Pierre Rust, Gauthier Picard, Fano Ramparany

Résilience et auto-réparation de processus de décisions multi-agents. Application à l'auto-configuration d'environnements intelligents

Volume 3, nº 5-6 (2022), p. 587-623.

DOI not yet assigned

La Revue Ouverte d'Intelligence Artificielle est membre du Centre Mersenne pour l'édition scientifique ouverte
www.centre-mersenne.org
e-ISSN : pending

Revue Ouverte d'Intelligence Artificielle
Volume 3, n° 5-6, 2022, 587-623

Résilience et auto-réparation de processus de décisions multi-agents. Application à l'auto-configuration d'environnements intelligents

Pierre Rust[a], Gauthier Picard[b], Fano Ramparany[a]

[a] Orange Labs, France

[b] ONERA/DTIS, Université de Toulouse
E-mail : pierre.rust@orange.com, gauthier.picard@onera.fr, fano.ramparany@orange.com.

Résumé. — Dans cet article, nous définissons la notion de k-résilience de graphes de calculs en support aux décisions d'agents opérées sur des systèmes dynamiques. Nous proposons une méthode d'auto-réparation de la distribution des calculs, DRPM[DMCM], afin d'assurer la continuité des décisions collectives suite à la disparition d'agents, grâce au déploiement de réplicas de calculs. Nous nous intéressons ici à la réparation de processus d'optimisation sous contraintes, où les calculs sont des variables de décision ou des contraintes distribuées sur l'ensemble des agents. Nous appliquons la modélisation sous contraintes distribuées à un problème de coordination multi-agents d'objets dans le cadre de la maison intelligente (SECP) afin d'y appliquer nos techniques de réparation. Nous évaluons expérimentalement les performances de DRPM[DMCM], sur différentes topologies de systèmes opérant des algorithmes (A-MaxSumou A-DSA) pour résoudre des problèmes classiques (aléatoire, coloration de graphe, Ising) et des instances de SECP, alors que des agents disparaissent en cours de fonctionnement.

Mots-clés. — DCOP, résilience, auto-réparation, environnement intelligent.

1. Introduction

Nous examinons ici le problème de la répartition d'un ensemble de *calculs* étayant les décisions sur un ensemble d'agents incarnés dans des objets physiques (ou *nœuds*), comme des robots, des capteurs ou des véhicules autonomes.

Les décisions collectives et coordonnées sont organisées dans un graphe de calculs, où les sommets représentent des calculs et les arcs représentent une relation de dépendance entre les calculs. Une telle organisation est utilisée dans de nombreux modèles de décision, comme par exemple les graphes de facteurs et les graphes de contraintes utilisés lors de la résolution de problèmes d'optimisation distribuée sous contraintes (DCOP) [7], ou les algorithmes pour graphes de calculs tels que ceux adressés par Pregel ou d'autres frameworks basés sur BSP [21]. Bien que ces dernières structures ciblent généralement l'informatique en grappes hautes performances, nous considérons

ici l'Intelligence Ambiante (AmI) basée sur l'Internet des objets (IoT), des scénarios d'informatique embarquée ou des scénarios d'essaims de robots, où les calculs s'exécutent sur des nœuds distribués hétérogènes et où une coordination centrale pourrait ne pas être souhaitable, voire même impossible [2].

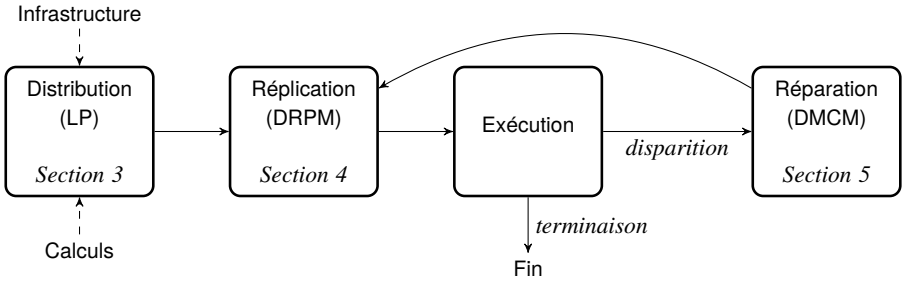


FIGURE 1.1 – Le cycle de vie de DRPM[DMCM].

Ici, les systèmes doivent pouvoir gérer les ajouts et les défaillances d'agents : lorsqu'un agent cesse de répondre, les autres agents du système doivent en assumer la responsabilité et exécuter les calculs partagés orphelins. De même, quand un nouvel agent est ajouté dans le système, il peut être utile de reconsidérer la distribution des calculs afin de tirer parti des capacités de calcul du nouveau venu, comme proposé dans [31]. Pour faire face à cette dynamique en maintenant les calculs et les décisions en cours alors que l'infrastructure évolue, une solution inspirée par les bases de données distribuées est la *réplication* [41, 40]. Nous proposons donc de répliquer les définitions de calculs plutôt que les données. Plus précisément, nous définissons la notion de *k-résilience*, qui caractérise les systèmes capables de fournir les mêmes fonctionnalités (ou prendre les mêmes décisions) même lorsque jusqu'à k nœuds disparaissent. L'adaptation de la distribution des calculs a été étudiée [31], mais aucune méthode distribuée en cours d'exécution n'a été évaluée. De plus, la disparition simultanée de plusieurs nœuds n'a pas non plus été prise en compte. Le problème de la distribution pour la résilience aux pannes a été étudié [3], mais la méthode proposée s'appuie sur un agent *dispatcher* et reste donc partiellement centralisée. D'autres travaux, comme [12], ont étudié la réplication pour garantir la tolérance aux pannes dans un système multi-agents (SMA), mais répliquent en général des agents là où nous proposons de répliquer les décisions affectées aux agents. Le processus général de notre approche est résumé à la figure 1.1.

Comme cadre d'application de technique de résilience, nous abordons un problème de configuration spontanée de scènes dans des environnements intelligents, via le paradigme multi-agents – plus particulièrement le cadre de l'optimisation distribuée (DCOP), où des algorithmes par envoi de messages mettent en œuvre un protocole de configuration. Ici, les objets font partie d'un SMA dont la tâche consiste à maximiser l'adéquation avec les besoins utilisateur tout en respectant des exigences d'économie d'énergie. Ce modèle de problème d'auto-configuration est appelé SECP, pour *Smart Environment Configuration Problem*.

Cet article est une version étendue de [34], et est structuré comme suit. La section 2 présente rapidement le cadre des DCOPs, au cœur de nos contributions. La section 3 définit la notion de distribution optimale de graphes de calculs sur une infrastructure physique. La section 4 expose la notion de k-résilience et présente une méthode de recherche itérative distribuée, DRPM, pour déployer des réplicas afin d'assurer la k-résilience. Nous avons conçu une méthode de réparation distribuée, DRPM[DMCM], basée sur un modèle DCOP utilisant la réplication pour adapter le déploiement de la décision à la suite de modifications apportées au système multi-agents physique (disparition des agents), dans la section 5. Nous nous intéresserons ensuite au cas particulier de la réparation des méthodes DCOP opérant sur des systèmes dynamiques, et proposons une méthode elle-même basée sur un DCOP pour réparer les processus de résolution. La section 6 présente le modèle de coordination d'objets intelligents, SECP. Nous évaluons expérimentalement nos contributions algorithmiques sur différentes topologies de systèmes dont la fonctionnalité consiste à exécuter des processus de raisonnement sous contraintes distribuées pendant que les agents quittent le système, dans la section 7. Nous exécutons notamment des versions asynchrones des algorithmes MaxSum et DSA dans de telles configurations dynamiques. La section 8 propose une discussion sur les travaux connexes aux contributions de cet article. Enfin, nous concluons le document avec des pistes de travaux complémentaires en section 9.

2. Optimisation sous contraintes distribuées

Cette section expose le cadre des DCOPs, au cœur de nos contributions, car à la fois utilisés pour modéliser des problèmes de décision multi-agents dans des environnements physiques (SECP par exemple) et pour réparer un tel processus en cours de fonctionnement.

2.1. Le cadre DCOP

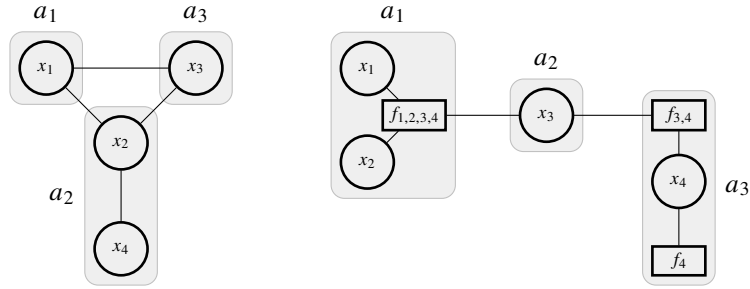
Alors que les méthodes de résolution pour les *problèmes de satisfaction (CSP) ou d'optimisation (COP)* sous contraintes [5] sont habituellement centralisées, une extension à ces modèles a émergé dans le domaine multi-agent, où le processus de raisonnement sous contraintes est partagé entre plusieurs agents intelligents, ayant chacun le contrôle d'une ou plusieurs variables de décision. Le cadre des DisCSP est le pendant distribué des CSP et ne considère que des contraintes dures pour formaliser des problèmes distribués [43]. Il a ensuite été étendu, puis remplacé dans les travaux plus récents, par le cadre des DCOPs.

Définition 2.1 (DCOP). — *Un* problème d'optimisation sous contraintes distribué *(ou DCOP) discret est formellement représenté par un tuple* $\langle \mathcal{A}, \mathcal{X}, \mathcal{D}, \mathcal{F}, \mu \rangle$*, où* $\mathcal{A} = \{a_1, \ldots, a_{|A|}\}$ *est un ensemble d'agents ;* $\mathcal{X} = \{x_1, \ldots, x_n\}$ *sont des variables discrètes, possédées par les agents ;* $\mathcal{D} = \{\mathcal{D}_1, \ldots, \mathcal{D}_n\}$ *est un ensemble de domaines finis, tels qu'une variable* x_i *prend ses valeurs dans* $\mathcal{D}_i = \{v_1^i, \ldots, v_k^i\}$*;* $\mathcal{F} = \{f_1, \ldots, f_m\}$ *est un ensemble de contraintes souples, où un* f_i *est une fonction qui définit un coût* $\in \mathbb{R} \cup \{\infty\}$ *pour chaque combinaison de valeurs de variables dans sa portée ;* $\mu : \mathcal{X} \to \mathcal{A}$ *est une fonction qui affecte le contrôle de chaque variable à un agent.*

Définition 2.2 (Solution). — *Une* solution *à un DCOP est une affectation complète σ qui minimise une fonction objectif global $F(\sigma)$ qui agrège les coûts individuels f_i's :* $\sigma^* = \text{argmin}_\sigma F(\sigma)$. *La somme est généralement utilisée comme fonction d'agrégation :* $\sigma^* = \text{argmin}_\sigma \sum_{f_i \in \mathcal{F}} f_i$.

Plus généralement, la notion de coût peut être remplacée par la notion d'utilité $\in \mathbb{R} \cup \{-\infty\}$. Dans ce cas, résoudre un DCOP devient un problème de maximisation de la somme totale des utilités (approche utilitariste).

De plus, les DCOPs sont souvent représentés par des modèles graphiques. Les *graphes de contraintes* sont habituellement utilisés pour représenter des DCOPs où les variables sont des nœuds du graphe, et les contraintes (toutes binaires) sont les arcs du graphe, avec l'ajout des nœuds composites représentant les agents et les variables qu'ils possèdent –voir la figure 2.1a. Ces nœuds forment un graphe de communication entre agents. Les *graphes de facteurs* sont également utilisés pour représenter des DCOPs. Ici, les contraintes (non nécessairement binaires et appelées également *facteurs*) sont explicitement représentées comme des nœuds du graphe, comme illustré dans la figure 2.1b.



(a) Graphe de contraintes (b) Graphe de facteurs

Figure 2.1 – Représentations graphiques pour les DCOPs

2.2. Algorithmes de résolution

Comme dans le cadre de l'optimisation sous contraintes classique, trouver une solution à un DCOP est NP-difficile en général. Un grand nombre d'algorithmes (et de variantes), optimaux ou approchés, ont été proposés par la communauté de chercheurs, très active. Dans cette section, nous ne serons pas en mesure de fournir un panorama complet des ces techniques. Pour cela nous redirigeons le lecteurs vers un article d'état de l'art très exhaustif [10]. Nous nous focaliserons ici sur les algorithmes qui sont utilisés dans cette étude, à savoir DSA, MGM et MaxSum.

Distributed Stochastic Search (DSA). — C'est une famille d'algorithmes de recherche locale, très légers, basée sur une idée très simple : les agents affectent initialement des valeurs aléatoires à leurs variables et évaluent régulièrement si la qualité de leur affectation locale, définie comme la somme des contraintes auxquelles ils sont

rattachés, peut être améliorée en sélectionnant d'autres valeurs [44]. Cette évaluation est basée sur la connaissances des valeurs actuelles des voisins (suivant le graphe de contraintes). Si la qualité peut être améliorée, les agents décident aléatoirement avec une *probabilité d'activation p*, de sélectionner la valeur correspondante et d'envoyer cette mise à jour aux voisins. Bien que non strictement *anytime*, DSA est un algorithme *itératif* qui peut être utilisé pour obtenir des affectations complètes à tout instant, en temps réel, avec une qualité des solutions s'améliorant en moyenne avec le temps. Cependant, dans le cas général, DSA ne fournit aucune garantie de monotonie : comme il n'y a pas de coordination dans le processus de décision, et que la connaissance des agents peut être obsolète, deux agents peuvent simultanément prendre des décisions contradictoires, résultant une diminution de la qualité de la solution.

Cinq variantes de ce principe ont été étudiées, se différenciant par la stratégie utilisée pour les changements de valeurs. DSA-B est considéré comme la variante la plus efficace dans le cas général. La valeur de p a également une très grande influence sur les performances, et peut montrer des propriétés de transition de phase [44]. Lorsque la bonne variante et la bonne probabilité p sont sélectionnées pour un problème donné, DSA fournit des solutions de très bonne qualité, avec peu de charge réseau et de calcul, ce qui le fait passer à l'échelle. A-DSA [11] est la version asynchrone de DSA, plus compatible avec un réel déploiement sur infrastructure physique et supportant la perte de messages.

Maximum Gain Message (MGM). — Cet algorithme met en œuvre un protocole de passage de messages de gain. [19]. Comme DSA, MGM est un algorithme de recherche locale qui peut prendre en charge des contraintes n-aires. Il est synchrone : à chaque tour, les agents calculent le changement maximum de qualité de solution, appelé *gain*, qu'ils peuvent obtenir en sélectionnant une nouvelle valeur, et envoient ce gain à leurs voisins. Un agent est alors autorisé à changer sa valeur uniquement si son gain est plus grand que celui de ses voisins. Ce mécanisme assure que deux variables impliquées dans la même contrainte ne changeront jamais de valeur au même tour. Ce processus se répète jusqu'à ce qu'une condition de terminaison soit validée (nombre de tours ou convergence). MGM garantit la monotonie du processus de recherche, en éliminant les aspects stochastiques de DSA. C'est une propriété très intéressante, au prix d'une plus grande tendance à rester piégé dans des optimums locaux. Pour répondre à cette limitation, une version coordonnée de MGM (MGM-k) a été proposée, où k agents peuvent coordonner un changement simultané de k valeurs au même tour, au prix d'une charge réseau assez importante, néanmoins. Aucune version asynchrone de MGM n'existe à ce jour.

MaxSum. — C'est un algorithme d'inférence approché [8], dérivé de l'algorithme par envois de messages et d'inférence sur modèle Bayésien, *max-product*, projeté dans un espace logarithmique. MaxSum est optimal sur des graphes de facteurs acycliques, mais approché sur des graphes cycliques. Il met en œuvre un processus de marginalisation des contraintes souples, et optimise les coûts pour chaque variable. Les affectations de valeurs prennent en compte leur impact sur la fonction de coût marginalisée. Il fonctionne sur un graphe de facteurs, et des messages de coûts circulent sur les arcs, des facteurs aux

variables et *vice et versa*. Quand un facteur ou une variable calcule deux fois le même message pour le même destinataire, il arrête la propagation et l'algorithme converge lorsqu'il n'y a plus aucune propagation. La terminaison est habituellement mise en œuvre en observant à la fois la convergence et en stoppant la propagation après un nombre de tours prédéterminé. En ajoutant tout simplement les messages de coût entrant, un agent peut évaluer à tout instant une approximation de la fonction marginale de ses variables, et ainsi sélectionner les valeurs qui maximisent la fonction de coût globale. Cela signifie que MaxSum peut être utilisé pour constamment mettre à jour une solution, sans attendre la terminaison, même dans le cas de problèmes dynamiques où les contraintes ou les variables changent. Les évaluations empiriques montrent qu'il peut obtenir de très bonnes qualités de solutions en un temps raisonnable, en comparaison d'autres algorithmes complets.

Notons que MaxSum peut être implanté de manière asynchrone, comme évoqué dans [8], avec des agents émettant des messages de mise à jour dès lors qu'ils reçoivent une mise à jour d'un de leurs voisins. Cette variante est appelée A-MaxSum et supporte la perte de messages.

3. Distribution des calculs

Le placement de calculs liés à des décisions sur des agents physiques peut avoir un impact important sur les performances du système : certaines distributions peuvent améliorer le temps de réponse, d'autres favoriser la charge de communication entre les agents et d'autres peuvent améliorer d'autres critères tels que la qualité de service, les coûts d'exploitation, ou l'impact sur l'environnement.

Soit $\mathbf{G} = \langle C, D \rangle$ un graphe connexe de calculs, où C est l'ensemble des calculs x_i, et D l'ensemble des arcs (i, j) représentant les dépendances entre calculs (qui impliquent l'envoi de messages). Soit $\mathcal{A}$ l'ensemble des agents pouvant héberger des calculs $x_i \in C$. Notons $\mu : C \mapsto \mathcal{A}$ la fonction associant les calculs aux agents et $\mu^{-1}(a_m)$ l'ensemble des calculs hébergés par a_m. Un agent ne peut héberger qu'une quantité limitée de calculs, contrainte par la *capacité* $\mathbf{w_{max}}(a_m)$ de l'agent, et le *poids* du calcul, $\mathbf{w}(x_i)$. Notons x_i^m le booléen spécifiant si x_i est hébergé par a_m.

Définition 3.1. — *Étant donnés un ensemble d'agents $\mathcal{A}$ et un ensemble de calculs C, une* ***distribution*** *est une fonction $\mu : C \mapsto \mathcal{A}$ qui affecte chaque calcul à un agent exactement et qui satisfait les contraintes de capacité des agents.*

Trouver une distribution est un problème de satisfaction de contraintes avec :

$$\forall x_i \in C, \quad \sum_{a_m \in \mathcal{A}} x_i^m = 1 \tag{3.1}$$

$$\forall a_m \in \mathcal{A}, \quad \sum_{x_i \in \mu^{-1}(a_m)} \mathbf{w}(x_i) \leqslant \mathbf{w_{max}}(a_m) \tag{3.2}$$

Lorsque les communications sont contraintes (e.g. IoT), la distribution devrait générer aussi peu de charge que possible et favoriser les liens les moins coûteux. Nous supposons que tous les agents peuvent communiquer entre eux, mais avec des coûts différents, représentés par une matrice : $\mathbf{route}(m, n)$ est le coût de communication entre a_m et a_n.

Soit $\mathbf{msg}(i,j)$ la taille des messages entre x_i et x_j, le coût entre x_i sur a_m et x_j sur a_n est :

$$\forall x_i,x_j \in C, \forall a_m,a_n \in \mathcal{A}, \mathbf{com}(i,j,m,n) = \begin{cases} \mathbf{msg}(i,j) \cdot \mathbf{route}(m,n) & \text{si } (i,j) \in D, m \neq n \\ 0 & \text{sinon} \end{cases} \quad (3.3)$$

où il n'y a aucun coût pour des calculs hébergés sur le même agent. Les coûts d'hébergement sur un agent sont modélisés par une fonction $\mathbf{c_{host}}$ affectant un coût pour chaque paire (a_m,x_j).

La qualité d'une distribution peut ainsi être évaluée par la fonction suivante, avec $\omega \in [0,1]$:

$$\omega \cdot \sum_{(i,j)\in D} \sum_{(m,n)\in \mathcal{A}^2} \mathbf{com}(i,j,m,n) \cdot x_i^m \cdot x_j^n \quad (3.4)$$

$$+ (1-\omega) \cdot \sum_{(x_i,a_m)\in C\times\mathcal{A}} x_i^m \cdot \mathbf{c_{host}}(a_m,x_i) \quad (3.5)$$

On peut ainsi définir la distribution des calculs sur les agents comme un problème d'optimisation sous contraintes où il s'agit de minimiser les coûts de communication entre deux agents *m* et *n* hébergeant des calculs *i* et *j* et les coûts d'hébergement du calcul *i* sur l'agent *m*) (3.5), tout en respectant les contraintes de capacité (3.2) et en s'assurant que tous les calculs soient bien hébergés (3.1). ω sert à pondérer l'importance de ces deux critères. Nous appelons ce problème CGDP pour *Computation Graph Distribution Problem*.

Pour résoudre le CGDP nous pouvons le transformer en programme linéaire, en linéarisant les objectifs (3.4) et (3.5) et contraintes (3.2) et (3.1). Nous appelons ce programme ILP-CGDP (*Integer Linear Program for CGDP*). Ce modèle est plus général que celui proposé dans la littérature [31], dédié aux graphes de facteurs sans coût d'hébergement.

ILP-CGDP étant NP-difficile, il peut facilement mettre à mal les solveurs commerciaux modernes. Il peut cependant être utilisé lors du démarrage du système, pour calculer de manière centralisée la distribution initiale de petites instances. Dans nos expérimentations nous l'utilisons pour évaluer la qualité des distributions obtenues par nos techniques.

En raison de cette difficulté à résoudre ILP-CGDP pour de grandes instances, nous proposons aussi GH-CGDP (*Greedy Heuristic for CGDP*), une heuristique gloutonne permettant de calculer une distribution sous-optimale. GH-CGDP place de manière itérative les calculs, en commençant par celui ayant le poids le plus élevé, sur l'agent possédant une grande capacité restante suffisante et qui induit les coûts de communication les plus faibles. En cas d'égalité, nous choisissons l'agent ayant la plus grande capacité résiduelle. Expérimentalement, GH-CGDP se révèle très efficace en produisant très rapidement des distributions de bonne qualité ; il faut cependant noter que dans le cas de systèmes très contraints (sur la capacité des agents) elle peut échouer à trouver une distribution valide là où ILP-CGDP réussit, dans la mesure où il est possible de résoudre le programme linéaire en un temps raisonnable.

4. Résilience et réplication

Dans un contexte centralisé, lorsque certains agents disparaissent, on peut utiliser ILP-CGDP ou GH-CGDP, discutés dans la section précédente, pour recalculer une nouvelle distribution optimale. Cependant, accéder à un tel calcul centralisé peut ne pas être possible ou souhaitable, en fonction des exigences du scénario d'application. Ainsi, nous étudions des techniques décentralisées pour faire face à une telle dynamique dans l'infrastructure.

4.1. k-résilience

Nous définissons la notion de k-résilience comme la capacité d'un système à se réparer et à fonctionner correctement même lorsque jusqu'à k agents disparaissent. Cela signifie qu'après une période de récupération, tous les calculs doivent être actifs sur exactement un agent et communiquer les uns avec les autres comme spécifié par le graphe $\mathbf{G}$.

Définition 4.1. — *Étant donnés les agents $\mathcal{A}$, les calculs C, et une distribution μ, un système est* ***k-résilient*** *si pour tout $F \subset \mathcal{A}, |F| \leqslant k$, une nouvelle distribution $\mu' : C \to \mathcal{A} \backslash F$ existe.*

Une condition préalable à la k-résilience est de toujours avoir accès à la définition de chaque calcul après disparition. Une approche consiste à conserver k *réplicas* (copies de définitions) de chaque calcul actif sur différents agents. À condition que les k réplicas soient placés sur des agents différents, il y aura toujours au moins un réplica restant après la disparition de n'importe quel groupe de k agents, comme en bases de données distribuées [21]. Ici, nous appliquons cette idée sauf que nous conservons des réplicas de définitions de calculs au lieu de données, ce qui implique que les calculs doivent être *stateless* ou que leur l'état doit être restaurable. La valeur maximale de k pour laquelle la k-résilience peut être atteinte dépend du système et surtout sur des capacités des agents. Il peut être déterminé, par exemple, grâce à la connaissance du taux de panne moyen des équipements déployés. Par ailleurs, la caractéristique de k-résilience du système devrait idéalement être elle aussi restaurée suite à une réparation, tant qu'il y a suffisamment de nœuds disponibles.

4.2. Placement des réplicas

Le problème de l'affectation de réplicas à des hôtes peut être considéré comme un problème d'optimisation, proche de la définition du CGDP. Idéalement, nous devrions optimiser l'emplacement des réplicas pour les coûts de communication et d'hébergement. Cela garantirait que lorsque des agents échouent, des réplicas sont disponibles sur les bons agents candidats. Cependant, l'espace de recherche pour cette optimisation est extrêmement large. Dans un système k-résilient avec n agents, il y a $\sum_{0<i\leqslant k} \binom{n}{i}$ scénarios de défaillance potentiels (jusqu'à k agents sur n peuvent échouer simultanément). Avec m calculs, le nombre de *configurations de réplications* possibles est $m \cdot \binom{n}{k}$. Le problème de la distribution optimale des réplicas sur un ensemble d'agents ayant des coûts et des capacités différentes peut être transformé en un problème de sac à dos multiple quadratique (QMKP) [35], qui est NP-difficile.

Ensuite, pour chacune de ces configurations de réplicas, il existe m^k *configurations d'activations* (pour chaque calcul orphelin, activer l'un des k réplicas). En supposant que nous puissions calculer le coût de toutes ces configurations, il ne serait toujours pas évident de savoir quel placement de réplicas serait le meilleur : on pourrait envisager celui permettant la meilleure configuration d'activation, ou celui permettant, en moyenne, des configurations d'activation de bonne qualité ou même celle offrant les meilleures configurations d'activation sur l'ensemble des scénarios de défaillance possibles. Évidemment, la définition de l'optimalité pour le placement de réplicas dépend très fortement du problème. Par conséquent, étant donné cette complexité, nous optons pour une approche heuristique distribuée, décrite dans la section suivante.

4.3. Méthode distribuée de placement

Nous proposons ici une méthode distribuée, DRPM (*Distributed Replica Placement Method*), pour déterminer les hôtes des k réplicas d'un calcul x_i donné. DRPM est une version distribuée de *iterative lengthening* (recherche de coût uniforme en fonction du coût des chemins) avec une mémorisation de chemins minimaux pour trouver les k meilleurs chemins, ne nécessitant que la connaissance du voisinage des agents. L'idée est d'héberger des réplicas sur les voisins les plus proches en ce qui concerne les coûts de communication et d'hébergement et les contraintes de capacité, en effectuant une recherche dans un graphe induit par des dépendances de calcul. Il génère une distribution de k réplicas (et le coût des chemins d'accès vers leurs hôtes) avec des coûts minimum pour un ensemble d'agents interconnectés. S'il est impossible de placer les réplicas k, en raison de contraintes de mémoire, DRPM place autant de calculs que possible et génère le meilleur niveau de résilience possible. Un agent d'hébergement, appelé *initiateur*, demande *itérativement* à chacun de ses voisins les moins chers, par ordre de coût croissant, jusqu'à ce que tous les réplicas soient placés. Les hôtes candidats sont considérés de manière itérative par ordre croissant de coût, qui comprend à la fois le coût de la communication (tout au long du chemin entre le calcul initial et son réplica) et le coût d'hébergement de l'agent hébergeant le réplica.

Définissons tout d'abord le graphe modélisant les coût de communication qui sera développé lors de la recherche :

Définition 4.2. — *Étant donné un graphe* $\langle C, D\rangle$*, le* **route**–*graph est un graphe pondéré* $\langle \mathcal{A}, E, w\rangle$ *où* $\mathcal{A}$ *est l'ensemble des nœuds,* E *est l'ensemble des arcs avec* $E = \{(a_m, a_n) | \exists (x_i, x_j) \in D, and\ \mu(x_i) = a_m, \mu(x_j) = a_n\}$ *et* $w : E \rightarrow \mathbb{R}$ *est la fonction de pondération* $w(a_m, a_n) = \mathbf{route}(m, n)$.

Afin de prendre aussi en compte l'hébergement, étendons le **route**–graph avec des nœuds supplémentaires attachés à chaque agent, excepté l'agent initiateur, par un arc pondéré par le coût d'hébergement, comme dans la figure 4.1.

Définition 4.3. — *Étant donné le* **route**–*graph*$\langle \mathcal{A}, E, w\rangle$ *et un calcul* x_i*, le* **route+host**–*graph est un graphe pondéré* $\langle \mathcal{A}', E', \mathbf{cost}\rangle$ *où* $\mathcal{A}' = \mathcal{A} \cup \widetilde{\mathcal{A}}$ *est l'ensemble des nœuds,* $\widetilde{\mathcal{A}} = \{\tilde{a}_m | a_m \in \mathcal{A}, a_m \neq \mu(x_i)\}$ *est l'ensemble des nœuds supplémentaires (un pour chaque élément de* $\mathcal{A}$*),* $E' = E \cup \{(a_m, \tilde{a}_m) | \tilde{a}_m \in \mathcal{A}\}$ *est l'ensemble des arcs et*

cost : $E' \to \mathbb{R}$ *est la fonction de pondération t.q.* $\forall a_m, a_n \in \mathcal{A}$, $\mathbf{cost}(a_m, a_n) = w(a_m, a_n)$, $\forall \tilde{a}_m \in \widetilde{\mathcal{A}}$, $\mathbf{cost}(a_m, \tilde{a}_m) = \mathbf{c_{host}}(a_m, x_i)$.

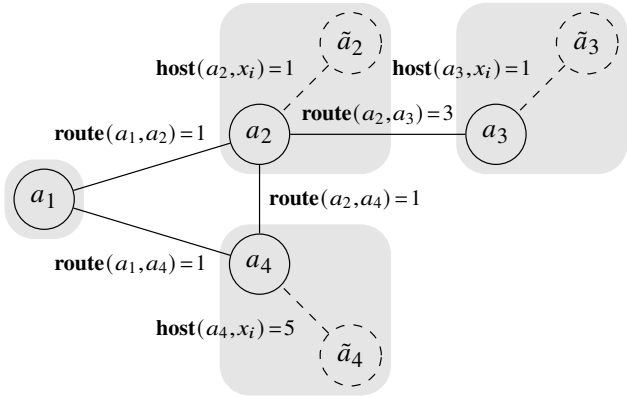


FIGURE 4.1 – Un **route+host**–graph avec 4 agents

Un **route+host**–graph est un graphe de recherche, développé à l'exécution et exploré pour un calcul particulier x_i. Chaque agent exécute autant de DRPM que de calculs à répliquer sur plusieurs **route+host**–graph s. Pour un **route+host**–graph donné, chaque agent peut encapsuler deux sommets (un dans $\mathcal{A}$ et son image dans $\widetilde{\mathcal{A}}$) et peut recevoir des messages concernant leurs deux sommets, et même des messages réflexifs. En outre, lors de l'évaluation de la possibilité pour un agent d'héberger un réplica pour x_i, nous nous assurons qu'il accepte uniquement s'il dispose d'une capacité suffisante pour activer tout sous-ensemble de taille k de ses réplicas. Bien sûr, cette contrainte est plus forte que ce qui pourrait être réellement nécessaire, ainsi, cette distribution n'est pas optimale en ce qui concerne les coûts d'hébergement, puisqu'un agent peut refuser d'héberger un calcul alors qu'il peut disposer de suffisamment de mémoire, au final. Comme de multiples instances de DPRM sont exécutées simultanément, une par calcul à répliquer, l'algorithme peut entraîner une distribution sous optimale, y compris en terme de communication. Cependant, si le placement de réplicas ne concerne qu'un seul calcul, la distribution est optimale puisque notre algorithme implémente une stratégie de recherche itérative par rallongement, ou *iterative lengthening*, avec mémorisation des chemins (pour éviter les boucles) [30, p. 90].

Exemple 4.4. — La figure 4.1 représente un **route+host**–graph avec 4 agents (gris), où a_1 cherche à répliquer x_i. Pour $k = 2$, DRMP place des réplicas sur a_2 (coût de $1+1=2$) et sur a_3 (coût de $1+3+1=5$) s'il y a assez de capacité sur ces agents, comme le chemin minimal pour héberger sur a_4 est plus élevé ($1+5=6$).

DRPM, étant une version distribuée de cet algorithme, utilise deux types de messages (`REQUEST` et `ANSWER`) avec les mêmes champs : (i) `current` : chemin de la requête, i.e. une liste contenant tous les sommets par lesquels les messages ont été transmis depuis l'initiateur jusqu'à celui qui reçoit le message actuel, (ii) `budget`, `spent` : budget restant

pour l'exploration du graphe et budget déjà dépensé sur le chemin courant, (iii) `known` : tableau associatif affectant le coût aux chemins déjà découverts à des sommets non visités, qui comptabilise les chemins les moins chers jusqu'à présent, (iv) `visited` : liste des sommets déjà visités, (v) `k` : le nombre restant de réplicas à héberger, (vi) x_i : calcul à répliquer. Les messages `REQUEST` descendent le long du graphe depuis l'initiateur, et correspondent au développement du graphe. Les messages `ANSWER` remontent les solutions (s'il y en a) jusqu'à l'initiateur.

Algorithme 1 : Traitement de `REQUEST`

Data : `current`, `budget`, `spent`, `known`, `visited`, `k`, x_i
1 `known` $\leftarrow$ `known` $\setminus$ `current`
2 **if** `me` $\notin$ `visited` **then**
3 `visited` $\leftarrow$ `visited` $\cup \{$`me`$\}$
4 **if** *can_host?*(x_i) **then**
5 `k` $\leftarrow$ `k` -1
6 add x_i to memory
7 **if** $k=0$ **then**
8 $a_p \leftarrow$ predecessor of `me` in `current`
9 send `ANSWER(current, budget+`**cost**`(me,`a_p`), spent-`**cost**`(me,`a_p`), known, visited, k,` x_i`)` to a_p
10 return
11 $p \leftarrow \text{argmin}_{e \in \{\text{paths in known starting with current}\}}$ `known`$[e]$
12 **if** $p \neq \emptyset$ **then**
13 $a_n \leftarrow$ successor of `me` in p
14 **if** **cost**`(me,`a_n`)` $\leqslant$ `budget` **then**
15 `current` $\leftarrow$ `current` $+ a_n$
16 send `REQUEST(current, budget-`**cost**`(me,`a_n`), spent+`**cost**`(me,`a_n`), known, visited, k,` x_i`)` to a_n
17 return
18 **foreach** $a_n \in \{a_m \mid (a_m,$`me`$) \in E', a_m \notin$ `visited`$\}$ **do**
19 **if** `spent+`**cost**`(me,`a_n`)` $<$ $\min_{e \in \{\text{paths in known leading to } a_n\}}$ `known`$[e]$ **then**
20 `known[current+`a_n`]` $\leftarrow$ `spent+`**cost**`(me,`a_n`)`
21 $a_p \leftarrow$ predecessor of `me` in `current`
22 send `ANSWER(current, budget+`**cost**`(me,`a_p`), spent-`**cost**`(me,`a_p`), known, visited, k,` x_i`)` to a_p

Au début, l'agent nécessitant une réplication de calcul initialise `known` avec les chemins de ses voisins directs dans le **route+host**–graph et envoie un message `REQUEST` avec un budget égal au chemin le moins cher connu. Ensuite, les agents traitent les messages comme expliqué dans le paragraphe suivant. Le protocole se termine lorsque toutes les réplicas possibles ont été placés (au plus k).

Àla réception d'un message `REQUEST` (algorithme 1), soit l'agent peut héberger un réplica et diminue ainsi le nombre de réplicas à placer, soit il transmet la demande à d'autres agents voisins. Dans le premier cas, si tous les réplicas ont été placés, l'agent répond à son prédécesseur avec un message `ANSWER`. Pour rechercher d'autres agents

Algorithme 2 : Traitement de `ANSWER`

```
   Data : current, budget, spent, known, visited, k, x_i
1  if k = 0 then
2      if me is root of current path then
3          terminate with target number of replicas placed
4      else
5          a_p ← predecessor of me in current
6          send ANSWER(current,
               budget+cost(me,a_p), spent-cost(me,a_p), known, visited, k, x_i) to a_p
7  else
8      p ← argmin_{e ∈ {paths in known starting with current}} known[e]
9      if me is root of current path then
10         if p ≠ ∅ then
11             budget ← budget+known[p]
12             a_n ← successor of me in p
13             current ← current + a_n
14             send REQUEST(current,
                   budget-cost(me,a_n), cost(me,a_n), known, visited, k, x_i) to a_n
15         else
16             terminate with fewer replicas than requested
17     else
18         if p ≠ ∅ then
19             a_n ← successor of me in p
20             if cost(me,a_n) ⩽ budget then
21                 current ← current + a_n
22                 send REQUEST(current, budget-cost(me,a_n),
                       spent+cost(me,a_n), known, visited, k, x_i) to a_n
23         a_p ← predecessor of me in current
24         send ANSWER(current,
               budget+cost(me,a_p), spent-cost(me,a_p), known, visited, k, x_i) to a_p
```

pour héberger des réplicas, s'il existe un chemin de coût minimum connu commençant par le chemin actuellement exploré qui est accessible avec le budget actuel, l'agent transmet la demande à son successeur dans ce chemin (avec un coût et un budget mis à jour). S'il n'y a pas un tel chemin, l'agent remplit le tableau known avec de nouveaux chemins menant à ses voisins dans le **route+host**–graph , si ces derniers améliorent les chemins connus existants, et renvoie known avec un message `ANSWER` à son prédécesseur pour qu'il explore ces nouvelles possibilités.

Àla réception d'un message `ANSWER` (algorithme 2), le message peut soit notifier que tous les réplicas ont été placés, soit qu'il reste au moins un réplica à placer. Dans le premier cas, si l'agent est l'initiateur, il termine l'algorithme en plaçant tous les réplicas demandés, sinon il renvoie la réponse à son prédécesseur, jusqu'à ce qu'il atteigne l'initiateur. Dans ce dernier cas, si l'agent est l'initiateur, il augmente le budget et envoie une demande au voisin le plus proche le cas échéant ; sinon cela signifie qu'il n'y a plus de chemin à explorer et que tous les réplicas ne peuvent être placés : l'agent termine

l'algorithme. Si l'agent n'est pas l'initiateur, mais qu'il existe un chemin accessible dans le budget actuel, il demande la réplication à son successeur dans le meilleur chemin connu, comme lors de la gestion des messages `REQUEST`. Enfin, s'il n'existe pas de tel chemin, il transmet simplement la réponse à son prédécesseur dans le chemin actuel.

Exemple 4.5. — Dans le cas de la figure 4.1, voici la séquence de messages qui sera générée. a_1, l'initiateur, peut envisager deux chemins $[a_1 \to a_2]$ et $[a_1 \to a_4]$ de coût identique, et initialise donc un budget de 1. a_1 envoie un message `REQUEST` à a_2, son premier successeur dans le premier chemin. a_2 répond avec un message `ANSWER` contenant deux nouveaux chemins $[a_1 \to a_2 \to \tilde{a}_2]$ de coût 2, et $[a_1 \to a_2 \to a_3]$ de coût 4. Le chemin $[a_1 \to a_2 \to a_4]$ de coût 2 n'est pas considéré car le meilleur chemin connu dans `known` vers a_4 est de coût 1. a_1, à la réception du message a_2 va explorer le meilleur chemin en envoyant un message `REQUEST` à a_4. a_4 rajoute un chemin à `known`, $[a_1 \to a_4 \to \tilde{a}_4]$ de coût 6 et répond à a_1. a_1 va ainsi envoyer un message `REQUEST` à a_2 pour explorer le meilleur chemin. a_2 peut héberger le réplica, car ce chemin même à une feuille $\tilde{a}_2$. a_2 peut même continuer l'exploration, car le prochain chemin dans `known` passe par lui et mène à a_3. a_2 envoie donc un message `REQUEST` à a_3, mais cette fois le nombre de réplicas à placer n'est plus de 2 mais de 1. a_3 peut rajouter le chemin $a_1 \to a_2 \to a_3 \to \tilde{a}_3$, qui est le meilleur actuel, et donc répondre qu'il peut héberger le dernier réplica. a_3 envoie donc un message `ANSWER` à a_2, qui le transmet à a_1. À la réception, comme tous les réplicas ont été placés, a_1 termine le placement.

Globalement, chaque agent est chargé de placer k réplicas de tous les calculs actifs qu'il héberge actuellement et exécute donc DRPM une fois pour chacun de ses calculs actifs. Ces multiples exécutions de DRPM peuvent être faites de manière séquentielle ou simultanée mais leur résultat dépend de l'ordre de réception des messages. Notez cependant que même si vous exécutez plusieurs DRPM simultanément, un agent ne dispose que d'une seule file de messages et traite les messages entrants de manière séquentielle, ce qui l'empêche d'accepter des réplicas qui dépasseraient sa capacité.

Théorème 4.6. — *DRPM se termine.*

Démonstration. — Pour k = 1, étant donné que les coûts du DRPM sont additifs et monotones et qu'il enregistre les chemins d'accès vers les sommets non consultés, il se termine comme un algorithme *iterative lengthening* classique, avec le chemin de coût minimum ou le chemin vide s'il n'y a pas assez de mémoire dans les agents pour héberger le calcul x_i. Pour $k > 1$, DRPM tente de placer chaque réplica de manière séquentielle. Il recherche d'abord le meilleur chemin (comme pour $k = 1$), puis exécute le même processus pour un deuxième meilleur chemin, et ainsi de suite jusqu'à ce que (i) les k réplicas soient placés ou (ii) il n'y ait pas assez de mémoire pour héberger le $n^{\text{ième}}$ réplica . La mémorisation dans `visited` garantit que le même chemin ne sera pas pris en compte deux fois. Ainsi, des itérations de recherche consécutives génèrent des chemins différents avec des coûts de chemin augmentant. Ainsi, dans le cas (i), DRPM se termine lorsque k réplicas ont été placés sur les k meilleurs hôtes ; et dans le cas (ii), il se termine lorsque $k' < k$ réplicas ont été placés, où k' est le nombre maximal de réplicas pouvant être placés. □

THÉORÈME 4.7. — *DRPM nécessite $O(b^l)$ messages pour terminer, b étant la largeur de l'arbre, $l = d/e$ le nombre d'itérations, d la profondeur de l'arbre, et $0 < e \leqslant 1$ le coût normalisé par étape.*

Démonstration. — Le pire cas pour DRPM survient lorsque les seuls hôtes possibles pour les réplicas sont les dernier explorés, où lorsque qu'il n'y a aucun hôte possible. Le nombre de nœuds est alors $(l)b + (l-1)b^2 + ... + (1)b^l$ qui est en $O(b^l)$. Cela nécessite deux fois plus de messages (requêtes et réponses), toujours en $O(b^l)$. □

Rappelons enfin que k est un paramètre à déterminer par le concepteur du système, et qu'il dépend des équipements et de l'infrastructure considérée. Cette méthode de distribution permet de s'assurer que k réplicas seront disponibles en cas de disparation d'au plus k nœuds. Cependant, suite à une telle disparition, il faudra redéployer les réplicas manquants, pour assurer à nouveau la k-résilience, et donc exécuter ces algorithmes, mais avec un nombre restreints de calculs et de nœuds.

5. Méthode d'auto-réparation

Les définition des calculs ayant été répliquées sur les agents, nous pouvons maintenant les utiliser pour réparer le système lors de la disparition d'un ou plusieurs agents (jusqu'à k). Nous modélisons ici ce problème de réparation comme problème d'optimisation distribuée sous contraintes (DCOP) [20], à résoudre par les agents eux-mêmes, à la suite d'apparitions ou de disparitions dans le système

5.1. Formulation DCOP

On note X_c l'ensemble des calculs candidats x_i qui peuvent ou doivent être déplacés lorsque l'ensemble des agents change. Pour chacun de ces calculs, notons A_c^i l'ensemble des agents candidats pour accueillir x_i. L'ensemble de tous les agents candidats, indépendamment des calculs, est noté $A_c = \cup_{x_i \in X_c} A_c^i$ et X_c^m désigne l'ensemble des calculs que l'agent a_m pourrait héberger. Décider quel agent $a_m \in A_c$ héberge chaque calcul $x_i \in X_c$ peut être traduit en un problème d'optimisation similaire à celui présenté dans la section 3, limitée à A_c et X_c. contrainte Pour s'assurer que chaque calcul candidat est hébergé sur exactement un agent, nous réécrivons les contraintes (3.1) pour chaque $x_i \in X_c$:

$$\sum_{a_m \in A_c^i} x_i^m = 1 \tag{5.1}$$

De même, les contraintes de capacité (3.2) peuvent être reformulées comme suit :

$$\sum_{x_i \in X_c^m} \mathbf{w}(x_i) \cdot x_i^m + \sum_{x_j \in \mu^{-1}(a_m) \setminus X_c} \mathbf{w}(x_j) \leqslant \mathbf{w_{max}}(a_m) \tag{5.2}$$

L'objectif de coût d'hébergement dans (3.5) peut être formulé de manière similaire à l'aide d'une contrainte souple pour chaque agent candidat a_m :

$$\sum_{x_i \in X_c^m} \mathbf{c_{host}}(a_m, x_i) \cdot x_i^m \tag{5.3}$$

Enfin, les coûts de communication dans (3.4) sont représentés par un ensemble de contraintes souples. Pour un agent a_m, les frais de communication liés à l'hébergement d'un calcul x_i peut être formulé comme la somme des coût des arcs coupés (x_i,x_j) à partir du graphe de calculs $\langle C,D\rangle$, (c'est-à-dire où $\mu^{-1}(x_j) \neq a_m$). Notons N_i les voisins de x_i dans le graphe de calcul. Quand un voisin x_n n'est pas un calcul candidat (c'est-à-dire qu'il ne sera peut-être pas déplacé et que $x_m \in N_i \setminus X_c$), le coût de communication de l'arc correspondant est simplement donné par $\mathbf{com}(i,j,m,\mu^{-1}(x_m))$. Pour les voisins qui pourraient être déplacés, le coût de la communication dépend de l'agent candidat choisi pour l'héberger, $\sum_{a_n \in A_c^j} x_j^n \cdot \mathbf{com}(i,j,m,n)$. Avec cela, nous pouvons écrire la contrainte souple de coût de communication pour l'agent a_m :

$$\begin{aligned} &\sum_{(x_i,x_j)\in X_c^m \times N_i \setminus X_c} x_i^m \cdot \mathbf{com}(i,j,m,\mu^{-1}(x_j)) \\ +&\sum_{(x_i,x_j)\in X_c^m \times N_i \cap X_c} x_i^m \cdot \sum_{a_n \in A_c^j} x_j^n \cdot \mathbf{com}(i,j,m,n)) \end{aligned} \tag{5.4}$$

Nous pouvons maintenant formuler le problème de réparation en tant que DCOP $\langle \mathcal{A},\mathcal{X},\mathcal{D},C,\mu\rangle$ où $\mathcal{A}$ est l'ensemble des agents candidats A_c, $\mathcal{X}$ et $\mathcal{D}$ sont respectivement l'ensemble des variables de décision x_i^m et leur domaine $\{0,1\}$ et C est composé de contraintes (5.1), (5.2), (5.3) et (5.4) appliquées pour chaque agent $a_m \in A_c$. (5.1) et (5.2) entraînent des coûts infinis en cas de violation, alors que (5.3) et (5.4) définissent directement les coûts à minimiser. La fonction μ affecte chaque variable x_i^m à un agent a_m.

5.2. Réparer avec MGM-2

Maintenant que notre problème de réparation a été exprimé en tant que DCOP, nous discutons de sa résolution en utilisant un algorithme DCOP, où les agents ne vont interagir qu'avec les agents hébergeant des réplicas pour des calculs communs. Plusieurs méthodes existent, telles que les algorithmes de recherche [20, 22] et les algorithmes d'inférence [27, 39, 7], pour en citer quelques-uns. En bref, en utilisant ces protocoles d'envoi de messages (synchrones ou non), les agents se coordonnent pour attribuer des valeurs à leurs variables.

Dans notre cas, nous optons pour une méthode légère, rapide et itérative, à savoir MGM [20]. Chaque agent assigne d'abord des valeurs aléatoires à ses variables et envoie les informations à tous ses voisins. En utilisant toutes les valeurs des voisins, un agent calcule le gain maximal s'il modifie sa valeur et l'envoie à tous ses voisins. Ensuite, en utilisant les gains de tous les voisins, l'agent modifie sa valeur si son gain est le plus grand. Ce processus se répète jusqu'à ce qu'une condition de terminaison soit remplie. Dans notre cas, les décisions nécessitent une coordination entre deux agents : pour déplacer un calcul de l'agent a_m à a_p, la variable binaire x_i^m doit prendre la valeur 0, tandis que *simultanément*, x_i^p doit passer de 0 à 1. Ce besoin de modifications simultanées justifie l'utilisation de MGM-2 [20]. La propriété de monotonie de MGM-2 répond très bien à nos besoins : une fois les contraintes dures (5.1) et (5.2) satisfaites, elles ne seront plus violées, tout en optimisant les contraintes souples (5.3) et (5.4).

En combinaison avec notre méthode de placement de réplicas, nous obtenons une méthode de réparation, appelée DRPM[MGM-2], qui peut être utilisée pour s'adapter au départ et à l'arrivée d'agents, en utilisant les définitions appropriées des ensembles A_c et X_c. Discutons le cas de départ d'agent, car c'est la situation la plus stressante. En supposant que le déploiement initial et le placement des réplicas aient été effectués au moment du démarrage du système, celui-ci exécutera le cycle de réparation suivant tout au long de son cycle de vie (voir la figure 1.1) : (a) Détecter départ / arrivée ; (b) Activer les réplicas des calculs manquants (avec MGM-2) ; (c) Placer les nouveaux réplicas pour les calculs manquants (à l'aide de DRPM) et poursuivre le fonctionnement nominal.

L'étape (a) suppose des mécanismes de découverte et de *keep alive* qui informent automatiquement certains agents de tout événement dans l'infrastructure. Ainsi, lorsqu'un agent a_m échoue ou est supprimé, nous considérons que tous les agents voisins de a_m dans le **route**–graph sont informés du départ. L'étape (b) déplace les calculs hébergés par les agents disparus A_d à d'autres agents. Les calculs candidats sont les calculs orphelins hébergés sur ces agents : $X_c = \cup_{a_m \in A_d} \mu(a_m)$. Pour éviter tout délai supplémentaire et toute communication lors de la phase de réparation, ces calculs orphelins doivent être affectés à des agents disposant déjà des informations nécessaires pour exécuter le calcul. Cela signifie que l'agent candidat pour un calcul orphelin x_i affecte l'ensemble d'agents encore disponibles hébergeant une réplique pour ce calcul : $A_c^i = \rho(x_i) \backslash A_d$. Dans un système k-résilient, tant que $|A_d| \leqslant k$, nous sommes sûrs qu'il y aura toujours au moins un agent dans A_c^i. Ainsi, l'étape (b) donne une affectation de chacun des calculs orphelins à l'un des agents hébergeant son réplica. L'étape (c) maintient un bon niveau de résilience dans le système en réparant la distribution des réplicas en utilisant DRPM sur un problème plus petit, car de nombreux réplicas sont déjà placés.

Notons que nous prenons pour hypothèse que le graphe de calculs est toujours connexe. Cela signifie qu'un agent ne peut être isolé des autres agents. Dans le cas contraire, si au moins une composante connexe est composée de plus de k agents, cela signifie, pour les autres composantes connexes que plus de k agents ont été retirés, donc cela ne rentre pas dans le cas de la k-résilience. Cependant, si le complémentaire de chaque composante connexe contient moins de k agents, chaque composante connexe sera capable de répliquer les calculs manquantd, et d'assurer, de manière concurrente et déconnectés, l'ensemble des fonctionnalités. Le cas de la reconnexion de telles composantes connexes n'est pas envisagée dans cet article.

6. Problème de configuration d'environnements intelligents

Dans cette section nous présentons et illustrons le problème de configuration d'environnement intelligent que nous abordons dans cet article.

6.1. Scénario exemple

Nous considérons le scénario d'AmI suivant. Notre système est composé d'objets connectés : ampoules, volets roulants, poste TV, des capteurs de luminosité et un détecteur de présence. Chacun de ces objets est défini par : (i) un identifiant unique (e.g. son adresse MAC), (ii) sa position (e.g. dans le salon), (iii) une liste de capacités (e.g. émettre

de la lumière ou jouer des vidéos), (iv) une liste d'actions (e.g. mettre la puissance à 2W), (v) une loi de consommation pour chaque action. Grâce à sa tablette, l'utilisateur peut configurer des comportements simples (ou *scènes*), faisant appel aux états de capteurs ou effecteurs comme des conditions de déclenchement d'actions domotiques. Par exemple, on peut configurer le système de telle sorte qu'un niveau de luminosité de 60 soit atteint dans le salon lorsque quelqu'un est présent dans la pièce. Notons que l'utilisateur n'a pas besoin de spécifier quelle ampoule doit être utilisée : le système décide de manière autonome de la meilleure façon d'atteindre l'objectif, en ouvrant les volets, allumant des ampoules, voire en allumant le poste TV si aucune autre source de lumière n'est disponible. Ceci signifie également que des ampoules peuvent être ajoutées ou supprimées en cours de fonctionnement, sans avoir à reprogrammer le système via la tablette. Nous souhaitons que notre système fournisse la configuration la plus économe en énergie pour chaque scène.

Exemple 6.1 (Spécification d'une scène). — La règle (6.1) définit une scène où la luminosité du salon doit être fixée à 60 lumens dès lors qu'une personne est présente dans la pièce :

```
IF    presence_living_room     =   1
THEN  light_level_living_room  ←   60
```
(6.1)

La règle (6.2) étend la règle (6.1) en ne déclenchant le comportement uniquement si la luminosité captée est inférieure à 60 lumens, et en fermant les volets du salon comme action additionnelle :

```
IF    presence_living_room      =   1
AND   light_sensor_living_room  <   60
THEN  light_level_living_room   ←   60
AND   shutter_living_room       ←   0
```
(6.2)

Ce problème de configuration peut être vu comme un problème d'optimisation visant à trouver des valeurs pour les effecteurs (e.g. puissance affectée à une ampoule), et aux objectifs de l'utilisateur (e.g. luminosité désirée dans le salon), tout en maximisant l'adéquation aux scènes définies par l'utilisateur et en minimisant la consommation énergétique globale.

6.2. Définition du problème

Soit $\mathfrak{A}$ l'ensemble des effecteurs disponibles. Notons $\mathcal{X}(\mathfrak{A})$ l'ensemble des variables d'état des effecteurs $i \in \mathfrak{A}$ (e.g. la puissance affectée à une lampe). Nous utiliserons la notation $\overline{x}_i$ pour faire référence à un état possible de la variable $x_i \in \mathcal{X}(\mathfrak{A})$, c.-à-d. $\overline{x}_i \in \mathcal{D}_{x_i}$ (domaine de x_i). Chaque effecteur i possède un coût d'activation, noté $c_i : \mathcal{D}_{x_i} \to \mathbb{R}$. Ce coût peut être directement dérivé de la loi de consommation énergétique de chaque objet. Notons $\mathcal{F}(\mathfrak{A}) = \{c_i | i \in \mathfrak{A}\}$. Parmi les valeurs possibles, tout effecteur i possède une valeur état "switched off", notée $\overline{0} \in \mathcal{D}_{x_i}$, avec un coût associé (très probablement 0).

Soit $\mathfrak{S}$ l'ensemble des capteurs disponibles, et $\mathcal{X}(\mathfrak{S})$ l'ensemble des variables d'état de ces capteurs. Nous noterons $\overline{s}_\ell \in \mathcal{D}_{s_\ell}$ l'état courant du capteur $\ell \in \mathfrak{S}$. Les valeurs

de capteurs ne sont pas directement contrôlables par le système : ce sont des valeurs en *lecture seule*.

Soit $\mathfrak{R}$ l'ensemble des règles de scènes définies par l'utilisateur. Chaque scène k est spécifiée comme une règle condition-action faisant appel aux objets disponibles (effecteurs et capteurs) et à leurs capacités. La partie condition de la règle est spécifiée comme une conjonction d'expressions booléennes faisant appel à l'état des effecteurs (e.g. la puissance de l'ampoule #1 est supérieure à 2W) ou l'état des capteurs (e.g. une présence est détectée dans le salon), et des prédicats binaires (e.g. >, <, =). La partie action d'une règle définit les valeurs *cibles* pour (i) des actions directes sur les effecteurs (e.g. puissance de l'ampoule #1) ou (ii) des actions indirectes (correspondant aux *objectifs de l'utilisateur*) sur des concepts de plus haut niveau d'abstraction (e.g. luminosité dans le salon). Les variables de décision correspondant à ces deux types d'actions sont appelées des *variables d'action de scène*.

Ces variables d'action de scène sont ainsi soit (i) des $x_i \in \mathcal{X}(\mathfrak{A})$ soit (ii) d'autres valeurs contraintes par les valeurs affectées à certains effecteurs (e.g. la luminosité du salon dépend de la puissance affectée aux deux ampoules de cette pièce). Notons y_j l'état d'une telle action de scène indirecte j (e.g. la luminosité dans la pièce). Chaque variable d'action de scène y_j dépend physiquement des valeurs de plusieurs effecteurs. Notons ce modèle de dépendance $\phi_j : (\prod_{\varsigma \in \sigma(\phi_j)} \mathcal{D}_\varsigma) \rightarrow \mathcal{D}_{y_j}$, où $\sigma(\phi_j) \subseteq \mathcal{X}(\mathfrak{A})$ est la portée de ce modèle, i.e. le sous-ensemble de variables influant sur y_j. Soit $\Phi = \{\phi_j\}$ l'ensemble de tous les modèles physiques entre effecteurs et valeurs cibles, et $\mathcal{X}(\Phi) = \{y_j\}$ l'ensemble des variables d'action de scène indirecte. De manière plus générale, un modèle de dépendance physique relie un ensemble d'objets – possédant une certaine capacité (e.g. émettre de la lumière, comme une ampoule ou un poste TV), dans une certaine pièce (e.g. salon) – à une valeur physique (e.g. luminosité) pouvant être mesurée par un capteur (e.g. capteur de lumière).

Exemple 6.2 (Modèle physique). — Nous pouvons considérer que le niveau de luminosité y_1 dans une pièce dépend de la puissance totale des objets émetteurs de lumière installés dans la pièce, i.e. ampoules x_1 et x_2, et poste TV x_3 :

$$y_1 = \phi_1(x_1, x_2, x_3) = 30x_1 + 30x_2 + 10x_3$$

Les poids affectés à chaque x_i dépendent de l'efficacité lumineuse de chaque objet [37].

Nous notons $\overline{x}_i^k$ (resp. $\overline{y}_j^k$) la valeur cible définie par l'utilisateur pour la variable d'action de scène x_i (resp. y_j) dans la règle k. Nous utiliserons $\overline{y}_j$ pour faire référence à un état possible de y_j, c.-à-d. $\overline{y}_j \in \mathcal{D}_{y_j}$ (domaine de y_j). Bien sûr, $\overline{x}_i^k \in \mathcal{D}_{x_i}$ et $\overline{y}_j^k \in \mathcal{D}_{y_j}$ pour tout i, j et k. Remarquons qu'une variable d'action de scène peut être utilisée dans plusieurs règles, mais qu'une règle ne peut spécifier qu'une seule valeur cible pour une variable d'action de scène.

Une règle de scène peut être soit *active* soit *inactive* en fonction de l'état des objets apparaissant dans la partie condition de la règle. Chaque scène active possède également une utilité de mise en œuvre, notée $u_k : \prod_{s \in \sigma(u_k)} \mathcal{D}_s \rightarrow \mathbb{R}$, avec $\sigma(u_k) \subseteq \mathcal{X}(\mathfrak{A}) \cup \mathcal{X}(\Phi)$ étant la portée de la règle (le sous-ensemble de variables utilisées dans cette règle). Plus les états des variables d'action de scène (de $\mathcal{X}(\mathfrak{A})$ et $\mathcal{X}(\Phi)$) sont proches des valeurs cibles

de l'utilisateur pour cette scène, plus l'utilité sera élevée. De plus, si la condition d'activation d'une règle (de $\mathcal{X}(\mathfrak{A})$ et $\mathcal{X}(\mathfrak{S})$) n'est pas vérifiée, l'utilité peut être neutre, i.e. égale à 0. On peut ainsi considérer les u_k comme étant fonctions de la distance entre les états des variables x_i (resp. y_j) et les valeurs cibles $\overline{x}_i^k$ (resp. $\overline{y}_j^k$). Nous noterons $\mathcal{F} = \{u_k | k \in \mathfrak{R}\}$.

Exemple 6.3 (Utilité de règle de scène). — Pour la règle (6.1), où s_1 est la valeur du capteur de présence, une fonction d'utilité envisageable est l'opposée de la *distance* entre la valeur courante de y_1 (luminosité) et la valeur cible $\overline{y}_1^1 = 60$:

$$u_1(y_1) = \begin{cases} -|y_1 - 60| & \text{si } s_1 = 1 \\ 0 & \text{sinon} \end{cases}$$

Voici une fonction d'utilité possible pour la règle (6.2), x_3 étant la puissance affectée à l'ampoule #3 :

$$u_1(y_1, x_3) = \begin{cases} -\sqrt{|y_1 - 60|^2 + |x_3|^2} & \text{si } s_1 = 1 \\ & \text{et } s_2 > 60 \\ 0 & \text{sinon} \end{cases}$$

Étant donnés tous les concepts et notations précédents, nous définissons le SECP comme suit :

Définition 6.4 (SECP). — *Étant donné un ensemble d'effecteurs $\mathfrak{A}$ (et leurs coûts respectifs $c_i \in \mathcal{F}(\mathfrak{A})$), un ensemble de capteurs $\mathfrak{S}$, un ensemble de règles de scènes $\mathfrak{R}$ (et leurs utilités respectives $u_k \in \mathcal{F}$), et un ensemble de modèles de dépendance physique Φ, le* problème de configuration d'environnements intelligents *(ou* SECP, *pour* Smart Environment Configuration Problem*)* $\langle \mathfrak{A}, \mathcal{F}(\mathfrak{A}), \mathfrak{S}, \mathfrak{R}, \mathcal{F}, \Phi \rangle$ *consiste à trouver la configuration des effecteurs qui maximise l'utilité des règles définies par l'utilisateur, tout en minimisant la consommation énergétique globale et en respectant les dépendances physiques.*

6.3. Formulation de SECP en DCOP

SECP peut être directement traduit en problème d'optimisation multi-objectif :

$$\begin{array}{ll} \min\limits_{x_i \in \mathcal{X}(\mathfrak{A})} & \sum\limits_{i \in \mathfrak{A}} c_i \\ \max\limits_{\substack{x_i \in \mathcal{X}(\mathfrak{A}) \\ y_j \in \mathcal{X}(\Phi)}} & \sum\limits_{k \in \mathfrak{R}} u_k \\ \textbf{avec} & \phi_j(x_j^1, \ldots, x_j^{|\sigma(\phi_j)|}) = y_j \\ & \forall y_j \in \mathcal{X}(\Phi) \end{array} \tag{6.3}$$

Le problème multi-objectif (6.3) consiste à minimiser les coûts énergétiques c_i et à maximiser la réponse aux besoins des utilisateurs u_k, tout en respectant les contraintes physiques de l'environnement. Ce problème peut être formulé comme un problème mono-objectif en agrégeant les deux objectifs, à condition que les échelles des u_k et c_i

soient *normalisées* ou *priorisées* (en utilisant des poids $\omega_u, \omega_c > 0$) :

$$\begin{aligned} \max_{\substack{x_i \in \mathcal{X}(\mathfrak{A}) \\ y_j \in \mathcal{X}(\Phi)}} \quad & \omega_u \sum_{k \in \mathfrak{R}} u_k - \omega_c \sum_{i \in \mathfrak{A}} c_i \\ \textbf{avec} \quad & \phi_j(x^1_{y_j}, \ldots, x^{|\sigma(\phi_j)|}_{y_j}) = y_j \\ & \forall y_j \in \mathcal{X}(\Phi) \end{aligned} \tag{6.4}$$

Afin de traduire le problème (6.4) en DCOP, nous devons encoder les contraintes dures correspondant aux dépendances physiques comme des facteurs notés φ_j, qui sont des contraintes souples d'utilité nulle ou infinie. Notons $\mathcal{F}(\mathbf{\Phi})$ l'ensemble des φ_j correspondants.

$$\varphi_j(x^1_j, \ldots, x^{|\sigma(\phi_j)|}_j, y_j) = \begin{cases} 0 & \text{si } \phi_j\left(x^1_j, \ldots, x^{|\sigma(\phi_j)|}_j\right) = y_j \\ -\infty & \text{sinon} \end{cases} \tag{6.5}$$

Grâce à l'équation (6.5), SECP peut être formulé comme un DCOP $\langle \mathcal{A}, \mathcal{X}, \mathcal{D}, \mathcal{C}, \mu \rangle$ où : $\mathcal{A}$ est l'ensemble des objets connectés ; $\mathcal{X} = \mathcal{X}(\mathfrak{A}) \cup \mathcal{X}(\Phi)$; $\mathcal{D} = \{\mathcal{D}_{x_i} | x_i \in \mathcal{X}(\mathfrak{A})\} \cup \{\mathcal{D}_{y_j} | y_j \in \mathcal{X}(\Phi)\}$; $\mathcal{C} = \mathcal{F} \cup \mathcal{F}(\mathfrak{A}) \cup \mathcal{F}(\mathbf{\Phi})$; μ est une fonction qui associe les variables et contraintes aux objets ; avec l'objectif suivant :

$$\max_{\substack{x_i \in \mathcal{X}(\mathfrak{A}) \\ y_j \in \mathcal{X}(\Phi)}} \quad \omega_u \sum_{k \in \mathfrak{R}} u_k - \omega_c \sum_{i \in \mathfrak{A}} c_i + \sum_{\varphi_j \in \mathcal{F}(\mathbf{\Phi})} \varphi_j \tag{6.6}$$

Un SECP étant traduit en DCOP suivant l'équation (6.6), il est immédiat de le représenter sous la forme d'un graphe de facteurs incluant tous les u_k, les c_i et les φ_j, ainsi que leurs variables associées. Un exemple de SECP pour un étage d'une maison intelligente est fournit dans la figure 6.1. Pour plus de clarté les règles de scène ont été omises.

7. Évaluation expérimentale

Analysons maintenant l'impact de la réparation sur les performances de processus de raisonnements sous contraintes – A-MaxSum et A-DSA – ainsi que la qualité des distributions obtenues après réparation. Nous allons étudier deux ensembles de problèmes : des benchmarks aléatoires et des SECPs.

7.1. Analyse sur des DCOP aléatoires

Notre premier ensemble d'évaluations se focalise sur des problèmes *benchmark* sur lesquels les DCOPs sont habituellement éprouvés.

7.1.1. *Cadre expérimental*

Pour évaluer notre cadre de réparation, nous générons des instances définies par trois composants : une définition de problème DCOP, une topologie multi-agents (l'infrastructure), et un scénario de perturbation.

Nous étudions dans cette section trois types de DCOPs, habituellement utilisés pour évaluer la qualité des solveurs DCOPs : (i) coloration de graphes aléatoires avec

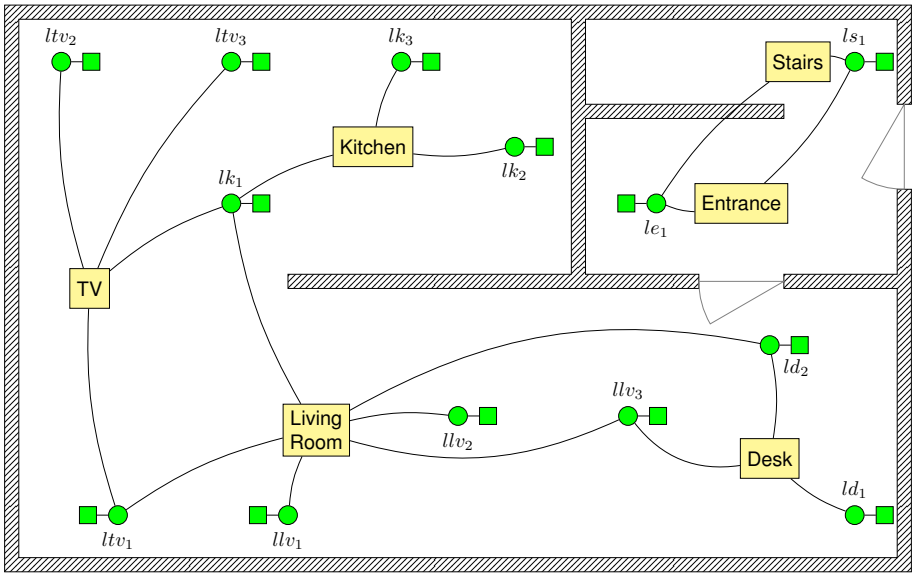


FIGURE 6.1 – Un graphe de facteurs pour un SECP sur l'étage d'une maison intelligente.

densité $p = 0.3$, (ii) coloration de graphes *scale free* [1], et (iii) modèles d'Ising. Ces paramétrages sont ceux habituellement utilisés dans la littérature.

Pour les colorations de graphes, chaque nœud est associé à une variable et chaque arc est associé à une contrainte souple dont le coût de chaque paire de valeurs possibles est tiré aléatoirement et uniformément dans $U[0-9]$.

Le modèle d'Ising est une référence largement utilisée en physique statistique ; nous utilisons ici les mêmes paramètres dans la littérature [38]. Nous générons une grille toroïdale régulière et associons chaque nœud à une variable binaire x_i et chaque arc à une contrainte binaire dont la fonction de coût r_{ij} est déterminée en échantillonnant d'abord une valeur k_{ij} à partir d'une distribution uniforme $U[-1.6, 1.6]$, et ensuite assignons $r_{ij}(x_i, x_j) = -k_{ij}$ si $x_i = x_j$, k_{ij} sinon. De plus, chaque variable a une contrainte unaire dont la fonction de coût r_i est déterminée en échantillonnant k_i à partir d'une distribution uniforme $U[-0.05, 0.05]$ puis en affectant $r_i(0) = k_i$ et $r_i(1) = -k_i$.

Pour les 3 types de DCOP, nous dérivons deux graphes de calculs pour les deux méthodes de résolutions utilisées : sous la forme d'un graphe de facteurs (FG, pour A-MaxSum) et sous la forme d'un graphe de contraintes (CG, pour A-DSA). Pour un même problème, $\langle \mathcal{A}, \mathcal{X}, \mathcal{D}, C, \mu \rangle$, il faut placer soit $|\mathcal{X}| = |\mathcal{X}| + |C|$ calculs pour un FG ou $|\mathcal{X}| = |\mathcal{X}|$ calculs pour un CG.

Une fois les graphes de calcul dérivés, nous générons deux infrastructures multi-agents différentes : une infrastructure uniforme et une infrastructure dont les caractéristiques dépendent de la structure du problème, similaire à ce qu'on observe dans les

infrastructures physiques des systèmes IoT. Une infrastructure est composée d'agents $|\mathcal{A}|$, chacun contenant une variable de décision ($|\mathcal{A}| = |\mathcal{X}|$), et est définie par $\mathbf{c_{host}}$, **route**, $\mathbf{w_{max}}$, **w** et **msg**.

L'infrastructure uniforme considère les systèmes où les coûts de communication sont uniformes : $\forall a_m, a_n, \mathbf{route}(a_m, a_n) = 1$.

Dans le cas de l'infrastructure de type IoT, nous considérons que les coûts de communication dépendent de la structure du problème : les coûts $\mathbf{route}(a_m, a_n)$ sont définis de manière à respecter la structure du graphe de calcul : les agents ayant plusieurs voisins ont une faible communication tandis que les agents ayant peu de voisins ont un coût de communication plus élevé. Plus précisément, $\mathbf{route}(a_m, a_n) = \frac{1+||N(a_m)|-|N(a_n)||}{|N(a_m)|+|N(a_n)|}$ où $|N(a_i)|$ est le nombre de voisins de a_i dans le graphe de calculs.

Pour les deux types d'infrastructure, nous définissons (i) $\mathbf{c_{host}}(a_m, x_j) = 0$ si le calcul x_j est initialement hébergé par l'agent a_m, $\mathbf{c_{host}}(a_m, x_j) = 10$ sinon ; (ii) la capacité de chaque agent dépend du poids de sa variable de décision et est fixé à une valeur suffisament grande, de manière à ce que tous les réplicas puissent être hébergés et que la k-résilience soit possible, même après plusieurs réparations : $\mathbf{w_{max}}(a_i) = 100 * \mathbf{w}(x_i)$; (iii) finalement, **w** et **msg** dépendent du processus utilisé. Pour Max-Sum, la taille des messages est une fonction directe de la taille de domaine de la variable : $\mathbf{msg}(i, j) = |D_j| = 10$ et le poids des calculs de variables et de facteurs est respectivement proportionnel à la taille du domaine de la variable et à la somme de la taille des domaines des variables liées. Pour A-DSA, $\mathbf{msg}(i, j) = 1$ et $\mathbf{w}(x_i) = |N(a_m)|$.

De part la définition du système, chaque variable de décision est initialement affectée à un agent. Dans le cas des FG, les facteurs sont placés de manière optimale à l'aide du programme linéaire décrit dans la section 3. Nous utilisons $\omega = 0.5$ (les coûts d'hébergement et de communication sont pris en compte de manière égale).

Les problèmes ainsi générés sont résolus en utilisant les algorithmes DCOP A-DSA et A-MaxSum, qui sont particulièrement adaptés à ce type de situation de par leur capacité à continuer leur exécution en cas de perte de messages. Nous utilisons par ailleurs une implémentation de A-MaxSum modifiée pour améliorer son comportement suite à une réparation : une fois un calcul orphelin migré sur un agent, les tables de coûts de ses voisins dans le graphe de calculs sont automatiquement vidées et le mécanisme de propagation de croyances est relancé. Nous désactivons aussi à ce moment le mécanisme habituellement utilisé pour éviter l'envoi de messages en doublons (classiquement utilisé pour détecter la convergence [8]). Ce comportement peut aisément être implémenté de manière distribuée en utilisant une approche de passage de jeton.

Nous générons aussi des scénarios de perturbation sous forme de séquences d'événements toutes les 30 secondes, à partir de $t = 20s$. Àchaque événement, k agents choisis au hasard disparaissent et le système se répare de manière automatique en utilisant DRPM[MGM-2], qui utilise, comme décrit dans la section 5.2, un DCOP résolu avec MGM-2, pour rétablir un fonctionnement nominal du DCOP original (coloriage ou ising) résolu avec A-DSA ou A-MaxSum. Nous analysons ainsi l'impact de ce mécanisme de réparation sur les processus DCOP, et validons la k-résilience de notre système.

Nous fixons $k = 3$, et le déploiement des réplicas initiaux avec DRPM dure moins d'une seconde en moyenne sur les tailles d'instances considérées, et même moins lorsque seuls les réplicas manquants sont replacés. Ce processus peut nécessiter jusqu'à 4Mo de données échangées sur les plus grandes instances lors du placement initial.

Nous générons 20 instances (infrastructure et problème), et exécutons le scénario 5 fois pour chaque instance. De plus, nous résolvons les mêmes problèmes (5 exécutions pour chacune des 20 instances) sans aucune perturbation, afin d'évaluer l'impact de nos méthodes de réparation sur la qualité de la solution renvoyée par A-DSA ou A-MaxSum. Dans les deux cas, les résultats sont moyennés sur toutes les instances.

Nous utilisons la librairie open-source python pyDCOP[(1)] [33] réaliser l'ensemble de ces expériences : la génération des instances de problème, des graphes de calcul et des agents, ainsi pour le calcul des distributions, la résolution des instances et le processus d'auto-réparation du système. Pour résoudre les trois types de DCOP, nous utilisons les algorithmes A-DSA (variant B, $p = 0.7$) et A-MaxSum (avec un facteur d'amortissement de 0.8 pour les variables et les facteurs). L'algorithme utilisé pour la réparation de la distribution est MGM-2 ($q = 0.5$). Ces paramétrages sont le résultat d'expérimentations préliminaires non exposées ici, qui ont permis d'identifier les meilleures configurations de ces algorithmes.

7.1.2. *Impact de la réparation sur A-DSA*

Regardons l'état d'exécution d'A-DSA avec et sans perturbations, et analysons comment DRPM[MGM-2] modifie le fonctionnement du processus en cours d'exécution. Nous générons des problèmes avec $|\mathcal{A}| = |\mathcal{X}| = 100$ et utilisons $k = 3$. La figure 7.1 montre le coût de la solution trouvée par A-DSA au fil du temps. Le coût de chacune des 100 exécutions est affiché en gris transparent, et la forme globale illustre le fait que le comportement du système est cohérent dans les différentes instances. Nous pouvons voir que les solutions avec perturbations se dégradent lorsque les agents sont supprimés, mais rapidement s'améliorent rapidement après la réparation, lorsque le système récupère.

Ici, de part le fonctionnement de A-DSA, les réplicas activés par la réparation n'ont pas besoin de connaissances accumulées pour retrouver un état cohérent. En effet, les calculs sont ici `stateless`, comme l'exige notre approche de la k-résilience : grâce aux messages passés entre voisins, ils collectent de nouvelles informations sur les coûts auprès de leurs voisins à chaque échange de messages.

La période de récupération est plus courte sur les modèles *scale free* et Ising. En effet, les problèmes de coloration des graphes aléatoires sont plus denses et plus difficiles à résoudre, et leur réparation nécessite plus de messages et de temps. Pendant la même durée, les agents qui résolvent et réparent les processus de coloration des graphes doivent gérer davantage de messages spécifiques à la réparation (MGM-2 et DRPM). Ils gèrent donc moins de messages A-DSA pour améliorer le coût de la solution. Le même phénomène explique le coût global est plus élevé dans une infrastructure dépendant du problème : les processus d'activation et de placement de réplicas nécessitent plus de messages.

[(1)] `https://github.com/Orange-OpenSource/pyDcop`

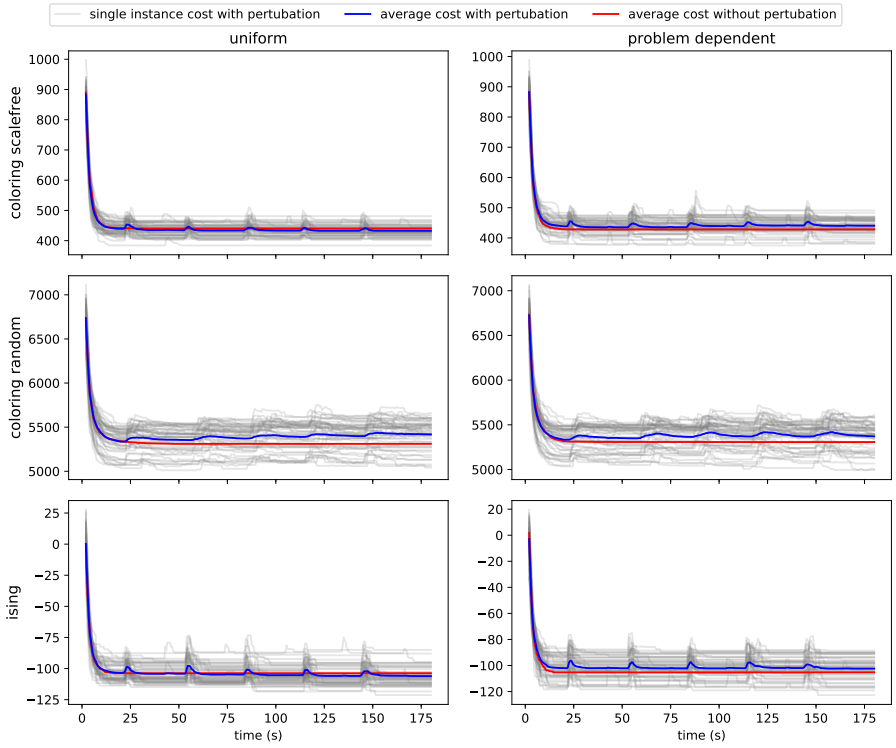


FIGURE 7.1 – Coût des solutions A-DSA en cours d'exécution, avec (bleu) et sans perturbations (rouge), sur des infrastructures uniformes (gauche) et dépendantes du problème (droite), et sur coloration *scale free* (haut), coloration aléatoire (milieu), et Ising (bas).

Nous avons évalué le temps moyen nécessaire pour réparer une distribution à moins de 3 secondes. Dans certains contextes, le coût moyen avec perturbation est inférieur au coût moyen sans perturbation. Il s'avère que la suppression de certains calculs et leur transfert à d'autres agents, avec la perte des informations sur le voisinage passé occasionnée, peut permettre à A-DSA de s'extraire de certains optima locaux. Nous pouvons également observer dans certains cas un décalage progressif dans le temps de la réparation du système. Cela est principalement dû à la latence de traitement des messages accumulés.

7.1.3. *Impact de la réparation sur A-MaxSum*

Ici, nous considérons des problèmes plus petits, car A-MaxSum fonctionne sur des graphes de facteurs nécessitant plus de calculs (un de plus par arête dans le graphe) à distribuer que les graphes de contraintes utilisés par A-DSA. Par conséquent, nous considérons $|\mathcal{A}| = |\mathcal{X}| = 25$, et $k = 2$. Néanmoins, sur un graphe aléatoire avec une densité de 0.3, de tels problèmes nécessitent en moyenne $25 + 0.3\frac{25\times24}{2} = 125$ calculs à gérer.

Dans la figure 7.2 nous pouvons voir que là aussi les solutions sur le système perturbé se dégradent lorsque les agents sont supprimés, mais s'améliorent à nouveau lorsque le système récupère, comme pour A-DSA. Cependant, le fonctionnement A-MaxSum sur des problèmes très cycliques tels que la coloration aléatoire est connu pour être très bruité, même en utilisant un facteur d'amortissement élevé (nous utilisons ici 0.8), comme proposé par les auteurs [4]. En outre, dans les algorithmes de propagation de croyances comme A-MaxSum, les calculs ne sont pas vraiment sans état : ils accumulent des informations sur les contraintes et les préférences de leurs voisins. Lors de l'activation d'un réplica, le nouveau calcul actif recommence à nouveau et un nombre indéterminé de tours de messages sont nécessaires pour restaurer ces informations. Globalement, le fonctionnement A-MaxSum est davantage impacté par la procédure de perturbations et de réparation qu'A-DSA mais parvient dépendant à maintenir, malgré les perturbation, une solution de qualité assez proche de celle atteinte en régime nominal.

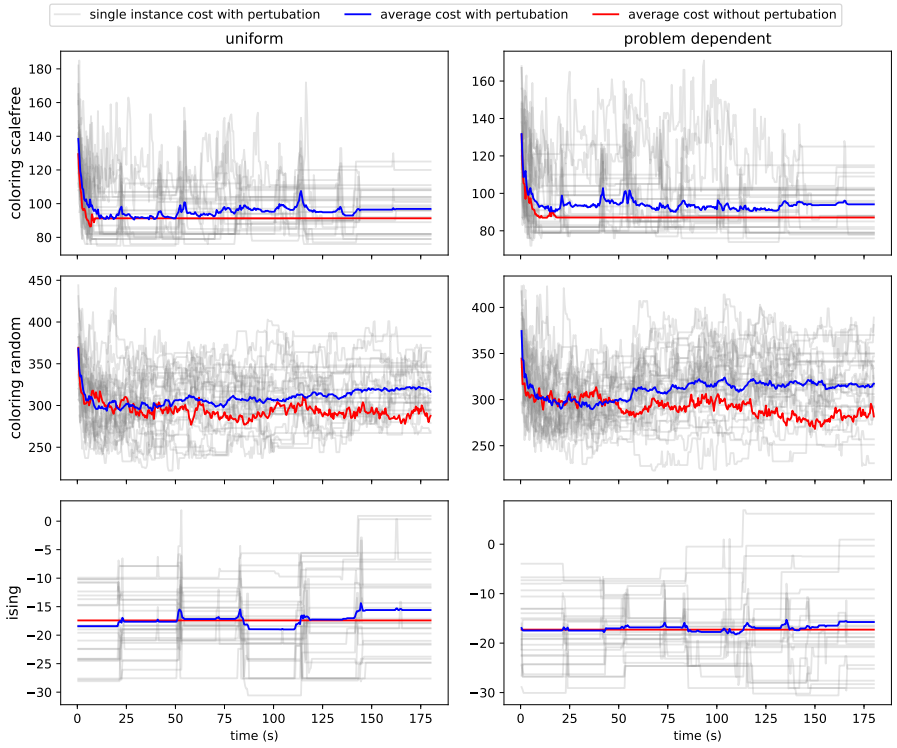


FIGURE 7.2 – Coût des solutions de A-MaxSum en cours d'exécution, avec (bleu) et sans perturbations (rouge), sur des infrastructures uniformes (gauche) et dépendantes du problème (droite), et sur coloration *scale free* (haut), coloration aléatoire (milieu), et Ising (bas).

7.1.4. *Qualité des distributions réparées*

Pour évaluer la qualité des distributions de calculs réparées, nous mesurons la dégradation de la distribution tout au long de la vie du système. Àchaque événement, nous évaluons le coût de la distribution actuelle des graphes de contraintes (pour A-DSA) et des graphes de facteurs (pour Max-Sum) à l'aide des équations 3.4 et 3.5, par rapport au coût de distribution initial (qui est optimal, mais ne peut pas être calculé au moment de l'exécution). Les figures 7.3 et 7.4 montrent les coûts de distribution pour les 100 instances. Comme le coût de distribution global est constitué des coûts de communication et d'hébergement, nous traçons également ces deux coûts indépendamment. Dans tous les cas, le coût de l'hébergement augmente logiquement de $10 \cdot k$ à chaque événement de perturbation, les k calculs étant déplacés de leur agent initial à un autre (où le coût d'hébergement est de 10). Sur les modèles *scale free* et Ising, les coûts d'hébergement et de communication ont le même ordre de grandeur. Mais, pour les graphes aléatoires, une densité plus élevée implique qu'il y a plus d'arcs dans le graphique et, par conséquent,

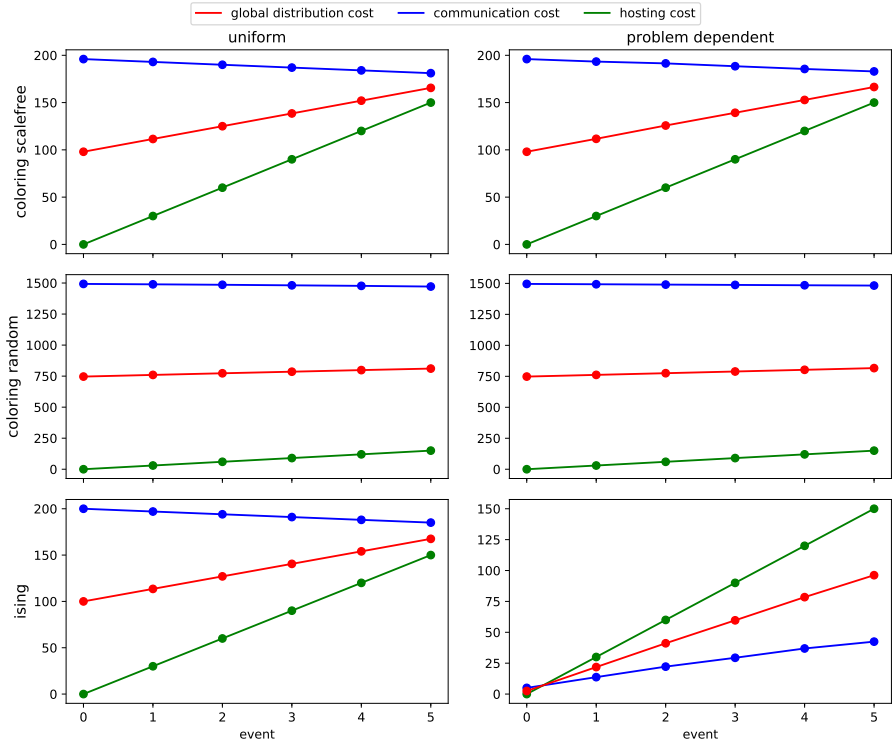


FIGURE 7.3 – Coût de la distribution des graphes de calculs pour A-DSA, après chaque événement, sur des infrastructures uniformes (gauche) et dépendantes du problème (droite), et sur coloration *scale free* (haut), coloration aléatoire (milieu), et Ising (bas).

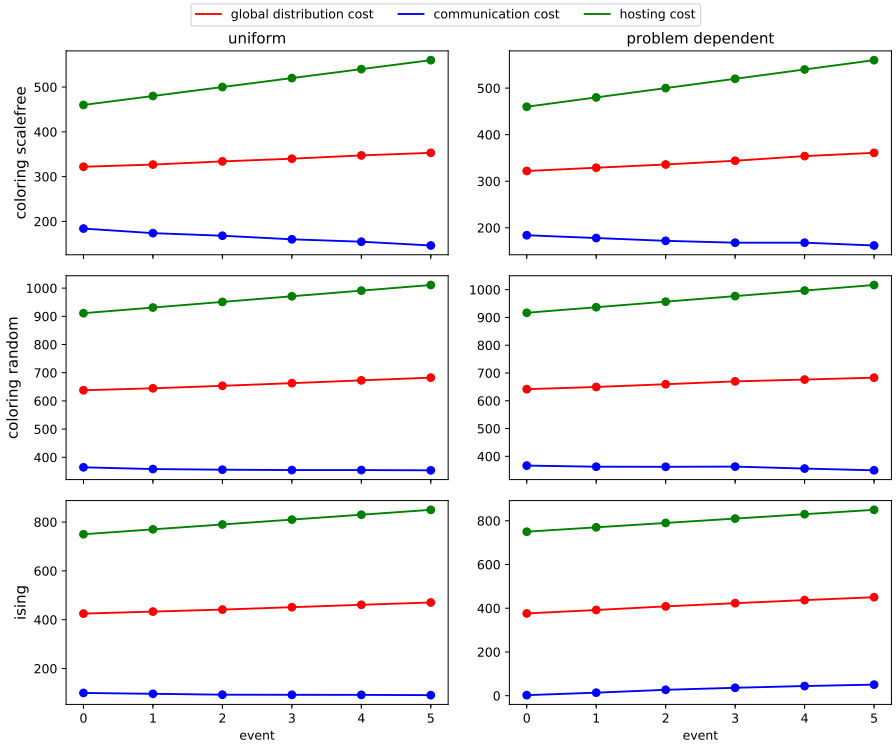


FIGURE 7.4 – Coût de la distribution des graphes de calculs pour A-MaxSum, après chaque événement, sur des infrastructures uniformes (gauche) et dépendantes du problème (droite), et sur coloration *scale free* (haut), coloration aléatoire (milieu), et Ising (bas).

le coût de communication global est plus élevé. En général, les coûts de communication diminuent à chaque réparation, sauf dans Ising avec une infrastructure dépendant du problème. Ici, tous les agents sont homogènes et les calculs sont nécessairement transférés vers des agents plus coûteux, en termes de communication (il n'existe aucun agent moins coûteux ni équivalent aux agents manquants) et les coûts de communication augmentent donc progressivement. Dans les autres cas, les calculs peuvent passer à un agent moins coûteux, ce qui entraîne une diminution des coûts de communication.

7.2. ANALYSE SUR DES SECP

Nous évaluons maintenant de manière expérimentale notre solution de réparation DRPM[MGM-2] sur des instances de SECP générées de manière aléatoire.

7.2.1. *Cadre expérimental*

Nous considérons ici un environnement réaliste d'une maison équipée d'un grand nombre d'effecteurs (ampoules connectées), chacun représenté par une variable x_i pouvant avoir une valeur dans $[0..4]$, et associé à une fonction de coût linéaire dépendant de la luminosité émise.

Chaque modèle de dépendance physique est représenté par un couple (φ_j, y_j) où φ_j est une fonction qui retourne le niveau de luminosité théorique d'une zone dans la maison et est définit comme une somme pondérée (les poids sont tirés aléatoirement) des luminosités émises par un ensemble d'effecteurs.

Les règles r_k fixent des valeurs cibles à une ou plusieurs variables d'action (effecteur x_i ou modèle y_j) ; Les effecteurs, modèles et règles sont connectés de manière aléatoire et nous ne considérons que les règles actives (i.e. qui ont une influence réelle sur le problème). Nous utilisons $\omega_c = 1$ et $\omega_u = 10$ comme poids pour agréger les deux objectifs sur l'utilité des règles et l'économie d'énergie (cf. objectif 6.6).

Les coûts de communication sont uniformes (1) entre les agents et les coût d'hébergement sont identiques (100) pour tous les calculs exceptés ceux correspondant aux effecteurs (0), hébergés directement par les effecteurs eux-même. La capacité des agents est fixée à 1000 pour toutes les instances possédant moins de 60 effecteurs, et à 1500 au-delà.

Nous générons 100 instances de SECP avec 30 effecteurs, 9 modèles de dépendances physique et 6 règles. La distribution initiales des SECP est générée en utilisant l'heuristique GH-CGDP. Pour chaque instance, un scénario de perturbation de 5 évènements espacés de 30 secondes, est généré, où deux effecteurs tirés aléatoirement sont retirés à chaque évènements

Comme dans l'expérimentation précédente, ces 100 instances sont résolues en utilisant A-DSA et la version modifiée de A-MaxSum décrite à la section 7.1.1. Durant le processus de résolution, les perturbations sont injectées dans le système et la réparation automatique est réalisée avec DMCM[MGM-2]. Nous exécutions aussi le même processus sans perturbation, afin de pouvoir évaluer l'impact du processus de réparation sur la qualité des résultats produits lors de la résolution du SECP.

Nous utilisons ici aussi pyDCOP [33] pour la génération des instances de problème et le processus d'auto-réparation du système. Pour résoudre les instances SECP, nous utilisons également les algorithmes A-DSA et A-MaxSum. L'algorithme utilisé pour la réparation de la distribution est MGM-2 ($q = 0.5$, fixé après paramétrage expérimental). L'implémentation de tous les générateurs et algorithmes utilisés est disponible dans la librairie pyDCOP.

7.2.2. *Impact de la réparation des SECP résolus avec A-DSA*

La figure 7.5 représente la qualité des solutions aux instances SECP produites par A-DSA, avec et sans perturbations. Comme la formulation du problème SECP contient à la fois des contraintes souples et dures, nous représentons à la fois la somme des coûts des contraintes souples et le nombre de contraintes dures violées. Comme précédemment, le résultat des différentes instances est représenté en gris transparent et la moyenne, sur

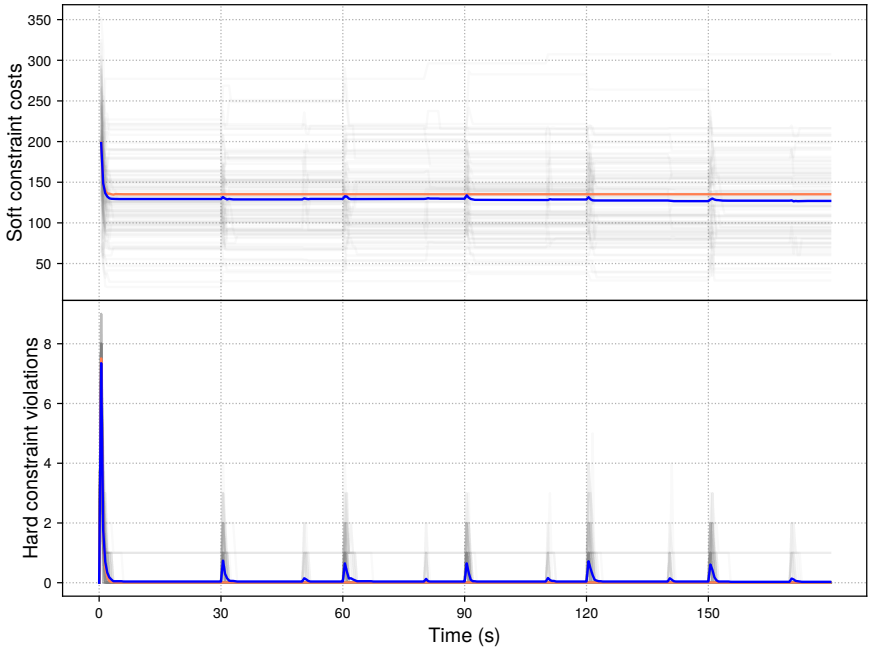


FIGURE 7.5 – Coût et violation de contraintes dures lors de la résolution des SECP avec A-DSA, réparé par DRPM[MGM-2] (bleu : avec perturbations, rouge : sans perturbation)

l'ensemble des instances, des coûts et du nombre de violations est représenté en bleu pour les exécutions avec perturbations et en rouge pour les exécutions sans perturbation.

Le comportement après une réparation automatique du système est remarquable : la disparition des agents entraine la violation de contraintes dures, mais suite à la réparation ces dernières sont très rapidement à nouveau satisfaites, dès que le système reprend la résolution du SECP avec A-DSA. La qualité globale des solutions, en terme de coût des contraintes souples, est aussi très bonne après réparation, dépassant même en moyenne celle des exécutions sans perturbation (là aussi, ce sont les perturbations qui permettent à A-DSA de s'extraire de minima locaux).

7.2.3. *Impact de la réparation des SECP résolus avec A-MaxSum*

La figure 7.6 représente de la même manière les coûts et le nombre de contraintes violées durant l'exécution des SECP résolus avec A-MaxSum, avec et sans perturbation.

Là aussi le comportement est très satisfaisant : La courbe des coûts des contraintes souples après réparation est tellement proche de celle obtenue sans perturbation qu'on ne peut pratiquement pas les distinguer sur la figure. On remarque aussi que A-MaxSum produit d'une manière générale des solutions d'une meilleure qualité que A-DSA. La

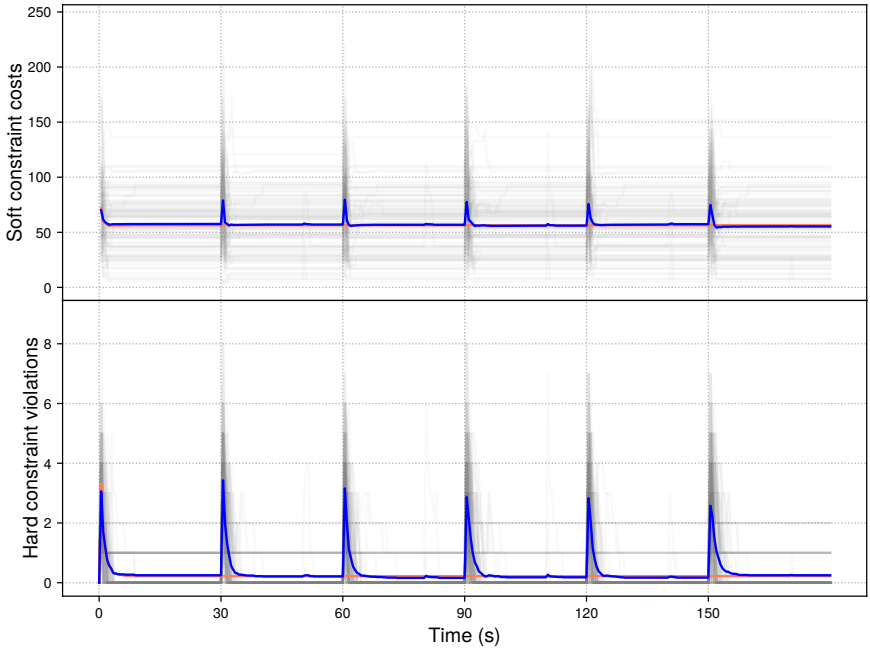


FIGURE 7.6 – Coût et violation de contraintes dures lors de la résolution des SECP avec A-MaxSum, réparé par DRPM[MGM-2] (bleu : avec perturbations, rouge : sans perturbation)

disparition des agents occasionne la violation d'un nombre plus important de contraintes dures qu'avec A-DSA, mais revient ensuite à la situation observée avant la perturbation, ce qui nécessite cependant un peu plus de temps que pour A-DSA. On remarque par ailleurs que la moyenne du nombre de violations n'est pas nulle, mêmes dans les cas sans perturbation ; en effet sur certaines instances A-MaxSum a du mal à trouver une solution satisfaisante.

D'une manière générale, nous pouvons dire que DRPM[MGM-2] est très bien adapté à la réparation des systèmes SECP, dont le fonctionnement et la qualité des résultats n'est pratiquement pas impacté par les opérations de réparation effectuées lors de la disparition des agents. A-MaxSum et A-DSA continuent de fournir, malgré ces perturbations, des résultats de qualité, démontrant bien la résilience de ces systèmes. La représentation sous forme de graphe de calcul (avec A-DSA) est plus rapide que celle utilisant un graphe de facteur, en effet la complexité du processus de réparation dépend fortement du nombre de calcul et de la densité du système. De plus, A-MaxSum est plus impacté par la disparition des agents : les augmentations des coûts et les violations de contraintes dures suite à une perturbation sont plus importantes qu'avec A-DSA. En effet, malgré nos adaptations pour accélérer la reconstruction de l'état, ses calculs ne sont pas complètement *stateless*, contrairement à A-DSA, plus robuste à ce type de modifications.

8. Discussion et travaux connexes

Nous présentons dans cette section une brève revue de certains travaux connexes axés sur le traitement distribué des contraintes pour l'intelligence ambiante basée sur l'Internet des Objets, la distribution des décisions et des calculs, et la résilience dans des contextes dynamiques.

8.1. Raisonnement sous contraintes en Intelligence Ambiante

Dans [6], Degeler *et al.* utilisent le raisonnement par contraintes dynamiques pour la gestion intelligente de l'environnement. Le comportement souhaité de la maison est spécifié à l'aide de règles logiques. Le problème est ensuite encodé comme un CSP dynamique, où les actionneurs sont représentés comme des *variables contrôlables*, afin de prendre en compte les changements du contexte environnemental. Kaldeli *et al.* soutiennent dans [15] que le comportement intelligent dans un environnement de maison intelligente nécessite des fonctionnalités complexes qui impliquent plusieurs services sélectionnés dynamiquement et fournis par des dispositifs indépendants. Ils proposent donc de combiner la conception d'une architecture orientée services avec la composition automatique et dynamique de services. La composition de services est implémentée en utilisant un planificateur CSP indépendant du domaine. Le planificateur raisonne sur ce modèle et génère une séquence d'actions qui doivent être exécutées. Song *et al.* conçoivent un système auto-adaptatif qui prend en compte les préférences de l'utilisateur et l'appliquent à un scénario de maison intelligente [36]. L'ensemble des politiques d'adaptation est modélisé comme un CSP, qui permet de détecter les conflits d'objectifs et de trouver la meilleure configuration qui satisfait le plus grand nombre d'objectifs possible. Parra *et al.* utilisent un CSP pour modéliser une sélection dynamique de fonctionnalités dans une ligne de produits logiciels et optimiser ce processus de configuration [25]. Ce travail est appliqué à un scénario de maison intelligente où une application doit s'auto-adapter au type d'appareils et aux options de connectivité actuellement disponibles dans la maison.

Cependant, les quatre approches susmentionnées reposent sur un nœud de décision central. Certaines recherches ont été menées sur l'application du cadre DCOP à des cadres qui peuvent être considérés comme faisant partie de l'Internet des objets et de l'informatique ubiquitaire : réseaux de capteurs, allocation de fréquences radio, coordination des feux de circulation, etc. Cependant, à notre connaissance, peu de travaux antérieurs ont porté sur l'utilisation du cadre DCOP pour l'AmI ou la maison intelligente. Dans les travaux de Pecora *et al.*, l'intégration de services complexes pour l'AmI est représentée par la coordination multi-agents, puis abordée à l'aide d'un DCOP [26]. Dans leur modèle, chaque application du foyer offre un ou plusieurs services et est représentée par un agent. Le DCOP résultant est continuellement résolu par les agents et donne une solution où les variables de sortie correspondent au comportement souhaité de la maison. Leur mise en œuvre est basée sur ADOPT-N, un solveur complet. Fioretto *et al.* proposent d'utiliser une approche DCOP pour la gestion de la demande dans un réseau électrique intelligent, en se basant sur le problème de la programmation des appareils domestiques intelligents (SHDS) [9]. Ce problème est modélisé comme un problème d'ordonnancement distribué, qui est traduit en un DCOP qui inclut à la fois

des contraintes souples, pour les préférences des utilisateurs et la consommation d'énergie, et des contraintes dures, pour les objectifs temporels. Leur implémentation utilise SH-MGM, un algorithme personnalisé basé sur MGM. En complément du problème SHDS, Kluegel *et al.* propose un ensemble de modèles physiques pour les appareils de la maison intelligente et un ensemble de données d'instances de problèmes qui peuvent être utilisées pour évaluer les méthodes de résolution [17].

8.2. Distribution de calculs et de décisions

Nous avons fait valoir que l'hypothèse couramment utilisée sur la distribution des DCOP ne tient pas lorsqu'on travaille sur des problèmes du monde réel et qu'il est nécessaire de considérer la question de la distribution des décisions sur un ensemble d'agents. Bien que peu de travaux existent actuellement dans ce domaine, nous considérons qu'il s'agit d'une partie importante de l'utilisation des approches DCOP dans les environnements physiques distribués. Lors de la conception de cette distribution, nous pouvons prendre en considération de nombreux éléments. En effet, le placement des calculs sur les agents peut avoir un impact important sur les caractéristiques de performance du système global. Certaines distributions peuvent améliorer les temps de réponse, d'autres peuvent favoriser la charge de communication entre les agents et d'autres encore peuvent être meilleures pour d'autres critères comme la QoS ou le coût de fonctionnement. Bien que la distribution soit rarement étudiée dans la communauté DCOP, quelques travaux récents ont commencé à s'y intéresser. Nous avons d'abord proposé une méthode de distribution dédiée aux graphes factoriels [32], qui vise à minimiser les coûts de communication et les coûts d'hébergement, comme les méthodes présentées dans la section 3, qui sont plus génériques et peuvent être appliquées à tout modèle graphique et solveur DCOP. Par ailleurs, dans [16], Khan *et al.* analysent le placement des nœuds de graphes de contraintes sur les agents du point de vue de la performance ; leur objectif est de trouver un placement qui minimise le temps d'achèvement de la résolution du DCOP. Comme nous, ils soutiennent que, dans de nombreux problèmes, il existe plusieurs affectations possibles des nœuds aux agents. Cependant, ils ne considèrent que les nœuds variables et ne définissent pas un concept plus général de distribution, qui prend en compte d'autres types de nœuds (facteurs, plusieurs variables, etc.).

8.3. Résilience de processus de décision dans des contextes dynamiques

Une extension du cadre DCOP, à savoir les DCOPs dynamiques, traite des problèmes dont la définition change pendant l'exécution, comme c'est le cas dans le modèle SECP. L'approche classique, décrite comme *réactive* [18, 28, 29, 42], est directement basée sur le modèle d'une séquence de DCOPs statiques, où les futurs DCOPs sont entièrement inconnus. Chaque DCOP statique est simplement résolu de manière séquentielle ; chaque fois que le problème change, le nouveau DCOP est résolu et la solution précédente est remplacée. L'avantage de cette approche est qu'elle peut théoriquement être utilisée avec n'importe quel algorithme DCOP. L'inconvénient de cette approche est qu'elle n'est applicable que si le taux de changement est suffisamment lent, comparé au temps nécessaire pour résoudre l'un des DCOPs de la séquence, pour terminer la résolution du

problème avant qu'un nouveau changement ne se produise. Sinon, le système continuerait à résoudre des problèmes périmés et pourrait même ne jamais produire de solution, car il recommencerait à résoudre un nouveau problème alors qu'aucune solution n'a encore été trouvée pour le précédent. Pour éviter ce problème, les chercheurs ont proposé des algorithmes qui réutilisent les informations des DCOP précédents pour accélérer la recherche du problème actuel. Une variante de cette approche, également *réactive*, consiste à utiliser des algorithmes DCOPs qui peuvent s'adapter dynamiquement aux changements et continuer à travailler sur le problème mis à jour sans repartir de zéro. Certains travaux considèrent également les coûts de passage d'une solution à une autre et prennent en compte ces coûts lors de la sélection d'une solution pour le DCOP suivant dans la séquence. L'objectif ici est la *stabilité de la solution*, qui prend en compte le changement de solution dans ces systèmes dynamiques et tente de minimiser ses effets.

Une autre approche, appelée *proactive* [13, 14, 23], consiste à considérer que les futurs DCOPs de la séquence sont connus à l'avance ou que les changements potentiels futurs peuvent être au moins partiellement anticipés. Dans ce cas, l'objectif est de rechercher des solutions qui sont robustes à ces changements, c'est-à-dire des solutions qui ne nécessitent pas ou peu de changements malgré la modification du problème. Malheureusement, les méthodes de résolution actuelles pour ce modèle sont soit hors ligne, soit trop coûteuses pour être utilisées avec autre chose qu'un nombre très limité d'agents. Dans notre cas, nous faisons valoir que nous ne pouvons pas prédire les changements futurs du système et nous considérons que nous n'avons pas vraiment besoin de la robustesse de la solution que vise l'approche auto-stabilisante. Cette caractéristique est intéressante lorsque le passage d'une solution à une autre induit un coût important pour le système, comme cela peut être le cas pour le routage des véhicules ou la planification des réunions. Dans ces cas, la stabilité de la solution est en effet requise, et il est primordial de prendre en compte le coût de la transition vers une nouvelle solution par rapport au bénéfice apporté par cette nouvelle solution. Par conséquent, il est potentiellement plus intéressant de troquer une certaine optimalité contre une plus petite quantité d'adaptation lors des changements.

Enfin, la notion de k-résilience, où k réplicas sont placées sur différents agents, est inspirée des techniques des systèmes de bases de données distribuées [24], sauf qu'ici nous sauvegardons les définitions des calculs au lieu de données.

9. Conclusions

Nous avons proposé un processus complet pour mettre en œuvre l'auto-configuration de systèmes multi-agents physiques. Nous avons modélisé des scénarios d'environnement intelligent comme des DCOPs, et fourni plusieurs méthodes pour les distribuer et les résoudre dans une configuration dynamique. Nous avons mis en place une prise de décision décentralisée et une coordination des objets et services intelligents en exploitant des algorithmes DCOP. Pour doter ce processus de décision d'une résilience face à la dynamique de l'environnement et de l'infrastructure, deux algorithmes distribués ont été conçus : (i) un protocole de placement de réplicats (DRPM) et (ii) un protocole de réparation (DRPM[DMCM]) s'appuyant sur des réplicas placées par DRPM et se basant lui-même sur la méthode de solution MGM-2. Cette réparation, suivie d'une

phase de placement des réplicas manquant, assure ainsi la k-résilience du système. Ces deux protocoles ne nécessite qu'une connaissance partiel du système de la part des agents (leur voisinage de communication et les agents hébergeant les réplicas de mêmes calculs). Nous fournissons ainsi une auto-adaptation des solveurs DCOP en utilisant un autre processus de réparation basé sur un DCOP.

Nos contributions ont été évaluées expérimentalement en faisant fonctionner A-MaxSum et A-DSA sur des systèmes dynamiques où les agents disparaissent pendant le processus d'optimisation. Nous avons étudié différentes métriques, à la fois sur des benchmarks classiques et des scénarios SECP synthétiques. Notre méthode de réparation de réparation DRPM[MGM-2] n'a pas d'impact sur le processus de résolution du DCOP, même si certains agents disparaissent, ce qui démontre la résilience de ces systèmes. Dans nos expériences, le fonctionnement de A-MaxSum est beaucoup plus affecté, en terme de dégradation de qualité de solution, par le retrait d'agent et la réparation que l'algorithme A-DSA, qui est très robuste à de telles dynamiques.

Cet article a soulevé des résultats prometteurs sur la résilience dans le fonctionnement des processus d'optimisation distribuée. Nous nous sommes uniquement concentrés sur un scénario avec retrait d'agents uniquement. Nous allons étudier des scénarios moins stressants, provenant d'un champ plus large de calculs basés sur des graphes, comme le calcul haute performance ou les fonctions de réseau virtuel, où des agents peuvent être ajoutés pour remplacer ceux qui ont disparu. Le cas de graphes déconnectés sera également envisagé. De plus, nous avons proposé d'utiliser MGM-2 comme algorithme de réparation de base, ce qui donne la méthode DRPM[MGM-2], mais d'autres méthodes DCOPs légères pourraient être envisagées ou même conçues pour le cas particulier de la réparation de graphes de contraintes ou de graphes de facteurs. Enfin, les approches visant à préserver la divulgation d'informations tout en assurant la résilience du système, et le compromis qui en résulte entre résilience et vie privée feront l'objet de recherches futures.

Bibliographie

[1] A. Barabasi & R. Albert, « Emergence of Scaling in Random Networks », *Science* **286** (1999-10-15), n° 5439, p. 509-512.

[2] F. Bonomi, R. Milito, J. Zhu & S. Addepalli, « Fog Computing and Its Role in the Internet of Things », in *Proceedings of the First Edition of the MCC Workshop on Mobile Cloud Computing* (New York, NY, USA), MCC '12, ACM, 2012, p. 13-16.

[3] F. Chakchouk, S. Piechowiak, R. Mandiau, J. Vion, M. Soui & K. Ghedira, « Fault Tolerance in DisCSPs: Several Failures Case », in *Distributed Computing and Artificial Intelligence, 15th International Conference* (F. De La Prieta, S. Omatu & A. FernÃ¡ndez-Caballero, éds.), vol. 800, Springer International Publishing, 2019, p. 204-212.

[4] L. Cohen & R. Zivan, « Max-sum Revisited: The Real Power of Damping », in *Autonomous Agents and Multiagent Systems* (Cham) (G. Sukthankar & J. A. Rodriguez-Aguilar, éds.), Springer International Publishing, 2017, p. 111-124.

[5] R. Dechter, *Constraint Processing*, Morgan Kaufmann, 2003.

[6] V. Degeler & A. Lazovik, « Dynamic Constraint Reasoning in Smart Environments », in *2013 IEEE 25th International Conference on Tools with Artificial Intelligence*, 2013-11, p. 167-174.

[7] A. Farinelli, A. Rogers, A. Petcu & N. R. Jennings, « Decentralised Coordination of Low-power Embedded Devices Using the Max-sum Algorithm », in *International Conference on Autonomous Agents and Multiagent Systems (AAMAS'08)*, 2008, p. 639-646.

[8] ———, « Decentralised Coordination of Low-Power Embedded Devices Using the Max-Sum Algorithm », in *Proceedings of the International Conference on Autonomous Agents and Multiagent Systems*, 2008, p. 639-646.

[9] F. Fioretto, E. Pontelli & W. Yeoh, « A Multiagent System Approach to Scheduling Devices in Smart Homes », in *Proceedings of the International Workshop on Artificial Intelligence for Smart Grids and Smart Buildings*, 2017, p. 7.

[10] ———, « Distributed Constraint Optimization Problems and Applications: A Survey », *Journal of Artificial Intelligence Research* **61** (2018-03-29), p. 623-698, `https://arxiv.org/abs/1602.06347`.

[11] S. Fitzpatrick & L. Meertens, « Distributed Coordination through Anarchic Optimization », in *Distributed Sensor Networks*, Springer, 2003, p. 257-295.

[12] Z. Guessoum, J.-P. Briot & N. Faci, « Un mécanisme de réplication adaptative pour des SMA tolérants aux pannes », in *JFSMA*, 2004, p. 135-148.

[13] K. D. Hoang, F. Fioretto, P. Hou, M. Yokoo, W. Yeoh & R. Zivan, « Proactive Dynamic Distributed Constraint Optimization », in *Proceedings of the 2016 International Conference on Autonomous Agents & Multiagent Systems*, 2016, p. 597-605.

[14] K. D. Hoang, P. Hou, F. Fioretto, W. Yeoh, R. Zivan & M. Yokoo, « Infinite-Horizon Proactive Dynamic DCOPs », in *Proceedings of the 16th Conference on Autonomous Agents and MultiAgent Systems*, AAMAS '17, International Foundation for Autonomous Agents and Multiagent Systems, 2017, p. 212-220.

[15] E. Kaldeli, E. U. Warriach, A. Lazovik & M. Aiello, « Coordinating the Web of Services for a Smart Home », *ACM Transactions on the Web* **7** (2013-05-01), nº 2, p. 1-40.

[16] M. Khan, L. Tran-Thanh, W. Yeoh & N. R. Jennings, « A Near-Optimal Node-to-Agent Mapping Heuristic for GDL-Based DCOP Algorithms in Multi-Agent Systems », in *Roceedings of the International Conference on Autonomous Agents and Multiagent Systems*, 2018, p. 1613-1621.

[17] W. Kluegel, M. A. Iqbal, F. Fioretto, W. Yeoh & E. Pontelli, « A Realistic Dataset for the Smart Home Device Scheduling Problem for DCOPs », in *Autonomous Agents and Multiagent Systems*, vol. 10643, 2017, p. 125-142.

[18] R. N. Lass, E. Sultanik & W. C. Regli, « Dynamic Distributed Constraint Reasoning », in *AAAI*, 2008, p. 1466-1469.

[19] R. T. Maheswaran, J. P. Pearce & M. Tambe, « Distributed Algorithms for DCOP: A Graphical-Game-Based Approach », in *ISCA PDCS*, 2004, p. 432-439.

[20] R. Maheswaran, J. Pearce & M. Tambe, « Distributed Algorithms for DCOP: A Graphical-Game-Based Approach », in *Proc. of the 17th International Conference on Parallel and Distributed Computing Systems (PDCS)*, 2004, p. 432-439.

[21] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser & G. Czajkowski, « Pregel: A System for Large-scale Graph Processing », in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, ACM, 2010, p. 135-146.

[22] P. J. Modi, W. Shen, M. Tambe & M. Yokoo, « ADOPT: Asynchronous distributed constraint optimization with quality guarantees », *Artificial Intelligence Journal* **161** (2005), nº 1, p. 149-180.

[23] Y. Naveh, R. Zivan & W. Yeoh, « Resilient Distributed Constraint Optimization Problems », in *8th International Workshop on Optimisation in Multi-Agent Systems (OPTMAS 2017)*, 2017, p. 14.

[24] M. T. Özsu & P. Valduriez, « Data Replication », in *Principles of Distributed Database Systems*, Springer, New York, third éd., 2011, p. 459-495.

[25] C. Parra, D. Romero, S. Mosser, R. Rouvoy, L. Duchien & L. Seinturier, « Using Constraint-Based Optimization and Variability to Support Continuous Self-Adaptation », in *Proceedings of the 27th Annual ACM Symposium on Applied Computing - SAC '12*, 2012, p. 486-491.

[26] F. Pecora & A. Cesta, « DCOP for Smart Homes: A Case Study », *Computational Intelligence* **23** (2007), nº 4, p. 395-419.

[27] A. Petcu & B. Faltings, « A scalable method for multiagent constraint optimization », in *International Joint Conference on Artificial Intelligence (IJCAI'05)*, 2005, p. 266-271.

[28] ———, « Superstabilizing, Fault-Containing Distributed Combinatorial Optimization », in *Proceedings of the National Conference on Artificial Intelligence*, vol. 20, 2005, p. 449-454.

[29] ———, « Optimal Solution Stability in Dynamic, Distributed Constraint Optimization », in *2007 IEEE/WIC/ACM International Conference on Intelligent Agent Technology (IAT'07)*, 2007-11, p. 321-327.

[30] S. Russel & P. Norvig, *Artificial Intelligence: a Modern Approach*, 3rd éd., Prentice-Hall, 2009.

[31] P. Rust, G. Picard & F. Ramparany, « On the Deployment of Factor Graph Elements to Operate Max-Sum in Dynamic Ambient Environments », in *8th International Workshop on Optimisation in Multi-Agent Systems (OPTMAS 2017)*, 2017.

[32] ———, « On the Deployment of Factor Graph Elements to Operate Max-Sum in Dynamic Ambient Environments », in *Autonomous Agents and Multiagent Systems – AAMAS 2017 Workshops, Best Papers, Sao Paulo, Brazil, May 8-12, 2017, Revised Selected Papers* (G. Sukthankar & J. A. Rodriguez-Aguilar, éds.), Lecture Notes in Artificial Intelligence (LNAI), vol. 10642, Springer, 2017, Extended Version, p. 116-137.

[33] ———, « pyDCOP, a DCOP library for IoT and dynamic systems », in *International Workshop on Optimisation in Multi-Agent Systems (OptMAS@AAMAS 2019)*, 2019.

[34] ———, « Résilience et auto-réparation de processus de décisions multi-agents », in *Systèmes Multi-Agents et simulation - Vingt-septièmes journées francophones sur les systèmes multi-agents, JFSMA 2019, Toulouse, France, July 3-5, 2019* (O. Simonin & S. Combettes, éds.), Cépaduès, 2019, p. 31-40.

[35] T. Saraç & A. Sipahioglu, « Generalized quadratic multiple knapsack problem and two solution approaches », *Computers & Operations Research* **43** (2014), nº Supplement C, p. 78-89.

[36] H. Song, S. Barrett, A. Clarke & S. Clarke, « Self-Adaptation with End-User Preferences: Using Run-Time Models and Constraint Solving », in *Model-Driven Engineering Languages and Systems*, vol. 8107, 2013, p. 555-571.

[37] A. Stimson, *Photometry and Radiometry for Engineers*, John Wiley & Sons Inc, 1974.

[38] M. Vinyals, M. Pujol, J. A. Rodriguez-Aguilar & J. Cerquides, « Divide-and-Coordinate: DCOPs by agreement », in *Proceedings of the 9th International Conference on Autonomous Agents and Multiagent Systems: Volume 1 - Volume 1*, AAMAS '10, International Foundation for Autonomous Agents and Multiagent Systems, 2010, p. 149-156.

[39] M. Vinyals, J. A. Rodriguez-Aguilar & J. Cerquides, « Constructing a unifying theory of dynamic programming DCOP algorithms via the generalized distributive law », *Autonomous Agents and Multi-Agent Systems* **22** (2010), nº 3, p. 439-464.

[40] R. Wattenhofer, « Principles of Distributed Computing », 2015.

[41] O. Wolfson & A. Milo, « The Multicast Policy and Its Relationship to Replicated Data Placement », *ACM Trans. Database Syst.* **16** (1991), nº 1, p. 181-205.

[42] W. Yeoh, P. Varakantham, X. Sun & S. Koenig, « Incremental DCOP Search Algorithms for Solving Dynamic DCOP Problems », in *2015 IEEE/WIC/ACM International Conference on Web Intelligence and Intelligent Agent Technology (WI-IAT)*, 2015, p. 257-264.

[43] M. Yokoo, T. Ishida, E. Durfee & K. Kuwabara, « Distributed Constraint Satisfaction for Formalizing Distributed Problem Solving », in *[1992] Proceedings of the 12th International Conference on Distributed Computing Systems*, 1992, p. 614-621.

[44] W. Zhang, G. Wang & L. Wittenburg, « Distributed Stochastic Search for Constraint Satisfaction and Optimization: Parallelism, Phase Transitions and Performance », in *Proceedings of AAAI Workshop on Probabilistic Approaches in Search*, 2002.

Glossaire

ABSTRACT. — In this paper we define the notion of k-resilience for computational graphs in support of agents' decisions on dynamic systems. We propose a method to self-repair the computation distribution, DRPM[DMCM], as to ensure the continuity of collective decisions following the disappearance of agents, through the deployment of replicas calculations. We are interested here in constraint optimization process repair, where the computations are decision variables or distributed constraints. We model a smart environment configuration problem as a distributed constraint optimization problem to be repaired by our repair techniques. We experimentally evaluate the performance of DRPM[DMCM] on different topologies of systems operating algorithms (A-MaxSumor A-DSA) to solve classic problems (random, graph coloring, Ising) and SECP instances, while agents disappear at runtime.

KEYWORDS. — DCOP, resilience, self-repair, smart environment.

Manuscrit reçu le 15 juillet 2021, révisé le 31 janvier 2022, accepté le 14 mars 2022.