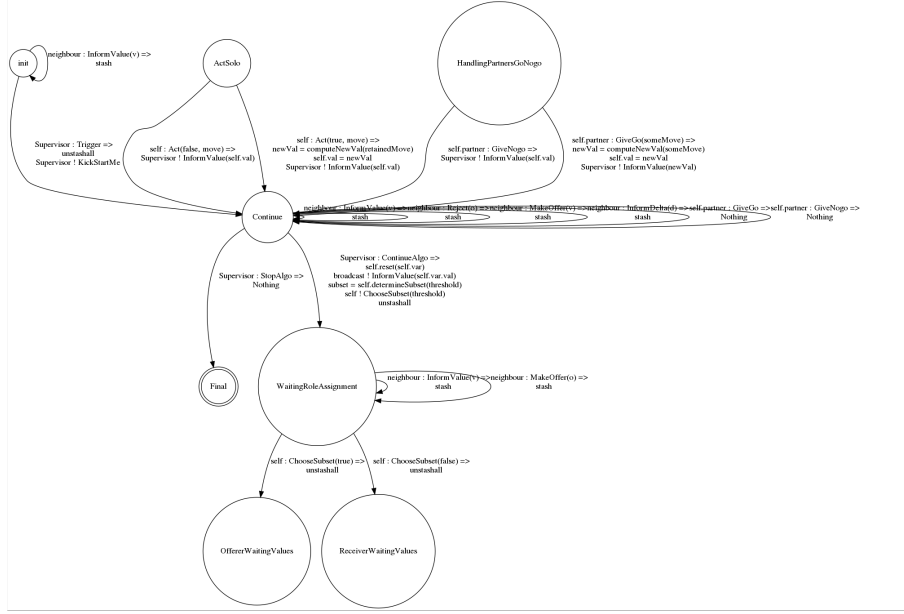


Figure 2: Agent's beginning

### 5.2.3 Continue

This state can be reached either via `Init`, `GivingPartnerGoNogo` or `HandlingPartnersGoNogo`, it should be considered as the central state from which is sparked each round. Messages received in this state:

- **InformValue(someVal):** neighbour's `InformValue(someVal)` are stashed to be dealt with when leaving `waitingForRole`.
- **GiveGo:** If the agent has reached this state after completing a round where it had been committed but could not act because it did not win the neighbourhood's delta competition, it will receive its partners' `GiveGo` or `GiveNoGo` here, but nothing should be done about it.
- **GiveNoGo:** If the agent has reached this state after completing a round where it had been committed but could not act because it did not win the neighbourhood's delta competition, it will receive its partners' `GiveGo` or `GiveNoGo` here, but nothing should be done about it.
- **StopAlgo:** agent transitions to `Final` with the whole process terminating
- **ContinueAlgo:** agent called `mind.reset()` method, emptying `mind.receivedOffers`, `mind.context` and `mind.deltas`. early received `InformValue()` messages are unstashed, then the agent sends its own value via `InformValue(mind.value)` to all its neighbours. It performs

`mind.determineSubset(threshold)` and send the outcome to self via `ChooseSubset()` before transitioning to `waitingForRole` where `ChooseSubset()` will be dealt with as well as unstashed `InformValue()`.

- `Reject(o)`: stashes it.
- `MakeOffer(o)`: stashes it.
- `InformDelta(d)`: stashes it.

#### 5.2.4 `waitingForRole`

This state acts as a crossroads between paths as Offerer or Receiver. Messages received in this state:

- `InformValue(someVal)`: are stashed to be dealt with when leaving this state for the targeted next one.
- `ChooseSubset(someSubset)`: triggers a transition to either `OffererWaitingValues` or `ReceiverWaitingOffers` according to the value of *someSubset*.

#### 5.2.5 `OffererWaitingValues`

Agents reaching this state are considered offerers. Messages received in this state:

- `InformValue(someVal)`: it is added to `mind.context`'s map while some are missing, when the last message of the kind is received, it triggers the transitioning steps towards `OffererMakingOffer`. Once `mind.context`'s map is complete (i.e. all neighbours have informed the agent of their value), the agent will transition to `OffererMakingOffer`. `MakeOffer(myOffer)` is sent to a neighbour chosen at random to be its potential partner; all other neighbours receive `MakeOffer(None)`.
- `MakeOffer(someOffer)`: all such messages will be answered with a `Reject()` since Offerers can not be Receivers.

#### 5.2.6 `OffererMakingOffer`

Before transitioning to the next state, the agent will wait here for the potential partner's response. Messages received in this state:

- `InformDelta()`: stashes it.
- `MakeOffer(someOffer)`: message is still answered with a `Reject(.)`
- `InformDelta(someDelta)`: messages received from neighbours which are one step ahead are stashed at this stage.

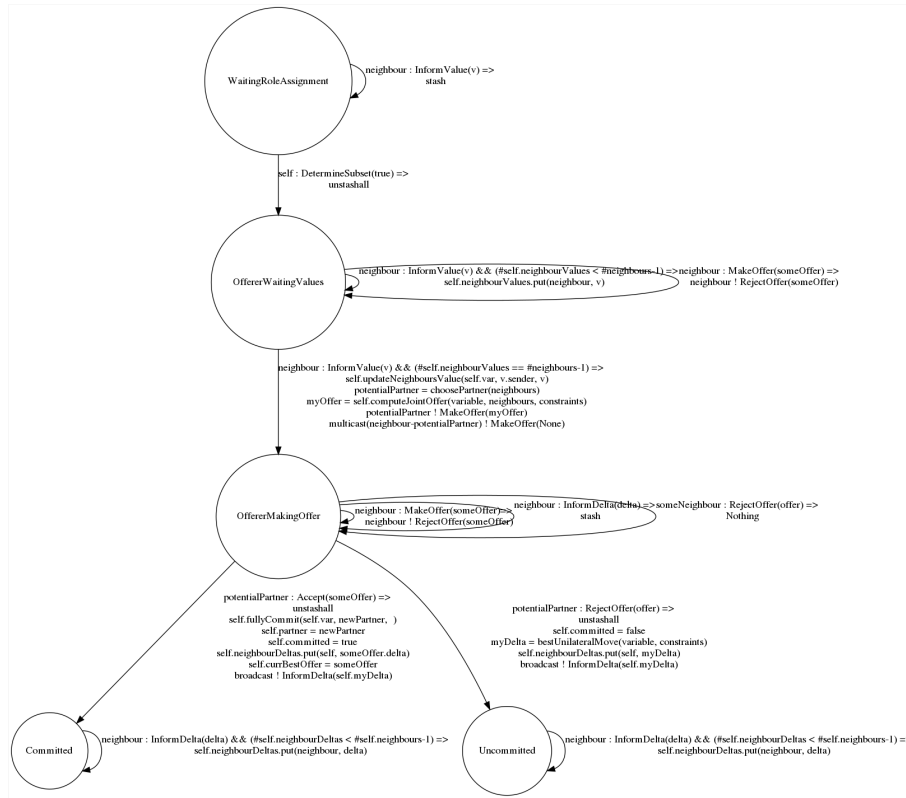


Figure 3: Offerer agent



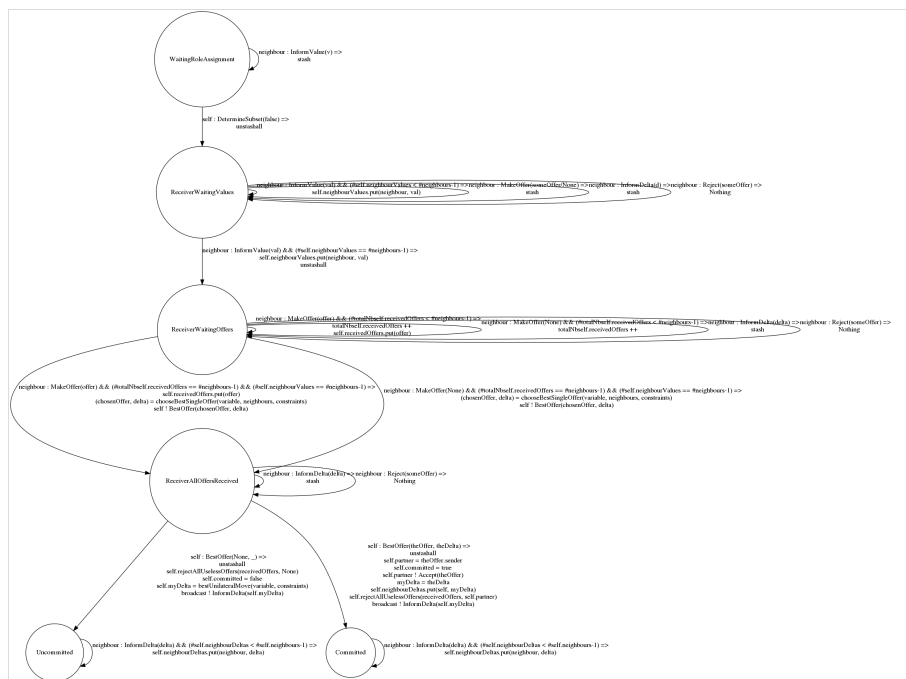


Figure 4: Receiver agent

- **Accept(someCombinedMove):** it will transition to committed and set `mind.isCommitted` to true as well as set `mind.partner` to `Accept(someOffer)`'s sender, this data will also help to compute the new delta.  
`InformDelta(mind.deltas.get(self.variable))` is sent to all neighbours after having computed its new delta either through its partner's `Accept(someCombinedMove)`'s delta or on its own if it has been rejected with `mind.computeSoloDelta(bestUnilateralMove)`.
- **Reject(someOffer):** the agent will transition to `Uncommitted`, set `mind.isCommitted` to true and set `mind.partner` to `None`.  
`InformDelta(mind.deltas.get(self.variable))` is sent to all neighbours after having computed its new delta either through its partner's `Accept(someCombinedMove)`'s delta or on its own if it has been rejected with `mind.computeSoloDelta(bestUnilateralMove)`.

### 5.2.7 ReceiverWaitingValues

In this state the receiver agent waits until it has been informed of all its neighbours' values. Messages received in this state:

- `InformDelta()`: stashes it.

- **InformValue(someVal)**: are stored in the `mind.context` map and become relevant only if the receiver does not get any acceptable offer. Once all of them have been received, the agent transitions to **ReceiverWaitingOffers**.
- **MakeOffer(None/someOffer)**: are stashed to be handled in **ReceiverWaitingOffers**.

### 5.2.8 ReceiverWaitingOffers

The last offer to be received causes the transition to the next state. At the end, when the last offer is received, whether the agent has received a real or **None** last offer, it will process the list of real offers received with `mind.chooseBestSingleOffer(variable, neighbours, constraints)`.

Messages received in this state:

- **MakeOffer(someOffer)**: stored in `mind.receivedOffers` map until all offers have been received.
- **MakeOffer(None)**: a counter is simply increased to keep track of the number and know when all offers have been received.

Once all offers have been received, the agent computes the result of `mind.chooseBestSingleOffer(variable, neighbours, constraints)` and sends it to itself with `BestOfferForMe(someOffer)` which will be handled in the next state **ReceiverAllOffersReceived**.

### 5.2.9 ReceiverAllOffersReceived

This state is a fork between committed and uncommitted Receiver agents. Messages received in this state:

- **InformDelta(someDelta)**: will be stashed and handled in either **Comitted** or **Uncomitted**.
- **BestOfferForMe(someOffer)**: will cause the agent to transition to **Comitted**, setting `mind.isCommitted` to **true** and setting its `mind.partner` variable to the agent it has committed with. the agent's current delta value is computed with `mind.computeJointDelta(somePartner)`. It will send a `Reject()` to all neighbours which are not the partner and an `Accept(someCombinedMove)` to the chosen partner.
- **BestOfferForMe(None)**: will cause the agent to transition to **Uncomitted** and set its `mind.isCommitted` to **false**. It will send a `Reject()` to all neighbours. The agent's current delta value is computed with `mind.computeSoloDelta(bestUnilateralMove)` and added to `mind.deltas`.

In both **BestOfferForMe(None/someOffer)** cases, **InformDelta(someDelta)** is sent to all neighbours, containing either the result of `mind.computeJointDelta(someDelta)` or `mind.computeSoloDelta(bestUnilateralMove)`.

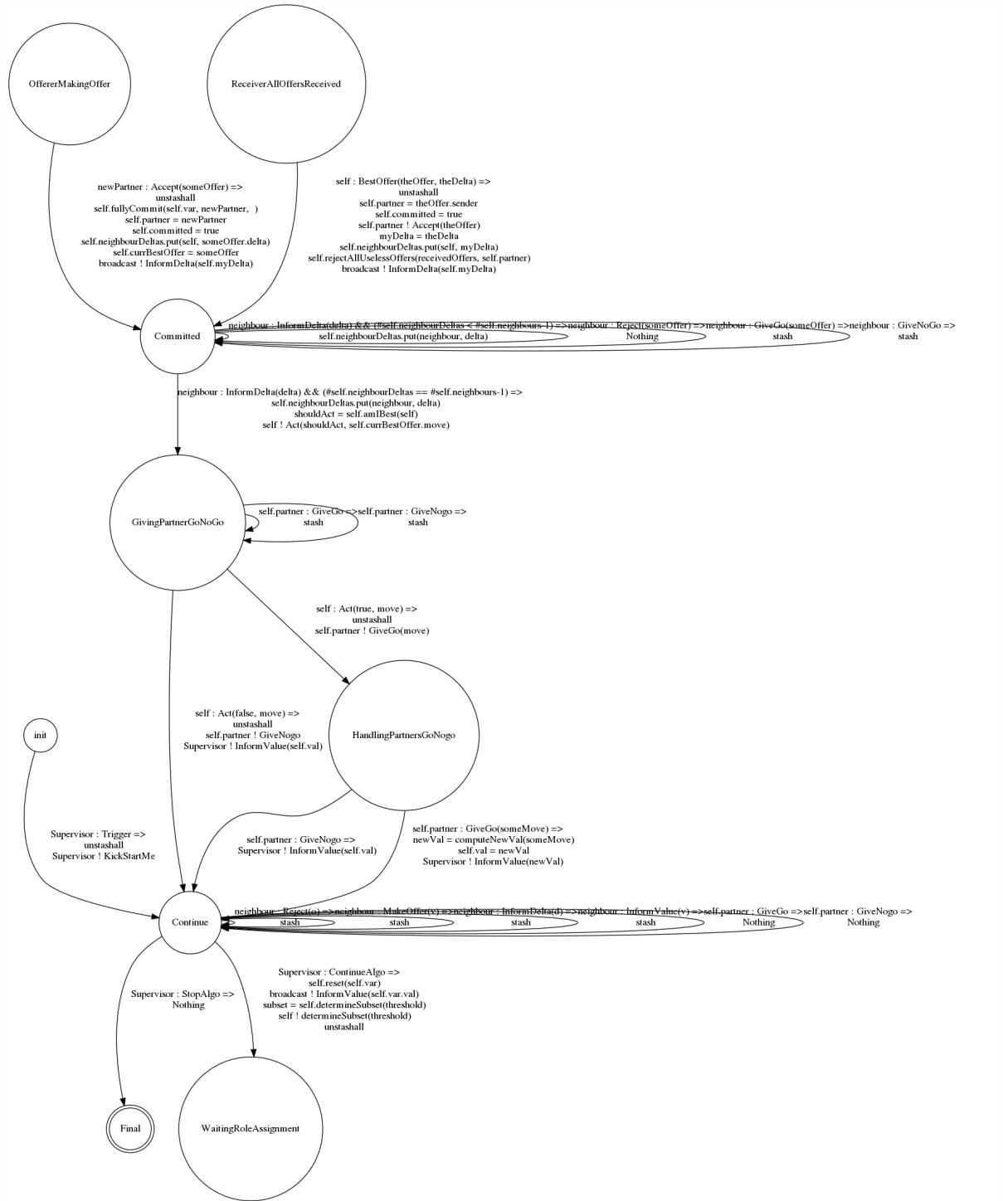


Figure 5: Committed agent



will compute whether it should act or not based on neighbourhood deltas and send either `Act(true)` or `Act(false)` and transition to `ActSolo`.

#### 5.2.12 ActSolo

Here the uncommitted agent processes the `Act(true/false)` message, it will transition to `Continue` either way. Messages received in this state:

- `Act(true)`: `mind.value` will be updated to `retainedMove`, inform supervisor of current value with `InformValue(mind.value)`.
- `Act(false)`: nothing except inform supervisor of current value with `InformValue(mind.value)`.

#### 5.2.13 GivingPartnerGoNogo

We postulate that all agents adopt the same behaviour, they speak before they listen. Hence the agent might receive its partner's `GiveGo` or `GiveNoGo` message which will be stashed and handled in `HandlingPartnersGoNogo`. In any case, the agent handles the self-sent message `Act(true/false)`. Messages received in this state:

- `Act(false)`: it will send its partner a `GiveNoGo` signal and transition to `Continue` by sending the Supervisor the `InformValue(mind.value)`.
- `Act(true)`: the agent sends its partner a `GiveGo` and transitions to `HandlingPartnersGoNogo`.

#### 5.2.14 HandlingPartnersGoNogo

Here the agent is the best among its neighbours and will check whether its partner has a similar situation. No matter the message received, both will lead to transitioning to `Continue`, however other actions taken to transition differ. Messages received in this state:

- `GiveGo`: the potentially stashed `GiveGo` and `GiveNoGo` messages are handled. If the agent got a `GiveGo` from its partner, it will update its value `retainedMove` and inform the Supervisor of it
- `GiveNoGo`: the potentially stashed `GiveGo` and `GiveNoGo` messages are handled. If the agent got a `GiveNoGo` from its partner, it won't update its value and simply transition to `Continue`. The agent will not update its value

#### 5.2.15 Final

This state is reached coming from `Continue` in the case where the Supervisor sent `StopAlgo`. No messages are sent nor received here, it merely represents the end of the algorithm.

### 5.3 Agent's Mind

The agent's mental state can be considered as the variables which hold the agent's perception of its surroundings and of its inner state. In our case, each agent possesses several *personal* beliefs and several other *interpersonal*.

#### 5.3.1 Personal beliefs

- `mind.context` : is the current beliefs about variables in the vicinity including its own variable.
- `mind.isCommitted`: is a boolean indicating whether an agent is committed or not, makes the variable `mind.partner` relevant.
- `mind.partner`: is the neighbouring agent with which the agent is committed, only relevant when committed is set to true.
- `mind.receivedOffers`: are the offers receives by the agent during this turn, potentially empty ones.
- `mind.nbReceivedOffers`: is the number of all offers received in the current turn.
- `mind.deltas`: is a map containing the agent's and all its neighbours' current deltas.
- `mind.currBestOffer` : is the potential best offer the agent has, eventually none.

### 5.4 Messages

In order to carry out negotiations, exchange information and coordiante themselves, agents can opt for a variety of messages. Each message can only be handled in dedicated states and will therefore be stashed to be processed later on.

#### 5.4.1 Standard agent messages

- `KickStartMe`: is sent by the agent at the very begining of the first round of the process to indicate to the Supervisor that it needs to send it the initial `ContinueAlgo` message.
- `ChooseSubset(Offerer/Receiver)`: is sent by the agent to itself, sent in the `waitingForRole` state and then causing the transition towards either `OffererWaitingValues` or `ReceiverWaitingOffers`. This message determines the path the agent threads for the next two states before joinging again in the `Uncomitted/Uncomitted` states.

- **InformValue**(someVal): is sent by an agent to all neighbouring agents in order to inform them about the current value of the variable controlled by the sender agent.
- **InformDelta**(mind.deltas.get(self.variable)): is sent by an agent which has computed its solo or joint delta to its neighbourhood upon transitioning to either **Uncommitted** or **Uncommitted** states.
- **MakeOffer**(someOffer/None): is sent by an offerer. An offer contains all coordinated moves between the offerer and the designated receiver with their respective delta from the offerer's perspective. A *None* offer is sent by an offerer to all receivers which have not been chosen by it for this round.
- **Reject**(): is sent by a receiver to all the non-chosen offerers. It can also be sent by an offerer if it has been made an offer, in this case no computation is needed since the offer is declined straight away.
- **Accept**(someCoordinatedMove): is sent by a receiver to the best offerer upon having computed the best offer from all the received ones. It contains a couple consisting of the desired coordinated move and the delta achieved by it.
- **BestOfferForMe**(someOffer): is sent by the receiver agent to itself after having received all offers from its neighbourhood. This message contains the offer which must be selected.
- **Act**(): is sent by an agent, either offerer or receiver, it is sent when transitioning either from **Uncommitted** or **Uncommitted** to **GivingPartnerGoNogo**.
- **GiveGo**: is sent by the partner which has received a **Act**(true).
- **GiveNoGo**: is sent by the partner which has received a **Act**(false).
- **StopAlgo**: is sent by the supervisor to inform agents that they should do another full cycle.
- **ContinueAlgo**: is sent by the supervisor to the agents to inform them that they should stop.

## 5.5 Outline

The supervisor has a simple behaviour based on mainly two states. In one of them, it awaits for all the agent's values. Once all values have been collected, it switches to the second state, evaluating whether the full process had ended or not. From there, it will branch, either back to the other state or to the final one once the condition for ending is reached.

## 5.6 Supervisor states

### 5.6.1 General overview

- **WaitingForAgentValues**: the main state where supervisor waits for agents' information.
- **Start**: the state in which the supervisor starts, from there it broadcasts the `Triggermessage` to all agents and directly goes to `WaitingForAgentValues`.
- **DecidingToStopOrContinue**: state in which the supervisor checks whether the stopping condition has been met or not.
- **Finish**: the final state the supervisor ends in after the full algorithm has ended and the stopping condition is met.

### 5.6.2 Start

Transient state where the supervisor doesn't stay, no messages can be received here.

### 5.6.3 WaitingForAgentValues

This is the main state the supervisor spends most time in. Here it collects values from agents in order to make its decision. Messages received in this state:

- **KickStartMe**: this is directly answered with a `ContinueAlgo` message.
- **InformValue(someVal)**: these messages are put into the supervisors' `supervisorMind.currentContext` until it is filled. The last one triggers a transition to `DecidingToStopOrContinue`.

### 5.6.4 DecidingToStopOrContinue

Here the supervisor either continues or stops, according to the message it send itself before transitioning here from `WaitingForAgentValues`. Messages received in this state:

- **InformValue()**: is stashed to be handled in `WaitingForAgentValues`.

### 5.6.5 Finish

Final state of the automaton, no messages are received here.

## 5.7 Supervisor's mental state

The supervisor only handles one mental state variable: `supervisorMind.currentContext`. In it it stores the current values all variables have.



## 5.8 Supervisor messages

- **Trigger:** is the initial message sent once by the supervisor to all agents to launch the process.
- **ContinueAlgo:** is sent by the supervisor to indicate that another round should be carried out.
- **StopAlgo:** is sent by the supervisor when the halting condition has been met.

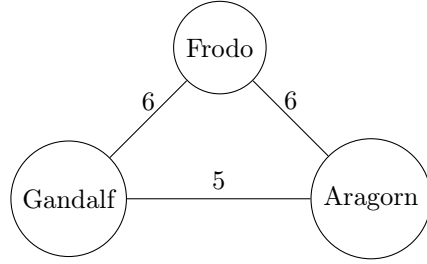


Figure 7: Favourable #1 case with sum of utilites = 17

## 6 Example 1: An audience with lady Galadriel

### 6.1 Introduction

Let us consider a company of three, sirs Gandalf, Frodo and Aragorn, visiting the halls of lady Galadriel in Lothlorien. The three of them are discussing which outfit they should wear and trying to find the best overall look their reduced fellowship can have. Worn out from the journey, their hosts show them to individual baths. Before leaving them to bathe and relax, their hosts leave them with two clean outfits sets of robes, made of either black velvet or white silk. After relaxing in the hot baths for a while, it will be up to them to decided which outfit they should wear.

This graph-colouring problem can be formalised as follows (see Figure 10):

- Set of agents, the three companions:  $= (G, F, A)$
- Set of variables, the colour of their each robe:  $= (R_1, R_2, R_3)$ .
- Domain of each variable, here the colour each robe can take:  $= (B, W)$
- The set of utility functions corresponding to the different combinations of each of their robes (see Table 1 and 2)
- $\alpha$  : simply the function (here the elf who gave each robe to each man!) mapping  $R_1$  to Gandalf,  $R_2$  to Frodo and  $R_3$  to Aragorn.

Here, the goal is to maximise the global **utility** function  $F_g$  -considered to be their charisma for intance- so they look as dashing as they can. We know by computing manually the sums that there are three best option which each yield a total of 17:

- $G=W, F=W, A=W$  (see Figure 7)
- $G=W, F=B, A=B$  (see Figure 8)
- $G=W, F=W, A=B$  (see Figure 9)

Since there are in this case only two possible values for each variable, namely black and white, we shall only note  $\delta$  for the opposite value since remaining with the current value always yields a  $\delta = 0$ . Let us look at the proceedings in detail.

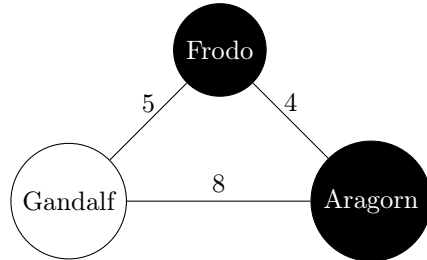


Figure 8: Favourable #2 case with sum of utilites = 17

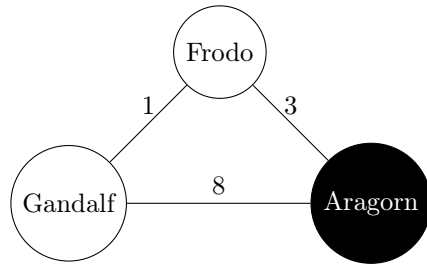


Figure 9: Favourable #3 case with sum of utilites = 17

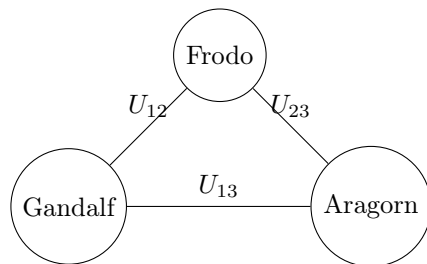


Figure 10: Graph representation with utility functions

Gandalf ( $R_1$ )	Frodo ( $R_2$ )	$U_{12}$
B	B	1
B	W	0
W	B	5
W	W	6

Frodo ( $R_2$ )	Aragorn ( $R_3$ )	$U_{23}$
B	B	4
B	W	0
W	B	3
W	W	1

Gandalf ( $R_1$ )	Aragorn ( $R_3$ )	$U_{13}$
B	B	3
B	W	0
W	B	8
W	W	5

Table 1: Utility functions according to robe colours

Gandalf ( $R_1$ )	Frodo ( $R_2$ )	Aragorn ( $R_3$ )	$U_{12}$	$U_{13}$	$U_{23}$	Total
B	B	B	1	3	4	8
B	B	W	1	0	0	1
B	W	B	0	3	3	6
B	W	W	0	0	6	6
W	B	B	5	8	4	<b>17</b>
W	B	W	5	5	0	10
W	W	B	6	8	3	<b>17</b>
W	W	W	6	5	6	<b>17</b>

Table 2: Full system values according to utilities

## 6.2 MGM approach

See Table 3.

1. **Round 0:** All three agents (G,A,F) start wearing black. Here each agent computes the difference  $\delta$  by which it can contribute to improve the situation as known to it. For instance
  - If Gandalf switches his robes to white,  $U_{12} = 5$  and  $U_{13} = 8$  so that gives deltas of respectively  $\delta(U_{12}) = 5 - 1 = 4$  and  $\delta(U_{13}) = 8 - 3 = 5$  with therefore  $\Delta = \sum_{\delta(U_{ij})} = 4 + 5 = 9$ . So his best offer is  $\Delta_G = +9$
  - Similarly, if Aragorn switches his robes to white,  $U_{13} = 0$  and  $U_{23} = 0$  so respectively  $\delta(U_{13}) = 0 - 3 = -3$  and  $\delta(U_{23}) = 0 - 4 = -4$ , with  $\Delta = \sum_{\delta(U_{ij})} = -3 + -4 = -7$ . So his best offer is a decrease in general utility  $\Delta_A = -7$ .

- Finally if Frodo switches his robes to white,  $U_{12} = 0$  and  $U_{23} = 3$  so respectively  $\delta(U_{12}) = 0 - 1 = -1$  and  $\delta(U_{23}) = 3 - 4 = -1$  with  $\Delta = \sum_{\delta(U_{ij})} = -1 + -1 = -2$ . So his best offer is  $\Delta_F = -2$ .

Here each agents now compares his own offer with the ones received and acts only if his offer is the max among all offers received. Since all three agents here are respectively neighbours  $\max(\Delta_G, \Delta_A, \Delta_F) = \Delta_G$  and hence Gandalf with his +9 is the one who acts and shifts to white robes.

2. **Round 1:** Now Gandalf is wearing white and both others are wearing black and a new evaluation of the situation for each begins, performed in the same way as for round 0. All  $\Delta$  are either null or negative, there is no improvement possible taking into account that each other agent is not going to change its colour (Nash Equilibrium). The game stops. Since this is one of the three optimal combinations, the gain for each agent will be 0 and hence the best setting has been reached.

Round	$R_1$	$R_2$	$R_3$	$U_{12}$	$U_{13}$	$U_{23}$	c(Gandalf)	c(Aragorn)	c(Frodo)	$F_g$
0	B	B	B	1	3	4	1+3=4	3+4=7	1+4=5	8 ?
1	W	B	B	5	8	4	5+8=13	8+4=12	5+4=9	17 ?

Table 3: MGM evolution of  $F_g$  and agent's contribution

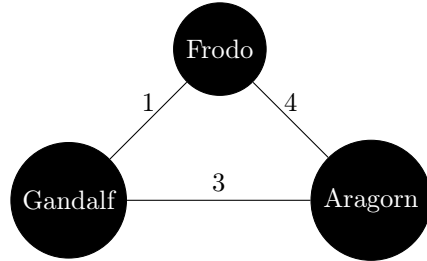


Figure 11: Round 0 situation

### 6.3 MGM-2 approach

This situation unravelled with our friends being unable to communicate with each other. Each time they had to put on their robes, exit the individual baths, meet and see what each of them was wearing, then they would think about the best change they could, announce it outloud, in a non RP manner "I can get a +x for our group charisma if you allow me to change my robes to y colour". This did result in an optimal solution, but it might have been achieved faster had they been allowed to communicate. This is what MGM2 allows us to do.

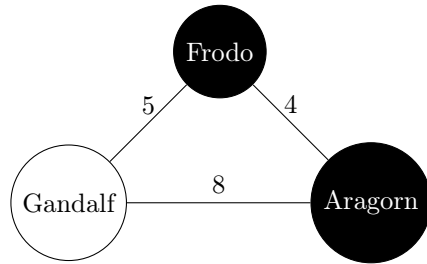


Figure 12: Round 1 situation

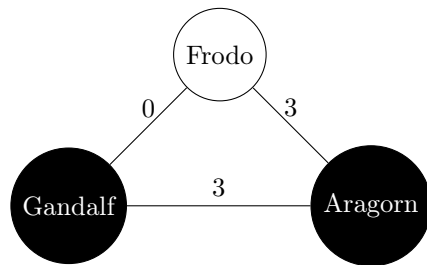


Figure 13: MGM-2 : round 0 situation

Let us now have a look at how this would have gone if they had been given a way to communicate with each other, this is the case in 2-coordinated algorithm, here MGM2. Imagine that, before leaving each of them to bathe, the elves give -them except for Gandalf, since he can easily do this without a thingummy- a thought stone. This allows them to create a mind link with one single other person and communicate with that person. Mind links can be renewed with different people as many times as they wish, they can not however be used to communicate with two people at the same time since the elves warn them that untrained novices such as them would catch a terrible headache for doing so. Now our companions can coordinate their decisions two by two.

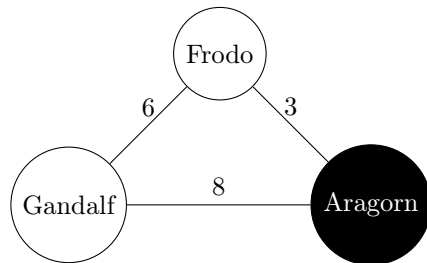


Figure 14: MGM-2 : round 1 situation

1. **Round 0** (see Figure 13):

- Agents inform each other of the current value of their variable.
- The set of Offerers and Receivers is determined at random, here  $O = \{G, F\}, R = \{A\}$
- **Gandalf**
  - Gandalf chooses a neighbour at random: F.
  - He receives an offer message from Frodo.
  - He declines right away since he is an Offerer and can't accept offers.
  - He computes all possible moves taking into account all possible combinations of both their values, so here B/B, B/W, W/B, W/W, for each of them he attaches the gain he would obtain in his neighbourhood.
  - He sends an offer message to Frodo.
  - He receives a reject message from Frodo.
  - He sets his commitment status to uncommitted.
  - He makes empty offers to all other neighbours (F, A)
  - He computes his gain by performing a solo change, just like in MGM.
  - He informs all his neighbours of his potential gain.
- **Frodo**
  - Frodo chooses a neighbour at random: G
  - He computes all possible joint moves.
  - He sends an offer message to Gandalf.
  - He receives an offer message from Gandalf.
  - He sends a reject message to Gandalf's offer.
  - He receives a reject message from Gandalf.
  - He sets his commitment status to uncommitted.
  - He makes empty offers to all other neighbours (G, A)
  - He computes his gain by performing a solo change, just like in MGM.
  - He informs all his neighbours of his potential gain.
- **Aragorn**
  - Aragorn received no offer since neither Frodo nor Gandalf chose him as potential partner.
  - He will perform a solo evaluation of his possible moves.
- All agents inform their neighbours of their best possible gains. Here Aragorn has a  $\delta = 0$  since changing to white would decrease his utility, so his best option is to remain black. Frodo computes a gain of 2:  $(U_{12} = 1) > 0 + (U_{23} = 4) > 3 \rightarrow 1 + 4 > 0 + 3$ . Gandalf computes a gain of 11:  $(U_{12} = 6) > 0 + (U_{13} = 8) > 3 \rightarrow 6 + 8 > 0 + 3$ .

- Gandalf wins this round and is allowed to act and update its value to white.

2. **Round 1** (see Figure 14):

- The process repeats. Now a new set of Offerers and Receivers is determined at random  $O = \{G\}, R = \{F, A\}$
- Gandalf chooses a partner at random: Aragorn. And makes an offer to him which is described in ??.
- Meanwhile Frodo being left without partner will proceed to compute its solo move.
- Aragorn receives Gandalf's offer and analyses it within its own neighbourhood.
- Here again a total gain of seventeen has been reached, each agent's best delta will be 0 and hence the algorithm will stop.

Gandalf ( $R_1$ )	Aragorn ( $R_3$ )	$U_{12}$	$U_{13}$	total
B	B	1	3	$4 < 14 \rightarrow \delta < 0$
B	W	1	0	$1 < 14 \rightarrow \delta < 0$
W	B	6	8	$14 = 14 \rightarrow \delta = 0$
W	W	6	5	$11 < 14 \rightarrow \delta < 0$

Table 4: Gandalf's offer to Aragorn for round 1

Table 5 summarizes the steps.

Round	$R_1$	$R_2$	$R_3$	$U_{12}$	$U_{13}$	$U_{23}$	c(Gandalf)	c(Aragorn)	c(Frodo)	$F_g$
0	B	W	B	0	3	3	$0+3=3$	$3+3=6$	$0+3=3$	6
1	W	W	B	6	8	3	$6+8=14$	$8+3=11$	$6+3=9$	17

Table 5: MGM-2 evolution of  $F_g$  and agent's contribution

## 6.4 Modelling of the problem with SCADCOP API

```

object RobeColouring extends App {
  val black = BooleanValue(false)
  val white = BooleanValue(true)
  val colourDomain = List(black, white)
  val gandalf = new Variable(id = 1, colourDomain)
  val aragorn = new Variable(id = 2, colourDomain)
  val frodo = new Variable(id = 3, colourDomain)

  val costGA = Array(Array(3.0, 0.0), Array(8.0, 5.0))

```



```

val costGF = Array(Array(1.0, 0.0), Array(5.0, 6.0))
val costFA = Array(Array(4.0, 0.0), Array(3.0, 1.0))

val cGA = new Constraint(gandalf, aragorn, costGA)
val cGF = new Constraint(gandalf, frodo, costGF)
val cFA = new Constraint(frodo, aragorn, costFA)
val pb = new DCOP(Set(gandalf, aragorn, frodo), List(cGA, cGF, cFA))

val a1 = new Context(pb)
a1.fix(Map(gandalf-> black, aragorn -> black, frodo -> black))

println(pb)
}

```

## 7 Example 2: Dalek's surveillance system

### 7.1 Introduction

Let us now switch to a different setting and for once, be on the *bad guy's* side. The Dalek's are building yet another stronghold in a dark corner of the galaxy. Since this is a new endeavour of theirs, only a small team of five has been sent to said planet. They know that, no matter how secret or insignificant this base might be, the Doctor is never that far... hence they try to devise a surveillance system to cover the small camp area. On the bright side, Dalek's neither sleep nor eat, so there is no question of organising shifts. Yet on the not-so-bright side, they haven't been designed with  $360^\circ$  sensors. Their sensors basically allow them to scan only a  $90^\circ$  portion at a time, and for simplification purposes, we will consider these  $90^\circ$  portions as the four cardinal directions, so not dalek can be scanning a South-East portion in our example, it will have to settle for scanning either South or East, tough luck. Last but not least, their chief, who knows about strategy, assigned them to fixed positions in the camp so these more junior Daleks wouldn't have to worry about where to be but only about where to look. To summarise, we can model the problem as follows (see Figure 15):

- Set of agents, the five Daleks:  $= (D_1, D_2, D_3, D_4, D_5)$
- Set of variables, the cardinal point they are looking at:  $= (C_1, C_2, C_3, C_4, C_5)$ .
- Domain of each variable, here the four directions available:  $= (N, E, S, W)$
- The set of utility functions corresponding to the different combinations of which direction they are looking at (see Table 6).
- $\alpha$  : in this case this parameter is somehow artificial since a sensor is part of the Dalek's anatomy, but we could think of the function as the Dalek's creation process, when each of them was given "arms", "eyes", sensors, etc...

Again, just like in our previous example, we shall consider agent and variable they control as one and the same.

In our example, the cost function will be defined as a maximisation of the surveillance space covered. In this case, Daleks who are the only ones capable of covering an area should be highly rewarded for doing so, and areas which can be covered by two Daleks who can also cover another area should ultimately be surveilled only by one of them.

Considering the position of the 5 Daleks, we can model the problem as the following constraints graph (see 15):

With utility functions modeled as given in 6, translating the necessity to cover a maximal number of zones.

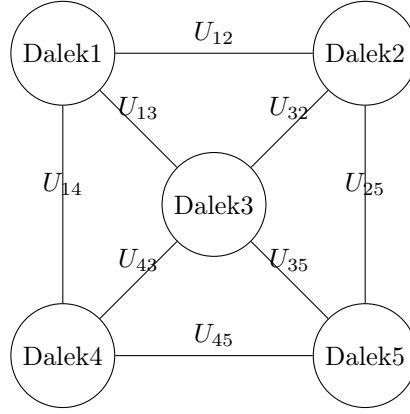


Figure 15: Dalek graph representation with utility functions

## 7.2 MGM resolution

1. **Round 0:** All Daleks are looking north with utilities as illustrated in Figure 16. Each Dalek computes its utility if it changed the direction where its looking, taking into account its neighbours are supposed not to change. For instance:

- $D_1$  computes  $U_{12} + U_{13} + U_{14}$ . Its goal is to adjust its value so that it maximizes the sum over  $U_{12} + U_{13} + U_{14}$ . Here, the domain of the controlled variable ranges over four different values, so it has 4 options, including the one of remaining North.
  - North (remaining still):  $10 + 5 + 5 = 20$
  - East :  $5 + 0 + 1 = 6$
  - South :  $5 + 1 + 0 = 6$
  - West :  $10 + 5 + 5 = 20$

From these computed utilities, it can now compute the change  $\delta$  it can achieve when shifting from its current value to the next, knowing that  $U_{current} = 20$ .

- North is the current orientaton, so  $\delta_N = 20 - U_{current} = 20 - 20 = 0$
- East:  $\delta_E = 6 - U_{current} = 6 - 20 = -14$
- South:  $\delta_S = 6 - U_{current} = 6 - 20 = -14$
- West:  $\delta_W = 20 - U_{current} = 20 - 20 = 0$

Here the best  $\delta$  which can be achieved is 0, either by staying North or shifting West.

- $D_2$  computes  $U_{12} + U_{23} + U_{25}$ 
  - North (remaining still):  $10 + 5 + 5$  which is the current utility  $U_{current}$

- East:  $10 + 5 + 5 = 20$
- South:  $5 + 1 + 5 = 11$
- West:  $5 + 5 + 5 = 15$

And from these, it can compute the  $\delta$ s:

- $\delta_N = 0$  since its the same position
- $\delta_E = 20 - U_{current} = 0$
- $\delta_S = 11 - U_{current} = -9$
- $\delta_W = 15 - U_{current} = -5$

Here the best  $\delta$  is again 0, this Dalek will not move since its current position is the best it can have in this setting.

- $D_3$  computes  $U_{13} + U_{23} + U_{34} + U_{35}$ 
  - North:  $U_{current} = 5 + 5 + 1 + 1 = 12$ ,  $\delta_N = 0$
  - East:  $5 + 5 + 1 + 0 = 11$ ,  $\delta_E = 11 - U_{current} = -1$
  - South:  $5 + 5 + 1 + 1 = 12$ ,  $\delta_S = 12 - U_{current} = 0$
  - West:  $5 + 5 + 0 + 1 = 11$ ,  $\delta_W = 11 - U_{current} = -1$

Hence the best  $\delta$  is the current one, no move would improve the situation.

- $D_4$  computes  $U_{14} + U_{34} + U_{45}$ 
  - North:  $5 + 1 + 1 = 7 = U_{current}$ ,  $\delta_N = 0$
  - East:  $5 + 1 + 1 = 7$ ,  $\delta_E = 0$
  - South:  $10 + 5 + 5 = 20$ ,  $\delta_S = 13$
  - West:  $10 + 5 + 5 = 20$ ,  $\delta_W = 13$

Here both E and W are good options since they both cause a gain of 13. Which one will be chosen depends on the implementation, here we shall postulate that the *closest* one is elected, hence W (see Figure 17).

- $D_5$  computes  $U_{25} + U_{35} + U_{45}$ 
  - North:  $5 + 1 + 1 = 7$
  - East:  $10 + 5 + 5 = 20$
  - South:  $10 + 5 + 5 = 20$
  - West:  $5 + 1 + 1 = 7$

Here both E and S are good with a  $\delta = 13$ , we assume E is chosen (see Figure 17).

Now that each Dalek has computed its utility and potential gain, they announce it. Both  $D_4$  and  $D_5$  have a potential gain of 13, so the way in which ties are handled depends on the implementation. Here we assume  $D_4$  wins the tie and is allowed to update its value. Resulting in the situation 1 graph. Because  $D_4$  will update its value, its neighbourhood

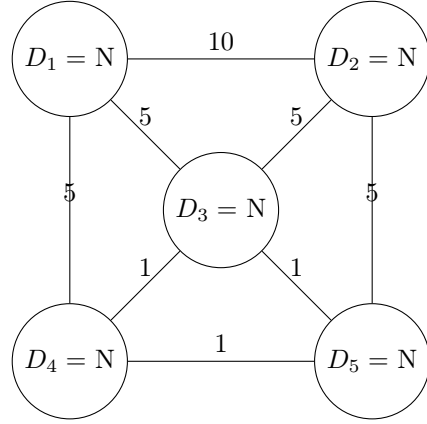


Figure 16: MGM situation 0: all Daleks looking North

consisting of  $\{D_1, D_3, D_5\}$  can't act this round because they have all received a gain message which was higher than theirs. Yet,  $D_2$  not being in the neighbourhood could act. However, despite not being in  $D_4$ 's neighbourhood,  $D_2$  had received a message from  $D_5$  with a higher gain than its own, so even if his gain had been positive (as opposed to null here) it would not have acted because of  $D_5$ 's message.

2. **Round 1:** The same process will go on as in round 0, here  $D_5$  will update to east.
3. **Round 2:** The same process is repeated, here no move can improve the situation since all Daleks are already in optimal positions, a Nash Equilibrium is reached.

### 7.3 MGM-2 resolution

### 7.4 Modelling of the problem with SCADCOP API

This problem can be modelled as follows with the scadcop library.

```
object DalekSurveillanceSystem extends App {
  val n = NominalValue("North")
  val e = NominalValue("East")
  val s = NominalValue("South")
  val w = NominalValue("West")

  val cardinalDirectionsDomain = List(n, e, s, w)

  val d1 = new Variable(id = 1, cardinalDirectionsDomain)
  val d2 = new Variable(id = 2, cardinalDirectionsDomain)
```

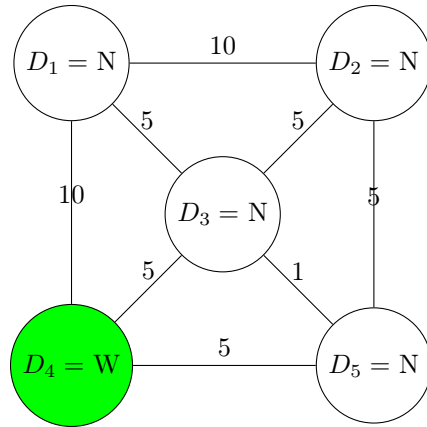


Figure 17: MGM situation 1: Dalek 4 has turned West

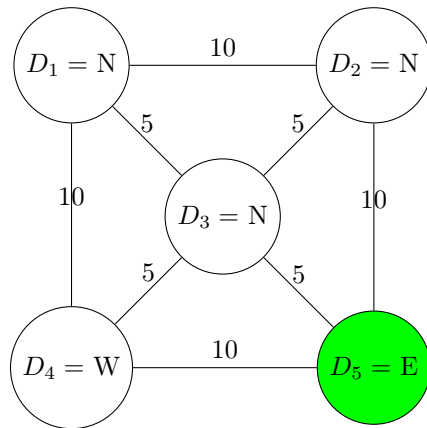


Figure 18: MGM situation 2: Dalek 5 has turned East

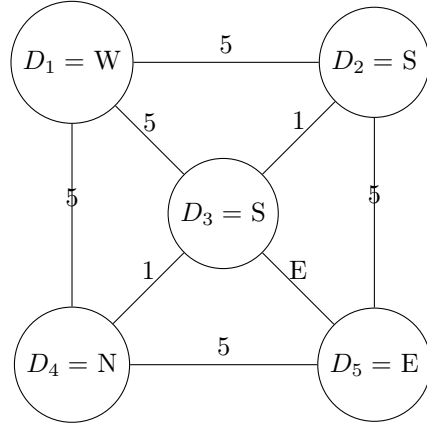


Figure 19: MGM-2 situation 0: random initialisation

```

val d3 = new Variable(id = 3, cardinalDirectionsDomain)
val d4 = new Variable(id = 4, cardinalDirectionsDomain)
val d5 = new Variable(id = 5, cardinalDirectionsDomain)

val cost12 = Array(Array(10.0, 10.0, 5.0, 5.0),
                    Array(5.0, 5.0, 1.0, 0.0),
                    Array(5.0, 5.0, 1.0, 1.0),
                    Array(10.0, 10.0, 5.0, 5.0))

val cost13 = Array(Array(5.0, 5.0, 5.0, 5.0),
                    Array(0.0, 1.0, 1.0, 1.0),
                    Array(1.0, 0.0, 1.0, 1.0),
                    Array(5.0, 5.0, 5.0, 5.0))

val cost14 = Array(Array(5.0, 5.0, 10.0, 10.0),
                    Array(1.0, 1.0, 5.0, 5.0),
                    Array(0.0, 1.0, 5.0, 5.0),
                    Array(5.0, 5.0, 10.0, 10.0))

val cost23 = Array(Array(5.0, 5.0, 5.0, 5.0),
                    Array(5.0, 5.0, 5.0, 5.0),
                    Array(1.0, 0.0, 1.0, 1.0),
                    Array(0.0, 1.0, 1.0, 1.0))

val cost25 = Array(Array(5.0, 10.0, 10.0, 5.0),
                    Array(5.0, 10.0, 10.0, 5.0),
                    Array(0.0, 5.0, 5.0, 1.0),
                    Array(1.0, 5.0, 5.0, 1.0))

```

```

val cost34 = Array(Array(1.0, 1.0, 5.0, 5.0),
                    Array(1.0, 1.0, 5.0, 5.0),
                    Array(1.0, 0.0, 5.0, 5.0),
                    Array(0.0, 1.0, 5.0, 5.0))

val cost35 = Array(Array(1.0, 5.0, 5.0, 1.0),
                    Array(0.0, 5.0, 5.0, 1.0),
                    Array(1.0, 5.0, 5.0, 0.0),
                    Array(1.0, 5.0, 5.0, 1.0))

val cost45 = Array(Array(1.0, 5.0, 5.0, 1.0),
                    Array(1.0, 5.0, 5.0, 0.0),
                    Array(5.0, 10.0, 10.0, 5.0),
                    Array(5.0, 10.0, 10.0, 5.0))

val constr12 = new Constraint(d1, d2, cost12)
val constr13 = new Constraint(d1, d3, cost13)
val constr14 = new Constraint(d1, d4, cost14)
val constr23 = new Constraint(d2, d3, cost23)
val constr25 = new Constraint(d2, d5, cost25)
val constr34 = new Constraint(d3, d4, cost34)
val constr35 = new Constraint(d3, d5, cost35)
val constr45 = new Constraint(d4, d5, cost45)

val pb = new DCOP(Set(d1, d2, d3, d4, d5), List(constr12, constr13, constr14,
val a1 = new Context(pb)
a1.fix(Map(d1-> n, d2 -> n, d3 -> n, d4 -> n, d5 -> n))
}

```



$D_1$	$D_2$	$U_{12}$	$D_1$	$D_3$	$U_{13}$	$D_1$	$D_4$	$U_{14}$	$D_2$	$D_3$	$U_{23}$
N	N	10	N	N	5	N	N	5	N	N	5
N	E	10	N	E	5	N	E	5	N	E	5
N	S	5	N	S	5	N	S	10	N	S	5
N	W	5	N	W	5	N	W	10	N	W	5
E	N	5	E	N	0	E	N	1	E	N	5
E	E	5	E	E	1	E	E	1	E	E	5
E	S	1	E	S	1	E	S	5	E	S	5
E	W	0	E	W	1	E	W	5	E	W	5
S	N	5	S	N	1	S	N	0	S	N	1
S	E	5	S	E	0	S	E	1	S	E	0
S	S	1	S	S	1	S	S	5	S	S	1
S	W	1	S	W	1	S	W	5	S	W	1
W	N	10	W	N	5	W	N	5	W	N	0
W	E	10	W	E	5	W	E	5	W	E	1
W	S	5	W	S	5	W	S	10	W	S	1
W	W	5	W	W	5	W	W	10	W	W	1
$D_2$	$D_5$	$U_{25}$	$D_3$	$D_4$	$U_{34}$	$D_3$	$D_5$	$U_{35}$	$D_4$	$D_5$	$U_{45}$
N	N	5	N	N	1	N	N	1	N	N	1
N	E	10	N	E	1	N	E	5	N	E	5
N	S	10	N	S	5	N	S	5	N	S	5
N	W	5	N	W	5	N	W	1	N	W	1
E	N	5	E	N	1	E	N	0	E	N	1
E	E	10	E	E	1	E	E	5	E	E	5
E	S	10	E	S	5	E	S	5	E	S	5
E	W	5	E	W	5	E	W	1	E	W	0
S	N	0	S	N	1	S	N	1	S	N	5
S	E	5	S	E	0	S	E	5	S	E	10
S	S	5	S	S	5	S	S	5	S	S	10
S	W	1	S	W	5	S	W	0	S	W	5
W	N	1	W	N	0	W	N	1	W	N	5
W	E	5	W	E	1	W	E	5	W	E	10
W	S	5	W	S	5	W	S	5	W	S	10
W	W	1	W	W	5	W	W	1	W	W	5

Table 6: Binary utility functions according to surveillance directions

Sit'	$D_1$	$D_2$	$D_3$	$D_4$	$D_5$	$U_{12}$	$U_{13}$	$U_{14}$	$U_{23}$	$U_{25}$	$U_{34}$	$U_{35}$	$U_{45}$	$U_g$
0	N	N	N	N	N	10	5	5	5	5	1	1	1	10+5+5 +5+5+ 1+1+1 =33
1	N	N	N	W	N	10	5	5	5	5	1	1	1	10+10 +5+5+ 5+5+ 5+1 =46
1	N	N	N	W	E	10	5	5	5	10	1	5	10	10+10 +10+10 +5+5 +5+5 =60

## 8 Glossary of technical terms

- **Asynchronous:** agents take decisions whenever they have collected the information they need to do so (i.e. received messages), not waiting for a particular general phase or timing.
- **Anytime:** anytime algorithms can be stopped at any moment and still yield a valid solution. They typically work by iteratively incrementing a valid base solution, each time yielding a more profitable solution. If not stopped artificially, they will come to an end and converge upon reaching a Nash Equilibrium.
- **Consistent assignment:** assignment which respects all constraints.
- **Cost function:** the function which is to be minimised in a DCOP (equivalent to utility function in maximisation)
- **Distributed Stochastic Search Algorithm (DSA):** an algorithm akin to MGM where agents which can act are selected through random.
- **Equilibrium (Nash):** a NE is a particular state of a non-zero sum adversarial game with  $n \in \mathbb{N}[2, +\infty[$  players. Non-zero sum adversarial games for  $n = 2$  players had been studied earlier by Von Neumann and Morgenstern (1944) but Nash was the one who generalised it to  $n$  players. Basically, it is a solution in which no player can benefit from changing its strategy. It assumes each agent acts independently, without collaboration or communication with any other agent and proves that *"a finite non-cooperative game always has at least one equilibrium point."*
- **Activation probability:** aims at diminishing potential issues linked to parallel processing.
- **Monotonic:** in this context usually used to refer to the strictly increasing property of the result function of anytime algorithms. Since it is monotonic (strictly increasing), no matter at which point we stop it, the result will always be better than the previous one.
- **Multigraph:** a multigraph is a graph that can have more than one edge between a pair of vertices.
- **Local variables:** the set of variables controlled by an agent, in our context a single one.
- **Partial assignment:** an assignment where a subset of variables is given a value.
- **Total assignment:** an assignment where the full set of variables is given a value, such assignment will be considered a solution to a DCOP if it satisfies all its cost functions.

- **Utility function:** the function which is to be maximised in a DCOP (equivalent to cost function in minimisation)

## 9 Conclusion

We detailed MGM and more particularly MGM-2 algorithms and presented examples of execution on toy DCOPs. We likewise detailed the automaton representing MGM-2 and proposed two examples of execution on toy problems.

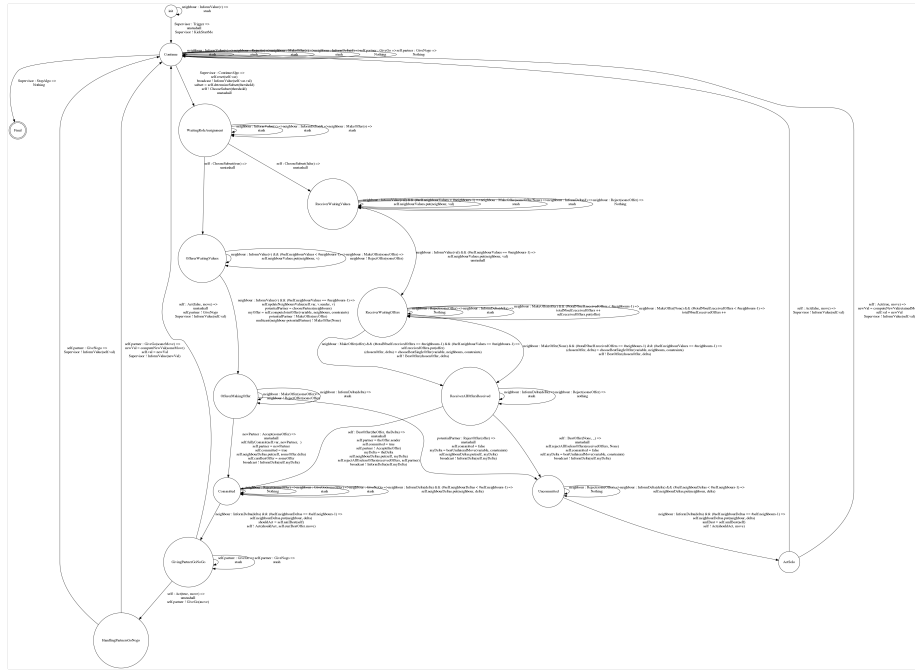


Figure 20: Complete MGM2 agent automaton

## 10 Annex

## References

- Ferdinando Fioretto, Enrico Pontelli, and William Yeoh. Distributed constraint optimization problems and applications: A survey. *Journal of Artificial Intelligence Research*, 61:623–698, 2018.
- Thomas Léauté, Brammert Ottens, and Radoslaw Szymanek. Frodo 2.0: An open-source framework for distributed constraint optimization. *Proceedings of the IJCAI'09 Distributed Constraint Reasoning Workshop (DCR'09)*, pages 160–164, 2009. URL <http://infoscience.epfl.ch/record/146585>.
- Rajiv T Maheswaran, Jonathan P Pearce, and Milind Tambe. Distributed algorithms for dcop: A graphical-game-based approach. In *Proc. of the ISCA 17th International Conference on Parallel and Distributed Computing Systems*, pp. 432–439, pages 432–439, 2004.
- Pierre Rust, Gauthier Picard, and Fano Ramparany. pyDCOP, a DCOP library for IoT and dynamic systems. In *International Workshop on Optimisation in Multi-Agent Systems (OptMAS@AAMAS 2019)*, 2019.
- Matthew E Taylor, Manish Jain, Yanquin Jin, Makoto Yokoo, and Milind Tambe. When should there be a "me" in "team"?: distributed multi-agent optimization under uncertainty. In *AAMAS*, pages 109–116, 2010.
- J. Von Neumann and O. Morgenstern. *Theory of games and economic behavior*. Princeton University Press, Princeton, NJ, US, 1944.