P-Androide : Diagnostic and Value Of Information

Ariana Carnielli et Ivan Kachaikin

Encadrants : Paolo Viappiani et Pierre-Henri Wuillemin

Table des matières

1 Introduction

Ce document présente les travaux réalisés pour le projet « Diagnostic and Value of Information » du master M1 Androide à Sorbonne Université. Il commence par une description de la problématique traitée et les objectifs initiaux à travers le cahier des charges de la Section 2. On donne ensuite un panorama de la littérature disponible sur le sujet dans l'état de l'art de la Section 3. La Section 4 détaille la méthodologie choisie dans ce projet, avant la description plus détaillée de notre contribution à l'étude du problème traité dans la Section 5. Une description des implémentations réalisées est donnée dans la Section 6 avant la présentation des résultats dans la Section 7.

2 Cahier des charges

Des dispositifs tombent souvent en panne : logiciels, téléphones, ordinateurs, imprimantes, voitures, rames de métro, parmi de nombreux autres dispositifs, peuvent présenter des malfonctionnements empêchant leur utilisation. Dans ces situations, plus qu'identifier l'origine du problème, on cherche à le réparer, de préférence de la façon la moins « coûteuse » possible, le coût ici ne se limitant pas au coût financier mais pouvant aussi prendre en compte d'autres facteurs comme le temps de réparation. Ce processus de réparation se passe dans l'incertain, face à plusieurs inconnues comme l'état de fonctionnement des différents composants du dispositif, l'origine du problème, mais aussi possiblement l'absence d'informations exactes sur les différents coûts. L'établissement de protocoles et de séquences d'actions sur le dispositif permettant de minimiser la valeur espérée du coût total de sa réparation est la base du problème de diagnostic automatique, appelé ici *Troubleshooting*, objet d'étude de ce projet.

Ce projet est à l'origine un projet prospectif : son objectif est de chercher quel apport la théorie de l'élicitation — qui s'intéresse, face à une situation d'incertitude concernant les coûts, à trouver les « meilleures » questions à poser, celles qui apportent le plus d'information — peut avoir sur le problème de *Troubleshooting*, afin d'avoir des meilleures séquences d'actions pour réparer un dispositif.

Le cahier de charges de ce projet est le suivant :

- Étude de l'état de l'art sur le problème de *Troubleshooting*.
  L'objectif est d'étudier le problème de *Troubleshooting* à travers une revue de la littérature dans ce domaine, étudiant en détails les principaux articles fondateurs et quelques articles plus récents. Cela permet de, dans un premier temps, comprendre ce qu'est le *Troubleshooting*, la difficulté de l'utilisation d'algorithmes exacts à cause de leur complexité, et quelles sont les principaux algorithmes approchés utilisés et les heuristiques les motivant. L'état de l'art doit présenter aussi un bref aperçu de la théorie de l'élicitation, avec ses idées principales, afin de comprendre si ses méthodes peuvent être utiles dans un contexte de *Troubleshooting*.
- Implémentation de techniques classiques de *Troubleshooting*.
  Une fois la revue de la littérature complétée, l'objectif suivant de ce projet est de faire une implémentation de méthodes classiques de *Troubleshooting* présents dans les articles étudiés. Cette implémentation a vocation à devenir un outil de support à la recherche qui dépasserait le cadre de ce projet : elle doit être robuste et bien documentée afin que des travaux futurs

sur le *Troubleshooting* puissent s'y appuyer. L'implémentation doit ainsi être une maquette sur laquelle on peut s'appuyer pour faire des changements et voir expérimentalement si des nouvelles idées et heuristiques marchent. On travaille de façon incrémentale en implémentant d'abord les algorithmes plus simples d'abord avant de passer à ceux plus sophistiqués.

- Utilisation de techniques d'élicitation sur le problème de *Troubleshooting*.
  À partir de l'implémentation de techniques classiques de *Troubleshooting*, l'objectif sera d'étudier l'effet de l'utilisation des techniques d'élicitation permettant de réduire des incertitudes concernant les prix de réparation des composants du dispositif. Il s'agira ainsi, d'une part, de proposer des méthodes d'élicitation permettant de le faire et, d'autre part, de les implémenter dans le module de base de *Troubleshooting* afin de les tester et des les comparer aux techniques existantes.
- Implémentation d'une interface graphique pour des problèmes de *Troubleshooting*
  Outre l'implémentation des techniques classiques de *Troubleshooting* et l'exploration de nouvelles techniques à l'aide de la théorie de l'élicitation, un des objectifs de ce projet est de concevoir une interface graphique qui pourrait permettre à un utilisateur de résoudre une panne d'un dispositif à l'aide des algorithmes implémentés. Cette interface doit être toujours destinée à la recherche, en présentant par exemple à chaque étape les coûts espérés de réparation sous chaque action possible, mais en gardant l'idée qu'elle devrait être facilement adaptable à une situation réelle.
- Études expérimentales et présentation des résultats.
  À la fin, le dernier objectif de ce projet sera de présenter les résultats de l'implémentation réalisée sur un cas concret. Le résultat espéré est que les techniques implémentées avec l'élicitation permettent d'améliorer les techniques existantes dans le cas où les coûts de réparation ne sont pas connus de façon exacte.

3 État de l'art

3.1 Description générale du problème de *Troubleshooting*

Étant donné un dispositif en panne, le problème du *Troubleshooting* consiste à chercher une stratégie de réparation de coût total minimal. Plus précisément, on considère que le dispositif est constitué d'un nombre fini de composants $c_1, \dots, c_n$, certains pouvant être en panne, et que l'on dispose de 2 types d'actions qui peuvent être réalisées de façon séquentielle : observations et réparations.

Les observations, que l'on dénote par $o_1, \dots, o_m$, peuvent être « locales », c'est-à-dire d'un seul composant du dispositif, ou « globales » quand elles dépendent de plusieurs composants. Il peut y avoir de composants qui n'ont pas d'observation locale associée. On considère aussi qu'on a une observation spéciale o_0 qui porte sur l'état général du dispositif. Les réparations portent toujours sur un seul composant à la fois et on note r_i la réparation du composant c_i.

Les ensembles d'observations, réparations et actions sont dénotés respectivement par $\mathcal{O} = \{o_0, \dots, o_m\}$, $\mathcal{R} = \{r_1, \dots, r_n\}$ et $\mathcal{A} = \mathcal{O} \cup \mathcal{R}$. Chaque action $a \in \mathcal{A}$ a un coût associé $C(a) \geq 0$ et l'objectif est donc de mettre le dispositif en état de marche en minimisant le coût total

$$\sum_{a \in \mathcal{A}_*} C(a),$$

où $\mathcal{A}_*$ est l'ensemble des actions prises. Dans certaines situations, il peut être intéressant de rajouter à $\mathcal{A}$ une action spéciale, a_0, correspondant à « appeler le service », qui résout le problème avec certitude mais a un coût très élevé, représentant, par exemple, la possibilité d'envoyer le dispositif à un centre plus spécialisé ou d'en acheter un nouveau.

3.2 Théorie de la décision et fonctions d'utilité

Le problème de minimisation présenté comporte des difficultés liées aux incertitudes: on ne connaît pas quels sont les composants défectueux, quels seront les résultats des observations ni les conséquences d'une réparation sur l'ensemble du dispositif. Il nous faut ainsi prendre des décisions dans l'incertain et, afin de traiter ce problème, on se sert des outils de la théorie de la décision telle que décrite de façon générale dans North (1968). Il s'agit d'un cadre formel qui permet de faire des choix d'actions parmi des alternatives lorsque les conséquences de ces actions ne sont connues que dans un sens probabiliste. Cette théorie repose sur la modélisation probabiliste et la représentation des préférences d'un agent, de façon synthétique, à travers une fonction d'utilité.

L'idée d'une fonction d'utilité est de donner des valeurs numériques à des résultats. Une hypothèse fondamentale faite pour cela est de supposer que chaque paire de résultats peut être comparée : un des résultats sera forcément meilleur ou aussi bon que l'autre (axiome de *complétude*). On demande aussi à ce que cette comparaison des résultats soit transitive : préférer A à B et B à C implique préférer A à C (axiome de *transitivité*). Une autre hypothèse fondamentale est que l'on peut comparer non seulement des résultats purs mais aussi des loteries, c'est-à-dire des situations où l'on peut avoir quelques résultats avec une certaine probabilité. En particulier, si un agent préfère A à B, alors, si on lui donne le choix entre deux loteries entre A et B, l'agent préférera la loterie donnant plus de probabilité à A.

Les loteries n'ont pas de valeur intrinsèque, ce qui implique que l'on n'a pas intérêt à faire des loteries imbriquées, où le prix d'une loterie serait de participer à une deuxième loterie (axiome d'*indépendance*). Finalement, on suppose une continuité des loteries: si l'agent a l'ordre de préférence $A > C > B$, alors il existe une loterie entre A et B telle que l'agent sera indifférent entre cette loterie et le résultat C (axiome de *continuité*). Sous ces hypothèses de complétude, transitivité, indépendance et continuité, d'après le Théorème d'utilité de von Neumann–Morgenstern von Neumann and Morgenstern (1953), il est possible de condenser les préférences de l'agent dans une fonction d'utilité u telle que $u(A) > u(B)$ si et seulement si l'agent préfère A à B. En plus, si l'agent est indifférent entre C et une loterie entre A et B avec probabilités respectives P et $1 - P$, alors

$$u(C) = Pu(A) + (1 - P)u(B),$$

c'est-à-dire, l'utilité de la loterie est l'espérance de l'utilité du gain.

Dans un cadre simple avec une seule action à prendre parmi $\alpha_1, \ldots, \alpha_k$ et des résultats possibles entre $R_1, \ldots, R_\ell$, on calcule l'espérance de l'utilité de l'action α_i par

$$\mathbb{E}u(\alpha_i) = \sum_{j=1}^{\ell} u(R_j)P(R_j \mid \alpha_i)$$

et on choisit celle qui maximise cette utilité espérée. En général, on doit faire face à des problèmes avec une séquence de décisions que l'on peut représenter par un arbre, comme celui de la Figure 1, mais dont le calcul exhaustif en général est trop complexe, nécessitant ainsi de méthodes d'approximation permettant de résoudre le problème en temps raisonnable.

Dans le cadre du *Troubleshooting*, les actions α_i sont des observations ou réparations de $\mathcal{A}$. L'utilité d'un résultat R_i est le coût des actions qui mènent de la racine à la feuille R_i, la maximisation de l'utilité est remplacée par la minimisation du coût, et tous les résultats R_i représentent la même situation, à savoir le fonctionnement normal du dispositif.

3.3 Valeur de l'information myope

Les concepts de bases de la théorie de la valeur de l'information sont présentés dans l'article Howard (1966). Plus précisément, supposons que l'on doit prendre une décision D dans l'incertain parmi celles proposées sachant que l'on connaît un ensemble d'évidences E. Supposons en

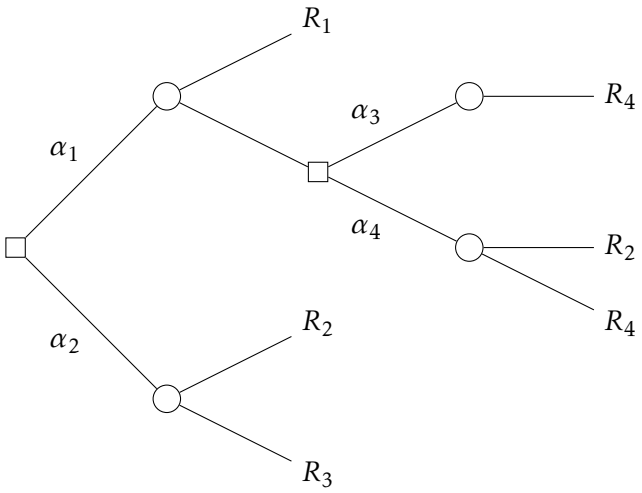

FIGURE 1 – Arbre avec une séquence d'au plus deux décisions à prendre. Les nœuds carrés représentent les décisions à prendre et ceux en forme de cercle, les loteries.

plus qu'il existe un voyant qui est capable de fournir une autre évidence X, ce qui diminue les incertitudes. Néanmoins, obtenir X ne serait pas gratuit : pour qu'un voyant nous dise son secret il faut lui payer C_X. Serait-ce alors mieux de payer au voyant ou doit-on prendre plutôt une décision sans informations additionnelles ? L'idée principale de l'approche proposée dans Howard (1966) est de comparer l'utilité que l'on obtiendrait sans et avec l'évidence fournie par le voyant. Cette approche est myope car on considère que l'on ne cherchera pas d'autres informations après avoir obtenu X.

Si $\mathbb{E}u(D \mid E)$ est l'utilité espérée lorsque l'on prend la décision D sachant E, selon la technique proposée, il suffit de connaître la valeur

$$\mathrm{EVOI}(X,E) = \max_D \sum_{x_i} \mathbb{E}u(D \mid X = x_i, E) P(X = x_i \mid E) - \max_D \mathbb{E}u(D \mid E), \tag{1}$$

donc la différence entre l'utilité espérée maximale si l'on connaît X et celle sans connaître X. Selon ce critère, on décide alors de payer pour évidence X si $\mathrm{EVOI}(X,E) > 0$ et prendre une décision directement sinon. De plus, si l'on avait un choix entre des évidences $X_1, X_2, \ldots, X_n$, on choisirait celle qui apporterait à $\mathrm{EVOI}(X_i,E)$ une valeur maximale toujours en vérifiant que $\mathrm{EVOI}(X_i,E) > 0$.

Cette idée est aussi celle utilisée dans l'approche myope au problème de *Troubleshooting* décrite ci-après dans la Section 3.5.3. On peut voir l'introduction des observations globales comme des actions pour obtenir plus d'information. Le choix si une de ces actions sera prise ou pas est fait à travers un critère tout à fait analogue à celui décrit par (1).

3.4 Réseaux Bayésiens

Les incertitudes dans le problème de *Troubleshooting* ont plusieurs origines : on ne connaît pas quels composants sont en panne ni les effets précis que le changement de l'état d'un composant peut avoir sur les autres composants, sur les observations et sur l'état du système. Ces incertitudes sont représentées dans notre approche par des probabilités. La représentation intégrale de la loi de probabilité jointe de tous les composants et de toutes les observations du système serait trop gourmande en mémoire et inutilement complexe puisque l'on peut imaginer qu'il y a plusieurs relations d'indépendance ou d'indépendance conditionnelle entre elles. Ainsi, les réseaux Bayésiens, décrits par exemple dans Jensen and Nielsen (2007), sont un outil mathématique adaptée à la représentation des probabilités de notre problème.

Les réseaux Bayésiens permettent de simplifier la représentation des lois de probabilité grâce aux relations d'indépendance conditionnelle entre les variables. Plus précisément, un réseau Bayé-

sien est défini comme un graphe orienté acyclique dans lequel les nœuds représentent les variables d'intérêt et, à chaque nœud X, on associe une loi de probabilité conditionnelle du type

$$P(X \mid Y_1, \ldots, Y_n),$$

où $Y_1, \ldots, Y_n$ sont les parents immédiats de X dans le graphe (si X n'a pas de parents dans le graphe, on associe à son nœud sa loi $P(X)$). Dans cette situation, conditionnellement à ses parents immédiats $Y_1, \ldots, Y_n$, un nœud X est indépendant de tout autre nœud qui n'est pas un de ses descendants. La loi de probabilité jointe de toutes les variables peut être déterminée par multiplication des lois de probabilité de chaque nœud.

Pour la construction d'un tel graphe dans des situations pratiques, les flèches ne représentent pas forcément de lien causal, puisque les parents influencent uniquement la *probabilité* des enfants. L'important est que deux nœuds conditionnellement indépendants dans le graphe représentent bien des variables conditionnellement indépendantes dans le cas pratique en question. Pour la manipulation des réseaux Bayésiens, on utilise la bibliothèque `pyAgrum` de Python Gonzales et al. (2017), qui contient, parmi d'autres modules, une implémentation efficace et facilement manipulable de ce type de réseau.

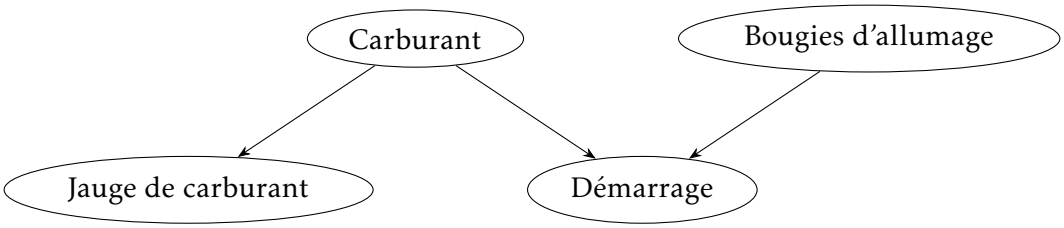

FIGURE 2 – Exemple de réseau bayésien pour le problème de démarrage de la voiture

Un exemple classique de réseau Bayésien pour le *Troubleshooting*, présenté dans Jensen and Nielsen (2007), est celui du *problème de démarrage de la voiture*, dans lequel on considère une voiture qui ne démarre pas. La Figure 2 représente ce réseau, avec quatre nœuds représentant quatre variables de la voiture (démarrage ou pas, affichage de la jauge de carburant, présence de carburant, fonctionnement des bougies d'allumage). Dans ce réseau, par exemple, la probabilité que la voiture démarre est exprimée comme une loi conditionnelle dépendant de la présence de carburant et du fonctionnement des bougies. La probabilité que la jauge de carburant affiche qu'il y a du carburant ou pas est représentée comme une loi conditionnelle dépendant uniquement de la présence de carburant. Le fonctionnement des bougies et la présence de carburant sont représentés par deux lois de probabilité simples, sans conditionnement.

3.5 Approches au problème de *Troubleshooting*

3.5.1 Approche exacte

L'approche immédiate pour résoudre le problème *Troubleshooting* est de construire l'arbre correspondant avec toutes les actions d'observation et réparation qui peuvent être réalisées à chaque étape, les feuilles correspondant aux états où le dispositif a été réparé après une séquence d'actions. À chaque feuille correspond ainsi un coût total de la réparation qui mène à cette feuille. On peut alors calculer les coûts espérés dans chaque nœud de façon inductive à partir des feuilles : dans un nœud de loterie (résultat d'une observation ou d'une réparation), on calcule l'espérance des coûts de ses nœuds fils alors que, dans un nœud de décision (choix d'une observation ou réparation), on prend comme valeur la valeur du fils avec le plus petit coût espéré. Cette approche permet de résoudre le problème de façon exacte, mais en temps exponentiel en la taille des données car il faut parcourir tout l'arbre pour déterminer le meilleur choix en chaque nœud.

Le fait que cet algorithme exact est exponentiel n'est pas surprenant : il a été démontré dans Vomlelová (2003) que, sauf sous des hypothèses simplificatrices assez fortes (par exemple, absence d'observations et présence d'un unique composant en panne sur le système), le problème de

Troubleshooting est NP-difficile. Cela motive ainsi la recherche d'heuristiques donnant de bonnes solutions pratiques ainsi que d'algorithmes approchés pour le problème de *Troubleshooting*. Il faut cependant noter que, en toute généralité, le problème de *Troubleshooting* est aussi NP-difficile à résoudre dans un sens approché, comme démontré dans Lín (2014).

3.5.2 Algorithme exacte sous hypothèses simplificatrices

Sous des hypothèses simplificatrices assez restrictives, il est connu Heckerman et al. (1994, 1995); Lín (2014); Vomlelová (2003) qu'il est possible de résoudre le problème de *Troubleshooting* en temps polynomial. Plus précisément, on suppose que :

- il n'y a qu'un seul composant présentant un défaut (*single fault assumption*) ;
- les coûts des réparations sont indépendants ;
- la seule observation possible est celle de l'état du dispositif, o_0, qui a un coût 0.

À travers le réseau Bayésien décrivant le dispositif, on dispose, pour tout $i \in [\![1, n]\!]$, de la probabilité p_i que la réparation r_i résout le problème. L'algorithme polynomial consiste alors à calculer les rapports $\frac{p_i}{C(r_i)}$ et réparer les composants dans l'ordre décroissant de ces rapports, observant après chaque réparation si le dispositif marche ou pas.

Ces hypothèses sont trop restrictives car, d'une part, il est simpliste de supposer qu'on n'a qu'un seul composant en panne et, d'autre part, on dispose souvent d'autres observations outre o_0 et les informations qu'elles peuvent apporter peuvent être assez importantes pour que l'on les ignore. Cette deuxième remarque est en lien avec la notion de *valeur de l'information* : la valeur qu'une information apporte, dans le cadre du *Troubleshooting*, est la différence entre le coût espéré de réparation sans cette information et le coût en prenant en compte l'information (auquel on ajoute aussi le coût d'obtention de l'information à travers une observation). Cette notion s'applique à des problèmes de décision plus généraux que le *Troubleshooting*, comme décrit dans Braziunas and Boutilier (2008) pour les problèmes d'élicitation. Ces derniers seront détailles ci-après dans la Section 3.6.

3.5.3 Approche myope

Les articles Heckerman et al. (1994, 1995) présentent un algorithme heuristique pour le problème de *Troubleshooting* qui fait des hypothèses moins restrictives que les précédentes. On suppose désormais qu'il peut y avoir plusieurs composants en panne mais on restreint les observations que l'on peut faire. Pour les composants non-observables (c'est-à-dire, qui n'ont pas d'observation locale associée), la seule action disponible est leur réparation. Pour les composants observables, on impose de toujours faire l'observation locale correspondante avant la réparation, ce qui est appelé une *paire observation-réparation*. Ainsi, on restreint l'ensemble $\mathcal{A}$ des actions possibles aux réparations de composants non-observables et aux paires observation-réparation pour les autres composants. Si r dénote l'action de réparation d'un composant observable et o dénote l'observation de ce même composant, alors le coût de la paire $a = (o, r)$ est

$$C^{or}(a, E) = C(o) + P(o \neq \text{normal} \mid E)C(r), \tag{2}$$

où E représente les informations dont on dispose. On définit $C^{or}(a, E) = C(r)$ dans le cas où $a = r$ représente la réparation d'un composant non-observable et $C^{or}(a_0, E) = C(a_0)$ pour l'action spéciale a_0 d'appel au service. L'algorithme suit la même idée que celui de la Section 3.5.2, en choisissant à chaque étape le plus grand rapport probabilité/coût, mais ces rapports doivent désormais être recalculés à chaque étape car les probabilités et les coûts évoluent en fonction des actions déjà effectuées.

À la fin de l'algorithme, cette heuristique basée sur les observations-réparations donnera une séquence d'actions $S^* = (a_1, \ldots, a_k)$ à prendre dans l'ordre, où chaque action a_i représente la réparation d'un composant non-observable, la paire observation-réparation d'un composant observable ou l'appel au service. Le coût espéré de réparation associé à cette séquence d'actions S^* calculée à

partir des informations initiales E_0, $\mathrm{ECR}(E_0, S^*)$, est alors donné par la formule

$$\begin{aligned}\mathrm{ECR}(E_0, S^*) = {} & C^{or}(a_1, E_0) \\ & + P(o_0 \neq \text{normal} \mid E_1) C^{or}(a_2, E_1) \\ & + P(o_0 \neq \text{normal} \mid E_1) P(o_0 \neq \text{normal} \mid E_2) C^{or}(a_3, E_2) \\ & + \cdots \\ & + P(o_0 \neq \text{normal} \mid E_1) P(o_0 \neq \text{normal} \mid E_2) \cdots P(o_0 \neq \text{normal} \mid E_{k-1}) C^{or}(a_k, E_{k-1}),\end{aligned} \tag{3}$$

où, pour $j \in \{1, \ldots, k-1\}$, E_j représente les informations à la fin de l'étape j, c'est-à-dire E_0 rajouté des informations que les composants correspondant aux actions $a_1, \ldots, a_j$ ont été réparés. Cette formule peut aussi s'écrire de façon récursive comme

$$\mathrm{ECR}(E_0, (a_1, \ldots, a_k)) = C^{or}(a_1, E_0) + P(o_0 \neq \text{normal} \mid E_1)\, \mathrm{ECR}(E_1, (a_2, \ldots, a_k)).$$

La contribution principale de Heckerman et al. (1994, 1995) est une autre approche heuristique qui, par rapport au cas précédent, rajoute la possibilité de faire des observations globales en dehors des paires observation-réparation. Pour éviter la complexité du cas général, ils développent une technique appelée *myope*, qui consiste à calculer l'espérance de coût après une observation globale o_i de façon approchée en supposant qu'aucune autre observation globale ne sera faite dans la suite. Cela donne ainsi l'espérance de coût myope suite à l'observation o_i, $\mathrm{ECO}(o_i, E, S^*)$, donnée par

$$\mathrm{ECO}(o_i, E, S^*) = C(o_i) + \sum_{j=1}^{n_i} \mathrm{ECR}(E \cup \{o_i = j\}, S^*) P(\{o_i = j\} \mid E),$$

où l'on considère que l'ensemble des résultats possibles de l'observation o_i est $\{1, \ldots, n_i\}$. À chaque étape, on compare ces espérances de coût $\mathrm{ECO}(o_i, E, S^*)$ (une pour chaque observation globale) avec l'espérance de coût sans observation $\mathrm{ECR}(E, S^*)$ (celle que l'on obtiendrait en appliquant l'algorithme précédent) pour décider s'il est intéressant de réaliser une observation globale à cette étape ou pas. On remarque aussi que la différence entre $\mathrm{ECO}(o_i, E, S^*)$ et $\mathrm{ECR}(E, S^*)$ peut être vue comme la valeur espérée de l'information myope apportée par l'observation o_i, dans le même sens que la formule de EVOI (1) de la Section 3.3.

Les travaux Heckerman et al. (1994, 1995) présentent aussi des méthodes pour calculer les probabilités de réparation en utilisant des réseaux Bayésiens. Pour ce faire, il est nécessaire non seulement de calculer ces probabilités mais aussi de les mettre à jour en fonction des informations acquises lors d'observations et de réparations. Afin de simplifier ce calcul, les articles introduisent la notion de réseaux de réponse, construits à partir d'un réseau Bayésien et d'une action effectuée. Des simplifications supplémentaires sont encore possibles sous l'hypothèse d'indépendance causale. Cet algorithme a été testé et validé pour certains modèles concrets.

3.5.4 Extensions de l'approche myope

Des extensions de l'approche de Heckerman et al. (1994, 1995) ont été proposées en particulier dans Jensen et al. (2001); Langseth (2003) où les méthodes développés ont permis d'obtenir des résultats assez efficaces pour des cas plus généraux. Plus spécifiquement, l'article Jensen et al. (2001) considère, d'abord, que les composants et les actions de réparation ne sont plus en correspondance univoque et que chaque action peut traiter les composants associés avec une certaine probabilité. Ainsi, les actions ne sont plus parfaites et ne conduisent pas toujours vers une réparation d'un composant. Par ailleurs, on suppose que chaque action a la possibilité de dépanner plusieurs composants et, réciproquement, chaque composant peut être réparé par des actions différentes. De plus, cet article propose une approche qui améliore la technique myope décrite ci-dessus.

Quant à l'article Langseth (2003), l'auteur y considère un cas encore plus général où chaque composant est constitué de sous-composants qui peuvent eux-mêmes être à l'origine de la panne

du dispositif et qui peuvent être réparées. En outre, les composants, vus comme des ensembles de sous-composants, ne sont pas forcément deux-à-deux disjoints. En conséquence, dans ce modèle il est possible qu'un sous-composant X soit partie de deux composants différents, c_i et c_j, $i \neq j$. Selon des résultats de simulations numériques de Langseth (2003), les algorithmes développés retournent des stratégies pour le *Troubleshooting* beaucoup plus efficaces que l'approche myope pour des problèmes concrets. En effet, pour les modèles considérés, les techniques de Jensen et al. (2001); Langseth (2003) retournent des solutions dont le coût espéré de réparation a un écart relatif moyen de 2,51% par rapport à l'optimum trouvé par une recherche exhaustive, au lieu de 21,5% pour l'approche myope.

3.6 Élicitation

3.6.1 Élicitation dans le *Troubleshooting*

Dans les problèmes de la théorie de la décision en général, la fonction d'utilité n'est pas parfaitement connue et il nous faut alors des mécanismes d'élicitation permettant de l'estimer. Cela est bien le cas du problème de *Troubleshooting* qui nous intéresse : malgré le fait que l'utilité provient du coût et que les coûts des actions individuelles sont connus, la connaissance du coût de réparation de façon exacte impliquerait un parcours complet de l'arbre des décisions, ce qui n'est pas réalisable dans la plupart des cas. Les articles Braziunas and Boutilier (2005, 2008) présentent le problème d'élicitation des fonctions d'utilité et donnent un panorama des techniques pour le résoudre.

L'idée principale est de considérer que la fonction d'utilité dépend des résultats à travers d'un certain nombre de caractéristiques de ces résultats. Autrement dit, on représente un résultat R comme un vecteur de caractéristiques $(x_1,\ldots,x_d)$ et on regarde $u(R)$ comme $u(x_1,\ldots,x_d)$. Les articles Braziunas and Boutilier (2005, 2008) supposent alors que u satisfait une hypothèse d'*indépendance additive généralisée*, ce qui permet de l'écrire comme combinaison linéaire de fonctions $u_1,\ldots,u_p$, chacune dépendant seulement d'une partie des caractéristiques x_i, par exemple

$$u(x_1,x_2,x_3,x_4) = \lambda_1 u_1(x_1) + \lambda_2 u_2(x_2,x_4) + \lambda_3 u_3(x_3,x_4). \tag{4}$$

Ainsi, l'élicitation peut être décomposée en deux étapes, une locale correspondant à estimer les u_i et une globale afin de déterminer les λ_i. Il présente aussi deux techniques pour représenter les incertitudes sur la fonction d'utilité, basées sur une approche Bayésienne et une approche ensembliste. Celle qui nous intéresse est la Bayésienne, qui repose sur la notion de valeur de l'information, comme expliqué précédemment.

3.6.2 Approche Bayésienne

Afin d'avancer vers le problème d'élicitation, il faut décider comment on pourrait représenter les incertitudes sur l'utilité d'un utilisateur en particulier. L'article Chajewska et al. (2000) utilise une hypothèse centrale : étant donnée une conséquence O d'une décision D, on considère que son utilité u_O est une variable aléatoire qui suit une loi de probabilité $P(u_O)$. L'article suppose empiriquement que la loi P est une mixture de gaussiennes, possiblement tronquées. Lorsque l'on dispose des statistiques assez significatives, on aura la possibilité de reconstruire sa densité de probabilité pour un utilisateur générique. Il faut cependant adapter la loi pour chaque nouvel utilisateur pour prendre en compte les différences de leurs utilités.

Pour faire cette adaptation, on pose des questions d'élicitation, dont les réponses mettent à jour la loi de probabilité de l'utilité. Bien que ces questions nous donnent des informations, on ne doit pas trop en poser car l'utilisateur peut se fatiguer de leur répondre. Par conséquent, il est nécessaire de déterminer un critère pour choisir quelles questions il vaut mieux poser. L'article Chajewska et al. (2000) définit de manière assez similaire qu'à la Section 3.3 une notion de *valeur de l'information* utilisée pour déterminer la question suivante à poser. Cependant, on remarque que cette approche est « *myope* » car on ne considère que la valeur de l'information

locale de chaque question. Une technique plus générale consisterait à observer toutes les combinaisons possibles que l'on pourrait construire à partir de questions prises en compte, mais il est bien évident que, pour la grande majorité de cas, une telle technique est intraitable à cause du nombre des combinaisons possibles qui augmente trop vite. C'est pourquoi l'approche proposée dans Chajewska et al. (2000) est une solution assez efficace afin de traiter ce problème.

Par ailleurs, l'article Boutilier (2002) fournit une extension de Chajewska et al. (2000) qui approfondit l'idée de considérer la valeur d'utilité comme une variable aléatoire. En effet, il propose de modéliser le problème d'élicitation par un *processus de décision markovien partiellement observable* (POMDP pour *Partially Observed Markov Decision Process* en anglais) qui est une généralisation des *processus de décision markoviens* (MDP pour *Markov Decision Process* en anglais) où des chaînes des Markov simples sont remplacées par des chaînes cachées. L'article Boutilier (2002) suppose plus précisément que l'ensemble d'états du système modélisé est U, l'ensemble de toutes les fonctions d'utilité possibles, alors que l'ensemble d'observations est tout simplement l'ensemble de toutes les densités de probabilité définies sur U. De plus, le système est capable d'effectuer deux types d'actions : poser des questions et prendre une décision. Ensuite, des méthodes particulières développées pour POMDP sont utilisées afin de résoudre le problème d'élicitation posé au début.

Les techniques proposées dans Boutilier (2002); Chajewska et al. (2000) ont été développés dans Braziunas and Boutilier (2005, 2008), comme décrit dans la Section 3.6.1, pour des fonctions d'utilité satisfaisant l'hypothèse d'indépendance additive généralisée (voir (4) dans la Section 3.6.1). Ces articles montrent comment exploiter une élicitation myope utilisant la notion de *valeur d'information* et proposent un moyen graphique pour représenter les relations d'indépendance.

Une autre extension possible de Boutilier (2002); Chajewska et al. (2000) est d'utiliser plutôt des *ensembles optimaux de recommandations* pour choisir quelles questions poser au lieu de les déterminer de façon séquentielle. Cette idée a été explorée, par exemple, dans les articles Price and Messinger (2005); Viappiani and Boutilier (2010), et consiste à déterminer un ensemble d'alternatives « convenables » parmi celles proposées.

Plus précisément, étant donné un décideur avec sa propre fonction d'utilité, on doit trouver les m meilleures solutions parmi $\alpha_1, \alpha_2, \ldots, \alpha_n$, où $m < n$. L'approche la plus simple consiste à trier les stratégies α_i en ordre décroissante de leurs utilités espérées et prendre ensuite les m premières décisions de la liste triée. En pratique, cette approche n'est pas toujours suffisante Price and Messinger (2005) car dans la grande majorité de cas on ne connaît l'utilité du décideur qu'avec des incertitudes. Il serait alors possible de choisir m solutions trop similaires, en particulier si n est relativement grand. Par exemple, supposons que l'on gère un magasin de livres en ligne et que l'on cherche un ensemble de 10 livres à montrer à un utilisateur particulier qui seraient les plus susceptibles de l'intéresser. Supposons que l'on a élicité son utilité et que l'on lui propose les 10 meilleures alternatives selon cette utilité. Comme chaque livre a presque toujours des éditions différentes, il est souhaitable de ne montrer à cet utilisateur qu'une seule proposition parmi ces plusieurs éditions.

Bien que l'on puisse introduire un critère de similarité selon lequel on filtre des alternatives, une approche différente a été proposée dans l'article Price and Messinger (2005) : on définit plutôt un critère pour chaque sous-ensemble de $\{\alpha_1, \alpha_2, \ldots, \alpha_n\}$ et on cherche ensuite un sous-ensemble qui maximisera ce critère. Ce sous-ensemble est appelé *ensemble optimal de recommandations* (de *Optimal Recommendation Set* en anglais). On note que cette technique assure une *diversité* des alternatives dans l'ensemble trouvé, ce qui permet de proposer à l'utilisateur des bonnes solutions malgré les incertitudes de son utilité. En plus, le choix de l'utilisateur dans cette ensemble contient aussi des informations sur ses préférences, ce qui peut alors être utilisé pour améliorer des connaissances sur sa fonction d'utilité, comme détaillé dans Viappiani and Boutilier (2010), qui relie les ensembles optimaux de recommandations avec les ensembles de questions à choix multiples maximisant la valeur espérée de l'information.

Pour le problème de *Troubleshooting*, il se peut que l'on ne connaisse pas les coûts des obser-

vations ou réparations de façon exacte, ce qui correspond à ne pas connaître de façon précise la fonction d'utilité. Il est alors important d'être capable de prendre en compte ces incertitudes pour la recommandation d'actions et éventuellement recommander plusieurs actions à la fois pour que l'utilisateur puisse choisir celle qu'il trouve plus convenable. Les techniques de Price and Messinger (2005); Viappiani and Boutilier (2010) peuvent alors être utiles pour donner une liste d'actions diverses qui pourra en plus, à partir du choix de l'utilisateur, d'avoir des informations supplémentaires sur sa fonction d'utilité.

3.6.3 Approche ensembliste

L'article Iyengar et al. (2001) utilise une autre approche pour la représentation des incertitudes lors de l'élicitation, basée sur des ensembles plutôt que des lois de probabilité. Les auteurs y considèrent un problème légèrement différent du nôtre, consistant à déterminer l'utilité d'un ensemble d'objets $\mathcal{O} = \{o_1, o_2, \ldots, o_m\}$, chaque objet étant décrit par n attributs $x_1, x_2, \ldots, x_n$. L'hypothèse faite dans l'article est que l'utilité d'un objet est une combinaison linéaire de fonctions de chacun de ses attributs, c'est-à-dire, l'utilité de l'objet $o = (x_1, x_2, \ldots, x_n)$ est donnée par

$$u(o, w) = \sum_{i=1}^{n} w_i f_i(x_i),$$

où les fonctions $f_1, f_2, \ldots, f_n$ sont connues mais les poids $w_1, w_2, \ldots, w_n$, avec $w_i \geq 0$ pour tout $i \in \{1, \ldots, n\}$, sont à déterminer. Plutôt que déterminer les valeurs précises des w_i, l'information disponible sur ces coefficients est représentée par un ensemble P auquel on est sûr qu'ils appartiennent. Cet ensemble est alors mis à jour lors que des nouvelles informations sont acquises.

Sans perte de généralité et sans aucune autre information a priori sur les w_i, P est défini initialement par

$$P = \left\{ (w_1, \ldots, w_n) \in [0,1]^n \;\middle|\; \sum_{i=1}^{n} w_i = 1 \right\}.$$

On considère que les valeurs qui seront prises par $w_1, \ldots, w_n$ correspondent au centre de P. Pour améliorer les connaissances des w_i, et par conséquent de la fonction u, on cherche à diminuer P à chaque étape. Pour cela, on propose itérativement à l'utilisateur de comparer des paires d'objets, mettant à jour à chaque fois la région P et recalculant les w_i en prenant le centre de P actualisé.

On cherche à minimiser l'hypervolume de P afin que les poids soient déterminés avec la plus petite erreur possible. Ainsi, les comparaisons entre objets (o_i, o_j) présentées à l'utilisateur sont déterminées pour rétrécir le plus la région P. On boucle ce processus jusqu'à ce que le critère de terminaison soit vérifié, par exemple après un nombre maximal de questions posées à l'utilisateur ou lorsque la taille de P est suffisamment petite. L'article Iyengar et al. (2001) présente une implémentation de ces idées, avec une description détaillée du choix des objets à comparer à partir de considérations géométriques. L'algorithme nécessite aussi le calcul des centres des régions P à chaque étape, ce qui est un problème non-trivial.

Une autre approche pour ce même problème est celle du regret minimax, décrit par exemple dans Boutilier et al. (2006). Le principe reste celui de définir initialement un ensemble P pour les coefficients $(w_1, \ldots, w_n)$ de la même façon et de chercher à poser des questions à l'utilisateur de façon à réduire la taille de P. L'idée du regret minimax est de, à chaque fois, déterminer l'objet o_* qui minimise le regret maximal par rapport à tous les autres objets et tous les éléments de P, soit

$$o_* = \operatorname*{argmin}_{o \in \mathcal{O}} \max_{q \in \mathcal{O}} \max_{w \in P} u(q, w) - u(o, w).$$

On choisit alors son pire adversaire q_*, celui qui maximise son regret, soit

$$q_* = \operatorname*{argmax}_{q \in \mathcal{O}} \max_{w \in P} u(q, w) - u(o_*, w),$$

et on demande à l'utilisateur de comparer o_* et q_*. La réponse de l'utilisateur donne ainsi une inégalité entre $u(q_*, w)$ et $u(o_*, w)$, qui est utilisée pour réduire la région P. On itère jusqu'à ce que P soit suffisamment petit ou que le regret soit nul.

4 Analyse et méthodologie choisie

En vue de la diversité de points de vue sur le problème du *Troubleshooting* et des sujets connexes présentés dans la Section 3, nous fixons dans cette section le cadre formel choisi pour représenter le problème de *Troubleshooting* dans ce projet. On donne ainsi dans cette section une présentation plus détaillée des outils survolés dans l'état de l'art qui seront utilisés dans la suite, avec les objectifs d'unifier les notations pour la suite du document et de donner une définition précise de toute la terminologie utilisée ci-après.

4.1 Description du réseau Bayésien

Un dispositif en panne est représenté par un *réseau Bayésien*, tel que décrit dans la Section 3.4, c'est-à-dire un graphe orienté $G = (V, E)$ avec, dans chaque sommet $s \in V$, une loi de probabilité conditionnelle $P(s \mid \text{parents}(s))$. Dans ce document, on identifiera les sommets du réseau Bayésien et les variables aléatoires qu'ils représentent.

Un des sommets de ce graphe correspond à l'observation spéciale o_0 de l'état global du système et est appelé *problem defining node*. Ce nœud peut avoir plusieurs valeurs mais, par souci de simplicité, on considérera dans la suite qu'il n'en a que deux, correspondant au fonctionnement normal du système ou pas.

Un autre nœud spécial du réseau est celui d'« appel au service », comme décrit dans la Section 3.1. Ce nœud a deux valeurs, *yes* ou *no*, correspondant au fait que le service a été appelé ou pas. Ce nœud ne doit pas avoir de parent dans le réseau Bayésien, sa loi de probabilité initiale doit être une probabilité de 100% de *no*, et il doit satisfaire que $P(o_0 = \text{normal} \mid \text{appel au service} = \text{yes}) = 1$, c'est-à-dire qu'un appel au service doit forcément résoudre le problème.

Les composants $c_1, \ldots, c_n$ sont représentés par des sommets de V. Tous les composants sont réparables, et l'action de réparation du composant c_i représenté par le sommet $s_i \in V$ correspond à considérer l'événement $s_i = \text{normal}$. Certains composants sont aussi observables, ce qui représente le fait que l'on peut observer la valeur de la variable aléatoire s_i.

Les observations globales, c'est-à-dire celles ne correspondant pas à l'observation d'un seul composant, sont aussi représentées par des sommets de V. L'ensemble des sommets V contient ainsi les deux sommets spéciaux de *problem defining node* et d'appel au service, les composants et les observations globales, mais cette inclusion peut ne pas être une égalité : on autorise à V d'avoir d'autres sommets qui ne rentreraient pas dans ces catégories et n'exerceraient qu'une influence indirecte sur le problème de *Troubleshooting*.

On suppose que l'on part d'un état d'information E_0, sous-ensemble de l'événement certain. Ce sous-ensemble doit contenir au moins l'information $o_0 \neq \text{normal}$, mais peut aussi contenir d'autres informations connues à l'état initial.

4.2 Problème de *Troubleshooting*

Avec les outils de la section précédente, on donne maintenant la définition formelle de ce que l'on veut dire par *problème de Troubleshooting* dans la suite de ce document.

Définition. Un *problème de Troubleshooting* est défini par la donnée de :

1. un ensemble fini $\mathcal{C}$ de *composants* ;
2. un sous-ensemble $\mathcal{C}_o \subset \mathcal{C}$ des composants dits *observables* ;
3. un ensemble fini $\mathcal{O}$ d'*observations globales* ;
4. un élément particulier $o_0 \in \mathcal{O}$ appelé *problem defining node* ;

5. un réseau Bayésien satisfaisant les propriétés de la Section 4.1 ;
6. un ensemble E_0, sous-ensemble de l'événement certain, donnant les informations à l'état initial ;
7. une fonction $C_r : \mathcal{C} \to \mathbb{R}_+$ donnant les *coûts de réparation* des composants ; et
8. une fonction $C_o : \mathcal{C}_o \cup \mathcal{O} \to \mathbb{R}_+$ donnant les *coûts d'observation* avec $C_o(o_0) = 0$.

On appellera *action* dans un problème de *Troubleshooting* tout ce qui permet de mettre à jour l'ensemble E d'informations disponibles sur le problème, en obtenant un nouvel ensemble E'. Comme décrit dans les Sections 3.1 et 3.5.3, les actions considérées ici sont les réparations des composants, les observations globales ou d'un composant, et les paires observation-réparation des composants observables. Dans le cas des observations, la mise-à-jour de E se fait en rajoutant le résultat de l'observation, qui est aléatoire et dépend de la loi correspondante au nœud de l'observation sur le réseau Bayésien. Dans le cas de la réparation d'un composant c_i, la mise-à-jour de E se fait en rajoutant l'information c_i = normal et en supprimant toute autre information devenue caduque suite à la réparation.

On remarque que l'ensemble des actions admissibles dépend du cadre considéré et de l'ensemble des informations E. Ainsi, le cadre décrit dans la Section 3.5.3, par exemple, n'autorise que les actions couplées d'observation-réparation pour les composants observables, ne permettant pas que les actions de réparation ou d'observation soient réalisées de façon séparée pour ces composants. En plus, si un composant a déjà été réparée, son action de réparation (ou d'observation-réparation, suivant le cadre) n'est plus disponible. Lorsqu'une observation est réalisée, sa nouvelle réalisation n'est pas disponible, jusqu'à ce qu'une action de réparation ne rende caduque l'information qui avait été apportée par cette observation. Enfin, lorsque E contient le fait que le dispositif est réparé (ce qui arrive, par exemple, après l'action d'appel au service), aucune autre action n'est possible.

4.3 Arbres de décision et stratégies

Étant donné un problème de *Troubleshooting* et un cadre définissant les actions admissibles, on peut associer à ce problème l'arbre de décision correspondant construit de façon inductive comme suit.

- La racine de cet arbre est un nœud de décision avec l'état d'information E_0, duquel partent autant d'arcs que d'actions possibles sous l'état E_0.
- Si l'action est une réparation ou une paire observation-réparation, l'arc correspondant mène à un nœud de loterie binaire déterminant si la réparation a résolu le problème du dispositif ou pas. Dans le cas où l'action résout le problème, la loterie mène sur une feuille de l'arbre. Dans le cas contraire, elle mène à un nouveau nœud de décision, associé à un état E_1 qui incorpore l'information issue de la réparation, et l'arbre se construit de façon inductive à partir de ce nœud.
- Si l'action est une observation, l'arc correspondant mène à un nouveau nœud de loterie avec la loi de probabilité correspondant au nœud d'observation correspondant du réseau Bayésien. Ce nœud de loterie a autant d'arcs sortants que de résultats possibles de l'observation. Chacun de ces arcs mène à un nouveau nœud de décision avec l'état de l'information contenant le résultat de la loterie, et l'arbre est construit de façon inductive à partir de ce nœud.

L'arbre ainsi construit est fini car le nombre d'actions possibles est fini, les actions de réparation (ou d'observation-réparation) ne peuvent être réalisées qu'une seule fois au maximum, et, même si les actions d'observation peuvent être répétées lorsque des nouvelles informations sont disponibles, il y a une borne sur le nombre maximal de fois que chaque action d'observation peut être répétée (donnée par le nombre total de composants).

À chaque feuille de l'arbre, on associe le *coût total de réparation* de cette feuille, qui correspond à la somme de tous les coûts des actions prises dans le chemin qui va de la racine à cette feuille.

Une *stratégie* S dans cet arbre est la donnée, pour chaque nœud de décision de l'arbre, d'une action à prendre dans ce nœud. L'*espérance de coût* de la stratégie S avec l'état d'informations

initial E_0, $\mathrm{EC}(E_0, S)$, est calculée de la façon suivante : à chaque feuille de l'arbre, on associe son coût total de réparation. À chaque nœud de loterie de l'arbre dont les descendants ont été calculés, on associe l'espérance de la loterie correspondante. À chaque nœud de décision de l'arbre dont les descendants ont été calculés, on associe la valeur de son descendant choisi dans la stratégie S. La valeur $\mathrm{EC}(E_0, S)$ est la valeur associée à la racine de l'arbre.

En dénotant par $\mathcal{S}$ l'ensemble de toutes les stratégies possibles, le problème de *Troubleshooting* peut être écrit, en termes de ces notations, comme le calcul de

$$\operatorname*{argmin}_{S \in \mathcal{S}} \mathrm{EC}(E_0, S). \tag{5}$$

Remarque. Étant donnée une stratégie S, lorsque l'on se place dans un nœud de décision quelconque dans lequel la stratégie S donne un choix d'action a, la valeur de $\mathrm{EC}(E_0, S)$ ne change pas si on modifie la stratégie S dans les sous-arbres issues de ce nœud de décision ne correspondant pas à l'action a. Dans la suite, on identifie deux stratégies qui peuvent être obtenues l'une à partir de l'autre avec ce type de modification.

Remarque. Dans le cas particulier où l'ensemble des actions possibles ne contient que des réparations ou des paires observation-réparation, une stratégie S peut être identifiée à une séquence d'actions $(a_1, \ldots, a_k)$, où a_1 est l'action à prendre dans le nœud de décision racine, a_2 est la décision à prendre si l'action a_1 ne résout pas le problème, et ainsi de suite. Cela arrive car, dans ce cas, les seuls nœuds de loterie possibles sont binaires, à la suite des réparations, avec un arc qui mène directement à une feuille. Ce cadre correspond à celui décrit dans la Section 3.5.2 et dans la première partie de la Section 3.5.3, et la définition de $\mathrm{EC}(E_0, S)$ donnée ici coïncide avec la définition de $\mathrm{ECR}(E_0, S)$ de (3).

4.4 Méthodes classiques d'approximation

Comme rappelé dans la Section 3.5.1, la résolution exacte de (5) est un problème difficile. Afin d'approcher ce minimum, plusieurs algorithmes basés sur des heuristiques ont été proposés, comme détaillé dans la Section 3.5. Ce projet s'intéresse à trois d'entre deux, donnés dans Heckerman et al. (1994, 1995) et décrits dans les Sections 3.5.2 et 3.5.3 :

1. L'algorithme dit *simple* de la Section 3.5.2, dans lequel les seules actions disponibles sont les réparations des différents composants.
2. L'algorithme dit *simple avec observations locales* décrit au début de la Section 3.5.3, dans lequel les seules actions disponibles sont les réparations des composants non-observables et les paires observation-réparation des composants observables.
3. L'algorithme dit *myope* décrit dans la suite de la Section 3.5.3, qui rajoute la possibilité d'observations globales.

Outre le fait que les algorithmes approchés sont plus rapides qu'une résolution exacte, ils ont également l'avantage d'être dynamiques par rapport aux résolutions exactes, c'est-à-dire ils arrivent à prendre en considération le fait que l'utilisateur peut ne pas choisir l'action conseillée par l'algorithme, ce que obligerait un nouveau calcul de l'arbre par une résolution exacte, gourmand en temps.

La description de l'implémentation des algorithmes approchés ci-dessus est donnée dans la Section 6 ci-après.

5 Contribution

Cette section décrit les contributions principales au problème du *Troubleshooting* réalisées dans ce projet. La première contribution décrite concerne le calcul approché de $\mathrm{EC}(E_0, S)$ pour une stratégie S donnée et est détaillée dans la Section 5.1. On décrit ensuite, dans la Section 5.2, les idées utilisées pour un algorithme résolvant (5) de façon exacte. Finalement, la Section 5.3

décrit une méthode permettant de traiter le cas où les coûts de réparation sont inconnus et des techniques d'élicitation sont utilisées pour déterminer la meilleure stratégie à adopter.

5.1 Calcul de l'espérance de coût par une méthode de Monte-Carlo

Étant donné une stratégie S, le calcul de $\mathrm{EC}(E_0, S)$ peut s'effectuer de façon inductive à travers le parcours de l'arbre décrit dans la Section 4.3. À une stratégie S donnée, ce parcours utilise beaucoup moins d'opérations que le calcul de la stratégie optimale pour (5), puisque, à chaque nœud de décision, il suffit de parcourir la décision prise par la stratégie S. Les branchements interviennent seulement dans les nœuds de loterie, et en plus les nœuds de loterie consécutifs à une réparation donnent lieu à des branchements triviaux, dans le sens où une des branches conduit directement à une feuille. La complexité provient uniquement des branchements des nœuds de loterie correspondants à des observations globales, qui peut être importante dans des situations où de nombreuses observations globales sont disponibles.

Même dans le cas où le calcul de $\mathrm{EC}(E_0, S)$ se fait de façon assez rapide, un inconvénient de son calcul exact est qu'il ne donne que la valeur de l'espérance du coût de réparation. Il peut aussi être intéressant de considérer le coût de réparation comme une variable aléatoire, auquel cas on peut s'intéresser à d'autres de ses caractéristiques, comme son histogramme, mais aussi à d'autres variables aléatoires associées, comme le nombre de réparations ou d'observations faites avant que le problème du dispositif ne soit résolu.

Pour cette raison, il peut être intéressant d'avoir des algorithmes de calcul empirique de $\mathrm{EC}(E_0, S)$ qui, en plus de retourner une approximation de cette espérance en soi, permettent d'obtenir facilement d'autres statistiques sur le déploiement de la stratégie S.

L'idée naturelle pour ce faire est de, étant donnée une stratégie S, la simuler un grand nombre N de fois. La simulation d'une stratégie se fait en résolvant les nœuds de décision selon la stratégie et en tirant les résultats des loteries de façon aléatoire selon la loi obtenue à travers le réseau Bayésien représentant le problème et l'état de l'information dans le nœud courant. Au fur de l'exécution, des statistiques sur chaque simulation peuvent être collectées : le coût total de réparation (de la feuille où l'algorithme s'arrête), le nombre de réparations réalisées, le nombre d'observations réalisées, et d'autres statistiques pouvant être pertinentes.

Pour que ces statistiques collectées soient significatives, il faut que le nombre N soit suffisamment grand. On propose ici le critère d'arrêt suivant. Soit K la variable aléatoire donnant le coût total de réparation pour une simulation. La suite de simulations réalisées correspond ainsi à une suite $(K_i)_{i\in\mathbb{N}}$ de variables aléatoires indépendantes et identiquement distribuées de même loi que K. Comme K est une variable aléatoire à support fini, on peut appliquer le Théorème central limite à la suite K_i, qui affirme que la suite de variables aléatoires $(Z_n)_{n\in\mathbb{N}}$ définie par

$$Z_n = \frac{\overline{K}_n - \mu}{\sigma/\sqrt{n}}, \qquad \text{avec } \overline{K}_n = \frac{1}{n}\sum_{i=1}^{n} K_i,$$

converge en loi vers une loi normale centrée réduite $\mathcal{N}(0,1)$, où μ et σ sont l'espérance et l'écart-type de K. Par définition, on a $\mu = \mathrm{EC}(E_0, S)$.

On peut alors proposer un critère d'arrêt de l'algorithme à partir de l'intervalle de confiance déduit de cette convergence. Soit Z une variable aléatoire suivant la loi $\mathcal{N}(0,1)$, fixons un niveau de confiance $\alpha \in (0,1)$ et considérons la valeur z_α telle que $\mathbb{P}(-z_\alpha < Z < z_\alpha) = \alpha$. On a alors

$$\lim_{n\to+\infty} \mathbb{P}\left(-z_\alpha < \frac{\overline{K}_n - \mu}{\sigma/\sqrt{n}} < z_\alpha\right) = \alpha.$$

L'erreur relative commise en approximant μ par $\overline{K}_n$ est $\frac{\overline{K}_n - \mu}{\overline{K}_n}$. En manipulant l'expression ci-dessous afin de l'écrire en termes de cette quantité, on trouve

$$\lim_{n\to+\infty} \mathbb{P}\left(-\frac{z_\alpha \sigma}{\overline{K}_n\sqrt{n}} < \frac{\overline{K}_n - \mu}{\overline{K}_n} < \frac{z_\alpha \sigma}{\overline{K}_n\sqrt{n}}\right) = \alpha.$$

Ainsi, on peut garantir que l'erreur relative asymptotique est plus petite qu'une valeur $\varepsilon > 0$ donnée, avec un niveau de confiance de α, dès lors que l'on a l'inégalité

$$\frac{z_\alpha \sigma}{\overline{K}_n \sqrt{n}} < \varepsilon.$$

Cette inégalité peut ainsi être utilisée comme critère d'arrêt de l'algorithme à un niveau de confiance α et une tolérance relative ε données. Cependant, elle n'est pas encore pratique sous cette forme, puisque l'écart-type σ de K n'est pas connu. Comme usuel en statistiques, on propose ici le remplacement de σ par l'écart-type empirique s_n de l'échantillon $K_1, \ldots, K_n$. Pour garantir que s_n est suffisamment proche de σ, on impose aussi à ce que n soit assez grand, c'est-à-dire qu'il soit supérieur à un certain nombre minimal d'échantillons n_0 donné. Cela conduit au critère d'arrêt

$$\frac{z_\alpha s_n}{\overline{K}_n \sqrt{n}} < \varepsilon \qquad \text{et} \qquad n \geq n_0. \tag{6}$$

5.2 Résolution exacte

Comme nous avons indiqué dans la Section 3.5, l'algorithme qui permet de résoudre le problème de *Troubleshooting* de manière exacte est intraitable même pour des tailles de données qui ne sont pas très grandes. C'est pour cette raison qu'en pratique on utilise plutôt des heuristiques afin de le résoudre. En revanche, les algorithmes exactes restent utiles puisqu'en les utilisant on a la capacité de tester des techniques approchées. Plus spécifiquement, pour des problèmes suffisamment petits, on peut appliquer une recherche exhaustive qui nous donnerait avec certitude une stratégie optimale pour la minimisation du coût espéré de réparation (5). Ensuite, on peut calculer une solution approchée en utilisant l'heuristique à tester et étudier comment cette heuristique fait face au problème proposé : plus on s'approche d'une solution optimale, meilleure est l'heuristique.

Pour cette raison, nous avons conçu deux algorithmes de recherche par force brute qui permettent de résoudre un problème de *Troubleshooting* de façon exacte. Le premier d'entre eux consiste à dénombrer toutes les stratégies possibles que l'on peut construire étant données les actions admissibles (des réparations, des observations et des couples « observation-réparation »). Le deuxième algorithme utilise le principe de programmation dynamique : un arbre de stratégie optimal ne doit contenir que des sous-arbres également optimaux.

5.2.1 Dénombrement complet

Soient $\mathcal{C}$, $\mathcal{C}_o$ et $\mathcal{O}$ les ensembles de la définition de la Section 4.2. L'idée de l'algorithme par dénombrement est d'évaluer tous les arbres de stratégie uniques que l'on peut construire à partir de $\mathcal{C}$, $\mathcal{C}_o$ et $\mathcal{O}$. Par exemple, étant donné $\mathcal{C} = \{A_1, A_2\}$, $\mathcal{C}_o = \varnothing$ et $\mathcal{O} = \{O\}$ avec O une observation pouvant prendre les valeurs o_1 et o_2, on peut construire exactement 8 arbres de stratégies dont les représentations graphiques sont données dans la Figure 3. Bien que pour le cas où $\mathcal{O} = \varnothing$ l'on puisse assez simplement dénombrer toutes les stratégies admissibles, car il suffit de considérer toutes les permutations de $\mathcal{C}$, si $\mathcal{O} \neq \varnothing$ le problème de dénombrement devient beaucoup plus compliqué, puisque chaque conséquence d'une observation globale produit dans l'arbre de stratégie une branche dévouée partant du nœud correspondant à cette observation. Ainsi, on doit remplir chacune de ces branches de toutes les manières possibles utilisant les actions pas encore ajoutés dans l'arbre, parmi lesquels on peut avoir encore d'autres observations dont les conséquences produiront encore plus de branches. Alors, nous avons eu besoin de mettre en place un algorithme assez spécifique afin de résoudre ce problème de dénombrement.

Remarque. Par la suite, pour représenter une stratégie sous forme d'arbre, on n'utilisera pas un arbre de décision complet mais plutôt une version simplifiée qui améliore la lisibilité de la stratégie. Puisqu'une stratégie consiste à donner un unique choix d'action dans chaque nœud de décision, on regroupe le nœud de décision et le nœud de loterie correspondant à l'action choisie dans

un seul nœud (et on ne représente pas les autres actions possibles du nœud de décision, celles non choisies par la stratégie). De plus, pour une action de réparation, la loterie correspondante n'a que deux résultats possibles, et on décide de ne pas afficher la branche triviale correspondant au cas où le problème est résolu. La Figure 3 donne des exemples d'une telle représentation.

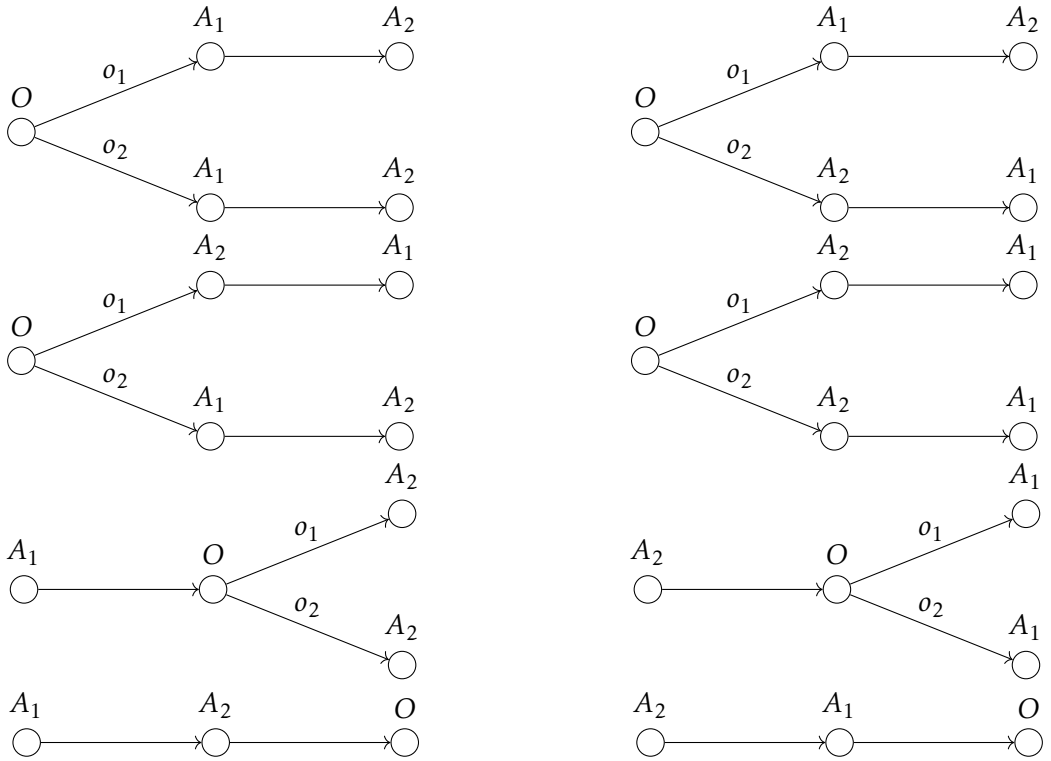

FIGURE 3 – Tous les arbres de stratégies admissible que l'on peut construire à partir de $\mathcal{C} = \{A_1, A_2\}$, $\mathcal{C}_o = \varnothing$ et $\mathcal{O} = \{O\}$ une observation pouvant prendre les valeurs o_1 et o_2.

Pour dénombrer tous les arbres de stratégie possibles, nous proposons de commencer par chaque action admissible $n \in \mathcal{O} \cup \mathcal{C}$ et de relancer de manière récursive le dénombrement excluant n de l'ensemble des actions possibles. Si n est une observation globale, on sauvegarde ses conséquences et son identifiant dans un arbre ainsi que les actions qui étaient admissibles à ce moment-là. Ensuite, on continue à dénombrer en remplissant une des branches qui part de n.

Sinon, on n'a qu'une seule branche à remplir puisqu'une action de réparation ou d'observation-réparation n'a qu'une seule conséquence à considérer : on a effectué l'action n donc le composant correspondant dès maintenant marche avec certitude, mais le dispositif ne marche pas encore (si le dispositif marche alors on a fini le processus de réparation). Dans ce cas-là, on continue le dénombrement en excluant n, mais sans sauvegarder sa position et ses conséquences dans l'arbre courant.

Dès que l'on obtient en entrée d'appel récursif un ensemble d'actions réalisables vide, on considère que la branche courante est remplie. Il faut alors remonter dans l'arbre en cherchant l'observation la plus récente qui a une conséquence dont la branche n'a pas encore été remplie. S'il y en existe au moins une, on relance ce processus récursif pour cette branche. Sinon, on a rempli toutes les branches de toutes les conséquences de toutes les observations, et la construction de l'arbre est donc finie.

Une fois la construction de l'arbre finie, on calcule son espérance de coût. Comme on cherche le meilleur arbre, on compare ce nombre avec la meilleure espérance de coût déjà trouvée afin de savoir quel arbre garder. Cela conclut un appel donné du dénombrement. L'algorithme proposé se ressemble au parcours en profondeur.

Par ailleurs, pour évaluer une stratégie $S \in \mathcal{S}$ déjà trouvée, on utilise comme critère son espé-

rance de coût, que l'on calcule avec la formule récursive

$$\mathrm{EC}(E,S) = P(o_0 \neq \text{normal} \mid E) \cdot \Big[\text{coût}(\text{racine}(S)) + \sum_r \mathrm{EC}(E \cup \{\text{racine}(S) = r, o_0 \neq \text{normal toujours}\}, \text{sous-arbre}(S, \{\text{racine}(S) = r\}))\Big], \quad (7)$$

où E est l'ensemble d'évidences dont on dispose au moment d'exécuter l'action racine(S). Initialement on pose $E_0 = \{o_0 \neq \text{normal au début}\}$. L'espérance de coût donnée par (7), qui coïncide avec la définition donnée dans la Section 4.3, est la fonction objectif que l'on cherche à minimiser. Par conséquent, l'algorithme de dénombrement complet nous retourne une stratégie S^* telle que

$$\mathrm{EC}(E_0, S^*) = \min_{S \in \mathcal{S}} \mathrm{EC}(E_0, S). \quad (8)$$

Remarque. Les évènements $\{o_0 \neq \text{normal}\}$, $\{\text{racine}(S) = r, o_0 \neq \text{normal toujours}\}$ et $\{o_0 \neq \text{normal au début}\}$ sont tous les trois différents car le résultat d'une observation o_0 dépend de l'instant où on l'exécute. En effet, $o_0 \neq \text{normal}$ dénote le fait que, si on fait une telle observation en ce moment, on obtiendrait un résultat qui n'est pas normal. En même temps, la notation $\{\text{racine}(S) = r, o_0 \neq \text{normal toujours}\}$ veut dire que le système ne marchait pas avant, et qu'il ne marche toujours pas même si racine(S) = r. Enfin, $\{o_0 \neq \text{normal au début}\}$ décrit le fait que le système ne marchait pas au tout début, avant que l'on ait commencé le processus de réparation. Il est alors assez évident que le calcul des probabilités correspondantes à ces événements ne serait pas très facile à effectuer en termes du modèle utilisé. C'est pourquoi on utilise la « *single fault assumption* » introduite dans la Section 3.5.2 et décrite dans la suite dans la Section 6.1.

Pour résumer on fournit l'algorithme en pseudo-code d'une fonction *Évaluer* qui réalise l'approche décrite ci-dessus donc qui dénombre tous les arbres de stratégie possibles :

5.2.2 Programmation dynamique

Le problème principal de l'algorithme de dénombrement complet de la Section 5.2.1 est qu'il devient intraitable trop vite. Même pour des problèmes assez petits, par exemple lorsque $|\mathcal{O}| = 4$, $|\mathcal{C}| = 3$ et $|\mathcal{C}_o| = 0$, dénombrer tous les arbres possibles prend des heures. Cependant, il existe une possibilité d'améliorer l'algorithme de recherche exhaustive en utilisant le principe de programmation dynamique. Observant encore une fois la formule (7), on constate que le coût espéré d'un arbre quelconque dépend linéairement des coûts espérés de ses sous-arbres avec des coefficients positifs ($P(o_0 \neq \text{normal} \mid E) \in [0,1]$). De plus, on remplit les branches d'un arbre de stratégie de manière indépendante : le sous-arbre construit pour l'une des branches n'influence pas les sous-arbres que l'on construit pour les autres branches. C'est pour cette raison que, pour minimiser l'espérance de coût d'une stratégie, il suffit de minimiser l'espérance de coût des sous-stratégies correspondantes.

Ainsi, l'idée principale de cette technique est de parcourir toutes les actions admissibles afin de connecter à chacune un sous-arbre optimal construit à partir des actions restantes. Puis, pour trouver un sous-arbre optimal, on relance récursivement la même méthode, c'est-à-dire, on parcourt toutes les actions restantes et on connecte à chacune d'entre elles un « sous-sous-arbre » encore optimal. On continue à effectuer ces appels récursifs jusqu'à ce que l'on n'ait plus d'actions à réaliser. À ce moment-là, on pose que toutes les espérances des coûts des sous-arbres dans la formule (7) sont égales à 0 et on retourne la stratégie optimale triviale de cet appel, ce qui nous permet de trouver une stratégie optimale de l'appel précédent, et ainsi de suite, ce qui nous permettra à la fin de construire un arbre optimal.

Par conséquent, utilisant les idées indiquées ci-dessus on formule l'algorithme proposé en pseudo-code (de la fonction *DynProg*, plus précisément) :

Algorithme 1 : *Évaluer*

Données : $\mathcal{N} = \mathcal{C} \cup \mathcal{O}$, l'ensemble de noeuds admissibles.

args, les paramètres complémentaires ; rien par défaut.

S^*, une variable globale que l'on peut utiliser dans tous les appels indépendants

de cet algorithme et qui représente la meilleure solution trouvée pour l'instant.
Résultat : S^*, le meilleur arbre de stratégie trouvé avec l'espérance de coût minimale.

```
// Parcours par tous les noeuds admissibles (s'il en existe au moins un)
```
pour $n \in \mathcal{N}$ **faire**
  si $args = \emptyset$ **alors**
    `// Le cas où l'on a appelé cette fonction la première fois`
    Évaluer$(\mathcal{N} \setminus \{n\}, CreateArgs(n))$
  sinon
    `// L'on connecte un noeud courant avec son parent qui est stocké avec la branche à remplir dans args`
    Connecter$(n, args)$
    `// L'on continue en remplissant des enfants d'un noeud n si c'est bien possible`
    Évaluer$(\mathcal{N} \setminus \{n\}, ActualiserArgs(n, args))$

si $|\mathcal{N}| = 0$ **alors**
  si *Il nous reste des branches à remplir* **alors**
    `// Danc ce cas-là, l'on doit se retourner vers une dernière observation dont une branche l'on n'a pas encore rempli`
    Évaluer$(NoeudsAdmissiblesPrecedents(\text{args}), ActualiserArgs(n, args))$
  sinon
    `// L'on a déjà rempli toutes les branches possibles alors l'on a fini la construction d'un arbre ; il faut alors calculer son espérance du coût`
    `// L'on suppose que l'on stocke un arbre courant dans args et que l'on peut l'obtenir en appelant une fonction dévouée`
    $ec \longleftarrow EC(\emptyset, ObtenirArbreCourant(args))$
    $S^* \longleftarrow$ **si** $ec < EC(\emptyset, S^*)$ **alors** $ObtenirArbreCourant(args)$ **sinon** S^*

5.3 Élicitation des coûts

Jusqu'à présent, on a considéré que le problème de Troubleshooting satisfait les hypothèses de la définition donnée dans la Section 4.2. En particulier, les propriétés 5, 7 et 8 impliquent que plusieurs paramètres définissant le problème — tables de probabilité du réseau Bayésien et coûts d'observation et de réparation de chaque composant — sont fixes et connus. Dans cette partie, on considère le cas où il y a une incertitude sur les coûts, en supposant que l'on peut ne pas connaître leur valeur exacte mais uniquement une loi de probabilité sur chaque coût. Pour simplifier, on considère que seuls les coûts de réparation peuvent être incertains, mais que les coûts d'observation sont connus. Cette hypothèse simplificatrice facilite l'exposition de la méthode et son implémentation mais on peut facilement adapter ces méthodes au cas de coûts d'observation incertains.

Plus précisément, on considère que l'hypothèse 7 de la Section 4.2 est remplacée par l'hypothèse

7′. une fonction $C_r : \mathcal{C} \to \mathcal{P}(\mathbb{R}_+)$ donnant, pour chaque composant, la *loi de probabilité de son*

Algorithme 2 : Résolution par la programmation dynamique

Données : $\mathcal{N} = \mathcal{C} \cup \mathcal{O}$, l'ensemble de noeuds admissibles.

p, la probabilité que le système ne marchait pas après avoir fait une action

précédente ; 1 par défaut.

E, un ensemble des évidences ; ∅ par défaut.

Résultat : S^*, le meilleur arbre de stratégie trouvé avec l'espérance de coût minimale.

```
// S'il n'y a pas de noeuds admissibles alors on ne retourne rien
```

si $|\mathcal{N}| = 0$ **alors**

  retourner $\emptyset, 0$

```
// Une initialisation par un arbre qui ne contient qu'un noeud de service
```

$S^* \longleftarrow \{ServiceNode \longrightarrow \emptyset\}$

```
// Parcours par tous les noeuds admissibles (s'il en existe au moins un)
```

pour $n \in \mathcal{N}$ **faire**

  $ec \longleftarrow p \cdot \text{coût}(n)$

  `// Une initialisation de l'arbre courant` S_c `qui a` n `pour sa racine`

  $S_c \longleftarrow \{n \longrightarrow \emptyset\}$

  pour tous les résultats possibles r d'exécution d'action qui correspond à n **faire**

    $pn \longleftarrow P(o_0 \neq \text{normal} \mid E \cup \{n = r, o_0 \neq \text{normal toujours}\})$

    $ec \longleftarrow ec + p \cdot connect(S_c, n, DynProg(\mathcal{N} \setminus \{n\}, pn, E \cup \{n = r, o_0 \neq \text{normal toujours}\}))$

  si $ec < EC(E, S^*)$ **alors**

    $S^* \longleftarrow S_c$

retourner $S^*, EC(E, S^*)$

coût de réparation,

où $\mathcal{P}(\mathbb{R}_+)$ dénote l'ensemble des lois de probabilité sur $\mathbb{R}_+$.

Afin de simplifier, on ne travaille dans la suite qu'avec des lois de probabilité uniformes sur des intervalles bornés de $\mathbb{R}_+$. Ainsi, la donnée de $C_r : \mathcal{C} \to \mathcal{P}(\mathbb{R}_+)$ équivaut à la donnée des fonctions $C_{r,\min} : \mathcal{C} \to \mathbb{R}_+$ et $C_{r,\max} : \mathcal{C} \to \mathbb{R}_+$ donnant les extrémités gauche et droite des supports des lois uniformes. On demande ainsi à ce que $C_{r,\min}(c) \leq C_{r,\max}(c)$ pour tout $c \in \mathcal{C}$, et on autorise le cas $C_{r,\min}(c) = C_{r,\max}(c)$ afin de pouvoir représenter le cas où le coût exact de réparation de c est connu. On dénote par $\overline{C}_r : \mathcal{C} \to \mathbb{R}_+$ la fonction qui, à chaque $c \in \mathcal{C}$, associe l'espérance de la variable aléatoire $C_r(c)$, soit

$$\overline{C}_r(c) = \frac{C_{r,\min}(c) + C_{r,\max}(c)}{2}.$$

Une première heuristique pour traiter ce cas correspond à appliquer les algorithmes de *Troubleshooting* décrits précédemment en utilisant $\overline{C}_r$ comme la fonction de coût de l'hypothèse 7 de la Section 4.2, c'est-à-dire, en considérant que les coûts de réparation sont déterministes et égaux aux espérances des lois fournies de coûts de réparations. L'objectif de cette section est de présenter une autre heuristique, basée sur l'élicitation, permettant de prendre en compte le caractère aléatoire des coûts au-delà de la simple espérance. On cherche ainsi à utiliser l'élicitation afin d'améliorer les connaissances sur les coûts de réparation, c'est-à-dire diminuer la taille des intervalles de valeur pour chaque coût. Pour cela, on utilise la notion de valeur d'information décrite dans la Section 3.3.

On remarque d'abord que le coût espéré $\text{EC}(E_0, S)$ introduit dans la Section 4.3 dépend, entre autres choses, de la fonction C_r donnant le coût (déterministe) de réparation de chaque composant. Afin de rendre cette dépendance plus explicite et simplifier l'exposition de la suite, on note cette valeur dans cette section par $\text{EC}(E_0, S, C_r)$.

L'objectif est de déterminer quelle question poser en premier à l'utilisateur sur les coûts de réparation pour acquérir le plus d'information possible. Dans la formulation proposée ici, on choisit de regarder une question par composant de $\mathcal{C} = \{c_1, \ldots, c_n\}$, de la forme « Est-ce que le coût de réparation du composant c_i est plus petit que α ? », où α est une valeur dans $[C_{r,\min}(c_i), C_{r,\max}(c_i)]$. Pour déterminer l'information apportée par la réponse de cette question, on définit d'abord les fonctions de coûts espérés $C^-_{r,i} : \mathcal{C} \to \mathbb{R}_+$ et $C^+_{r,i} : \mathcal{C} \to \mathbb{R}_+$ sous une réponse affirmative ou négative à la question, données respectivement par

$$C^-_{r,i}(c) = \begin{cases} \overline{C}_r(c) & \text{si } c \neq c_i, \\ \dfrac{C_{r,\min}(c_i) + \alpha}{2} & \text{si } c = c_i, \end{cases} \qquad C^+_{r,i}(c) = \begin{cases} \overline{C}_r(c) & \text{si } c \neq c_i, \\ \dfrac{\alpha + C_{r,\max}(c_i)}{2} & \text{si } c = c_i, \end{cases} \tag{9}$$

c'est-à-dire que les coûts de réparation de toutes les autres composants restent inchangés mais le coût de réparation de c_i devient l'espérance $C^-_{r,i}(c_i) = \frac{C_{r,\min}(c_i)+\alpha}{2}$ de la loi uniforme sur $[C_{r,\min}(c_i), \alpha]$ si la réponse à la question est affirmative ou l'espérance $C^+_{r,i}(c_i) = \frac{\alpha+C_{r,\max}(c_i)}{2}$ de la loi uniforme sur $[\alpha, C_{r,\max}(c_i)]$ si la réponse à la question est négative.

On peut alors considérer, étant donné un état initial d'information E, un composant $c_i \in \mathcal{C}$, et une valeur $\alpha \in [C_{r,\min}(c_i), C_{r,\max}(c_i)]$, la valeur espérée de l'information $\mathrm{EVOI}(E, c_i, \alpha)$ apportée par la réponse à la question précédente, définie, selon le principe de la formule (1), par

$$\mathrm{EVOI}(E, c_i, \alpha) = \mathrm{EC}(E, S^*, \overline{C}_r) - \Big[\mathrm{EC}(E, S^*_-, C^-_{r,i}) P(C(r_i) < \alpha \mid E) + \mathrm{EC}(E, S^*_+, C^+_{r,i}) P(C_r(c_i) \geq \alpha \mid E)\Big], \tag{10}$$

où S^*, S^*_- et S^*_+ sont les stratégies optimales pour les coûts $\overline{C}_r$, $C^-_{r,i}$ et $C^+_{r,i}$, respectivement, c'est-à-dire,

$$S^* = \operatorname*{argmin}_{S \in \mathcal{S}} \mathrm{EC}(E, S, \overline{C}_r), \quad S^*_- = \operatorname*{argmin}_{S \in \mathcal{S}} \mathrm{EC}(E, S, C^-_{r,i}), \quad S^*_+ = \operatorname*{argmin}_{S \in \mathcal{S}} \mathrm{EC}(E, S, C^+_{r,i}). \tag{11}$$

Par rapport à (1), la formule ci-dessus a un changement de signe, puisque ici il s'agit d'un problème de minimisation de coûts, alors que (1) a été présentée pour la maximisation d'une utilité.

On a la propriété suivante de la valeur espérée de l'information :

Proposition. *Pour tout état initial d'information E, tout composant $c_i \in \mathcal{C}$ et tout $\alpha \in [C_{r,\min}(c_i), C_{r,\max}(c_i)]$, on a* $\mathrm{EVOI}(E, c_i, \alpha) \geq 0$.

Démonstration. On remarque d'abord que, étant donnée une fonction de coût quelconque $\mathfrak{C} : \mathcal{C} \to \mathbb{R}_+$, d'après la définition de $\mathrm{EC}(E, S, \mathfrak{C})$ donnée dans la Section 4.3, pour une stratégie S *donnée*, $\mathrm{EC}(E, S, \mathfrak{C})$ est une combinaison *linéaire* des coûts $\mathfrak{C}(c)$ pour $c \in \mathcal{C}$ et des coûts d'observation, avec coefficients qui ne dépendent que de la stratégie S et des probabilités du réseau Bayésien. Alors, quitte à regrouper toutes les occurrences du coût de réparation de c_i dans les formules de EC, il existe des constantes C_0 et p_0 ne dépendant que de S, des probabilités du réseau Bayésien, des coûts d'observation et des coûts de réparation $\overline{C}_r(c)$ pour $c \neq c_i$ telles que

$$\begin{aligned} \mathrm{EC}(E, S, \overline{C}_r) &= C_0 + p_0 \overline{C}_r(c_i) \\ \mathrm{EC}(E, S, C^-_{r,i}) &= C_0 + p_0 C^-_{r,i}(c_i) \\ \mathrm{EC}(E, S, C^+_{r,i}) &= C_0 + p_0 C^+_{r,i}(c_i) \end{aligned} \tag{12}$$

Soit S^* donné comme dans (11). On montre d'abord que

$$\mathrm{EC}(E, S^*, \overline{C}_r) = \mathrm{EC}(E, S^*, C^-_{r,i}) P(C(r_i) < \alpha \mid E) + \mathrm{EC}(E, S^*, C^+_{r,i}) P(C_r(c_i) \geq \alpha \mid E). \tag{13}$$

En effet, on a, en utilisant (9) et (12),

$$\begin{aligned} &\mathrm{EC}(E, S^*, C^-_{r,i}) P(C(r_i) < \alpha \mid E) + \mathrm{EC}(E, S^*, C^+_{r,i}) P(C_r(c_i) \geq \alpha \mid E) \\ &= \Big(C_0 + p_0 C^-_{r,i}(c_i)\Big) P(C(r_i) < \alpha \mid E) + \Big(C_0 + p_0 C^+_{r,i}(c_i)\Big) P(C_r(c_i) \geq \alpha \mid E) \end{aligned}$$

$$
\begin{aligned}
&= C_0 + p_0\Big(C^-_{r,i}(c_i)P(C(r_i) < \alpha \mid E) + C^+_{r,i}(c_i)P(C_r(c_i) \geq \alpha \mid E)\Big) \\
&= C_0 + p_0\left(\frac{C_{r,\min}(c_i)+\alpha}{2}\frac{\alpha - C_{r,\min}(c_i)}{C_{r,\max}(c_i)-C_{r,\min}(c_i)} + \frac{\alpha + C_{r,\max}(c_i)}{2}\frac{C_{r,\max}(c_i)-\alpha}{C_{r,\max}(c_i)-C_{r,\min}(c_i)}\right) \\
&= C_0 + p_0\frac{C_{r,\max}(c_i)^2 - C_{r,\min}(c_i)^2}{2(C_{r,\max}(c_i)-C_{r,\min}(c_i))} \\
&= C_0 + p_0\frac{C_{r,\max}(c_i)+C_{r,\min}(c_i)}{2} = C_0 + p_0\overline{C}_r(c_i) = \mathrm{EC}(E, S^*, \overline{C}_r).
\end{aligned}
$$

D'autre part, d'après les définitions de S^*_- et S^*_+ de (11), on a

$$
\mathrm{EC}(E, S^*_-, C^-_{r,i}) \leq \mathrm{EC}(E, S^*, C^-_{r,i}), \quad \mathrm{EC}(E, S^*_+, C^+_{r,i}) \leq \mathrm{EC}(E, S^*, C^+_{r,i}). \tag{14}
$$

En combinant (13) avec (14), on en déduit que

$$
\mathrm{EC}(E, S^*, \overline{C}_r) \geq \mathrm{EC}(E, S^*_-, C^-_{r,i})P(C(r_i) < \alpha \mid E) + \mathrm{EC}(E, S^*_+, C^+_{r,i})P(C_r(c_i) \geq \alpha \mid E),
$$

et alors la définition (10) de $\mathrm{EVOI}(E, c_i, \alpha)$ permet de conclure que $\mathrm{EVOI}(E, c_i, \alpha) \geq 0$. □

Afin de déterminer quelle est la meilleure question à poser à partir d'un état d'information E donné, on peut calculer les valeurs de $\mathrm{EVOI}(E, c_i, \alpha_i)$ pour tous les composants c_i encore réparables dans l'état E, avec un α_i choisi dans $[C_{r,\min}(c_i), C_{r,\max}(c_i)]$. Avec la propriété montrée dans la proposition et suivant le principe de la Section 3.3, on choisit ainsi de poser la question donnant la plus grande valeur de $\mathrm{EVOI}(E, c_i, \alpha_i)$. Si toutes ces EVOI sont égales à 0, il n'y a aucune question intéressante à poser.

Il reste encore la question de quel α_i choisir pour chaque composant c_i. Idéalement, il faudrait trouver, pour chaque composant c_i, la valeur α_i qui maximise $\mathrm{EVOI}(E, c_i, \alpha_i)$. Cependant, il y a une infinité de valeurs possibles de α_i et le problème de maximiser $\mathrm{EVOI}(E, c_i, \alpha_i)$ est alors un problème difficile. Afin de simplifier, un choix heuristique possible pour α_i est $\alpha_i = \overline{C}_r(c_i)$, ce qui est un choix raisonnable en absence d'autres informations car $\overline{C}_r(c_i)$ est l'espérance de la loi uniforme $C_r(c_i)$.

Le calcul d'EVOI nécessite la résolution de trois problèmes de minimisation de EC. La minimisation d'EC est l'objet d'étude de ce projet et, comme indiqué dans la Section 3.5.1, sa résolution exacte est un problème difficile qui motive toutes les techniques considérées dans ce projet. Ainsi, il ne serait pas faisable, dans des implémentations pratiques, de calculer EVOI de façon exacte, et des approximations se font nécessaires. On remarque que, suivant l'algorithme d'approximation utilisé, la valeur d'EVOI trouvée peut être négative. Dans ce cas, la meilleure heuristique semble d'ignorer ces valeurs négatives, les traitant comme des zéros, et ne considérer dans le choix de la question que les composants dont le calcul d'EVOI a donné des valeurs positives.

6 Implémentation

Cette section décrit les implémentations réalisées dans le cadre de ce projet, qui ont été faites en Python. La description présentée ici est synthétique mais une documentation détaillée, produite avec Sphynx, est fournie avec le code (voir annexe).

Dans un premier temps, on décrit dans la Section 6.1 la représentation choisie pour le problème de *Troubleshooting*. Les Sections 6.2, 6.3 et 6.4 décrivent l'implémentation des trois algorithmes classiques énumérés précédemment dans la Section 4.4. Ensuite, on décrit, dans les Sections 6.5, 6.6 et 6.7 les implémentations de nos contributions présentées dans la Section 5. Finalement, la Section 6.8 détaille l'implémentation de l'interface graphique

6.1 Représentation du problème de *Troubleshooting*

Nous avons décidé de représenter un problème de *Troubleshooting* à travers une classe principale `TroubleShootingProblem`, qui doit contenir toutes les informations importantes pour un problème de *Troubleshooting* selon les spécifications de la Section 4.2.

La représentation du réseau Bayésien d'un problème de *Troubleshooting* se fait à l'aide du module `pyAgrum` de Python. Dans tout le formalisme précédent, le réseau Bayésien doit être connu, et nous supposons ainsi que le réseau Bayésien doit être fourni à la classe principale `TroubleShootingProblem` sous le format d'un réseau du type `pyAgrum.BayesNet`, qui peut par exemple avoir été préalablement chargé à partir d'un fichier dans le format `.bif`.

L'obtention des réseaux Bayésiens représentant les problèmes de *Troubleshooting* n'est pas évidente car la plupart des réseaux existants ne sont pas dans le domaine public. Même si notre implémentation permet, bien sûr, l'utilisation de n'importe quel réseau Bayésien comme décrit dans la Section 4.1, nous avons fixé, dans tous les tests, un réseau Bayésien classique représentant le problème de démarrage de la voiture, représenté sur la Figure 10 dans l'Annexe A. Cela est motivé par le fait que ce réseau n'est pas trop grand pour que l'on soit capable de comparer les méthodes d'approximation implémentées avec une résolution exacte, mais il a déjà assez de nœuds pour être non-trivial.

Dans les calculs de probabilités qu'il faut faire pour implémenter les algorithmes qui nous intéressent dans ce projet, les probabilités à calculer sont le plus souvent conditionnelles aux informations acquises. Afin de représenter les informations acquises et calcules les probabilités nécessaires de façon simple, nous utilisons la classe d'inférence exacte `LazyPropagation` de `pyAgrum`. Cette classe permet d'ajouter ou supprimer facilement des évidences au réseau Bayésien et obtenir facilement les lois de probabilité conditionnelles à ces évidences.

Cependant, la classe `LazyPropagation` ne permet pas de représenter les informations d'ordre *temporelle*. En effet, assez souvent, les algorithmes demandent le calcul d'une probabilité de la forme « quelle est la probabilité que le dispositif marche étant donné que l'on a fait telles et telles réparations et que le dispositif ne marchait pas avant ? » Ce type de question demande de considérer l'état du dispositif à deux instants différents, alors qu'il n'est représenté que par un seul nœud dans le réseau Bayésien. Il faudrait dans ce cas utiliser des réseaux Bayésiens qui prennent en compte l'aspect temporel (*Dynamic Bayesian Networks*), ce qui est possible mais a pour conséquence d'augmenter la complexité à la fois d'implémentation et de calcul.

Afin de simplifier, nous utilisons alors une approximation proposée dans Heckerman et al. (1994, 1995). Lorsque le dispositif n'a qu'un seul composant en panne (on parle alors de *single fault assumption*), la probabilité que le dispositif marche suite à la réparation d'un composant c_i est égale à la probabilité que ce composant était en panne avant sa réparation, soit

$$P(o_0 = \text{normal} \mid c_i = \text{normal}, E) = P(c_i \neq \text{normal} \mid E).$$

Selon Heckerman et al. (1994, 1995), cette égalité, qui n'est plus valide sans la *single fault assumption*, reste encore une bonne approximation pour la plupart de problèmes de *Troubleshooting* d'intérêt, car il est rare en pratique qu'un dispositif tombe en panne à cause de plusieurs composants simultanément fautifs. Cette approximation a donc été utilisée dans toute l'implémentation présentée ici.

Outre le réseau Bayésien, la classe `TroubleShootingProblem` doit prendre en argument aussi les autres informations concernant le problème de *Troubleshooting* : quels nœuds du réseau Bayésien correspondent à des observations globales, à des composants observables, à des composants non-observables, au *problem defining node* et à l'appel au service. En plus, il faut fournir également les coûts d'observation et de réparation nécessaires (ou les intervalles de coûts de réparation dans le cadre de la Section 5.3). Ces informations sont fournies par des dictionnaires passés en arguments lors de l'appel au constructeur de la classe `TroubleShootingProblem`.

La description de la Section 4.2 demande aussi la donnée de l'état initial des informations, E_0. Lors de la création d'une instance de la classe `TroubleShootingProblem`, nous supposons que E_0 se réduit à la seule information que le dispositif ne marche pas à l'état initial. Il est possible de rajouter ou supprimer des informations manuellement à l'aide des méthodes `add_evidence` et `remove_evidence` avant d'appeler les algorithmes de résolution pour modifier le contenu de E_0.

6.2 Algorithme simple

Une première implémentation réalisée est celle de l'algorithme donné dans la Section 3.5.2, qui a été codé dans la méthode `simple_solver`. Cet algorithme polynomial résout le problème de *Troubleshooting* de façon exacte sous les hypothèses simplificatrices de la Section 3.5.2 et peut être vu comme une première heuristique simple pour trouver une solution approchée de (5). Seules les actions de réparation des composants sont prises en compte. Comme rappelé dans une remarque dans la Section 4.3, dans ce cadre simplifié une stratégie peut être vue comme une séquence d'actions. L'algorithme retourne alors cette séquence S, ainsi que l'espérance de coût $\mathrm{EC}(E_0, S)$.

6.3 Algorithme simple avec observations locales

La deuxième implémentation réalisée est celle décrite au début de la Section 3.5.3, qui correspond à une amélioration de l'algorithme simple et a été codée dans la méthode `simple_solver_obs`. Elle suppose que les seules actions possibles sont la réparation de composants non-observables ou la paire observation-réparation des composants observables. Comme pour l'implémentation précédente, une stratégie se réduit à une séquence d'actions et l'algorithme retourne la séquence calculée S et l'espérance de coût correspondante $\mathrm{EC}(E_0, S)$.

6.4 Algorithme myope

L'algorithme myope décrit dans la Section 3.5.3 a été codé dans la méthode `myopic_solver`. Par rapport aux deux algorithmes simples, l'algorithme myope rajoute la possibilité de réaliser des observations globales. Dans ce cas, la stratégie calculée S n'est plus une simple séquence d'actions, car, à chaque fois qu'une observation est réalisée, la suite de l'algorithme dépend du résultat de cette observation et ne peut donc pas être connue au préalable.

On a ainsi fait le choix d'implémenter `myopic_solver` de façon à faire *un pas* de la résolution du problème de *Troubleshooting* à chaque appel. L'idée est ainsi qu'un utilisateur doit faire appel à `myopic_solver` pour savoir quelle est la prochaine action recommandée, cependant, l'action n'est pas réalisée : c'est à l'utilisateur de la faire (ou décider d'en faire une autre s'il le souhaite) à travers les autres méthodes fournies et ensuite appeler à nouveau la fonction s'il souhaite connaître le prochain pas à réaliser. Une interface interactive en mode texte a été implémentée dans la méthode `myopic_wraper`.

Une subtilité de `myopic_solver` est qu'il faut parfois remettre en cause les résultats des observations. Si une observation a été réalisée et ensuite on réalise une réparation d'un composant qui peut affecter le résultat de cette observation, le résultat précédent n'est plus valide et il faut l'effacer. On a ainsi codé une méthode `observation_obsolete` qui, étant donné un nœud qui vient d'être réparé, calcule tous les nœuds d'observation qui sont impactés par cette réparation. Cette fonction est utilisée par `myopic_wraper` (et par l'interface graphique décrite dans la Section 6.8) pour effacer ainsi les informations d'observations devenues obsolètes suite à une réparation.

La fonction `myopic_solver` calcule, dans un premier temps, le coût espéré de réparation ECR sans la réalisation d'observations globales à l'aide de la méthode `simple_solver_obs`. Ensuite, pour chaque observation globale possible, la méthode calcule la valeur du coût espéré de réparation ECO si on réalise cette observation et puis applique `simple_solver_obs` pour trouver la suite des actions. La méthode choisit alors, parmi l'action proposée par le premier appel de `simple_solver_obs` et les observations globales, celle de plus petite espérance de coût.

Un autre point de vue possible et très similaire à l'approche myope est de considérer que, plutôt que de comparer chaque observation globale avec l'action retournée par `simple_solver_obs`, on peut comparer *toutes* les actions possibles entre elles. Pour chaque action, on l'effectue et on utilise `simple_solver_obs` pour avoir une estimée du coût de réparation du reste de la séquence, et on choisit comme prochaine action à réaliser celle qui donne la plus petite valeur de coût espéré. L'avantage de cette méthode est de donner, à chaque action, une valeur d'espérance de coût, qui est celle estimée si l'action correspondante est la prochaine action à être prise. Cette modifi-

cation de l'algorithme myope a été implémentée dans la méthode `ECR_ECO_wrapper`, qui retourne la meilleure action, son type, et les espérances de coût de toutes les autres actions possibles.

6.5 Algorithme myope avec élicitation des coûts

Après avoir implémenté les trois algorithmes classiques de la Section 4.4, nous avons commencé l'implémentation des algorithmes décrits dans la Section 5 par celui de la Section 5.3, qui utilise l'élicitation pour améliorer les calculs quand on ne connaît pas la valeur exacte des coûts de réparation de quelques composants. Puisque l'idée est de l'utiliser avec la méthode `myopic_solver` (ou `ECR_ECO_solver`), les méthodes `best_EVOI` et `elicitation` implémentées font uniquement la partie d'élicitation.

Notre implémentation permet à l'utilisateur de choisir à quel moment il veut répondre à des questions sur les coûts. Lorsque l'utilisateur le choisit, la méthode `best_EVOI` doit être appelée. Cette méthode calcule les EVOI(E, c_i, α) pour tous les composants encore réparables c_i et choisit celui de plus grande valeur. Comme indiqué dans la Section 5.3, le calcul des EVOI par la formule exacte (10) n'est pas souhaitable en pratique car cette formule fait intervenir la résolution exacte du problème de *Troubleshooting*. On remplace ainsi le calcul exact des stratégies S^*, S^*_- et S^*_+ de (11) et des EC correspondants par un calcul approché en utilisant la méthode `simple_solver_obs`.

Une fois la meilleure question trouvée, la méthode `elicitation` permet de mettre à jour les informations sur les coûts à partir de la réponse de l'utilisateur. On remarque que `best_EVOI` retourne toujours une question, même si celle-ci n'est pas intéressante à poser (EVOI égale à ou très proche de 0). La gestion du choix de ne pas poser une question dans ce cas est fait directement par l'interface graphique dans notre implémentation.

6.6 Algorithme exact

En comparaison aux algorithmes approchés décrits ci-dessus, les méthodes exactes nécessitent une structure spécifique pour stocker et manipuler leurs solutions. Par conséquent, on a besoin d'une représentation des arbres de stratégie. C'est pourquoi, avant d'implémenter les algorithmes eux-mêmes, nous avons conçu des classes auxiliaires :

- `NodeST` : classe abstraite qui représente un nœud d'un arbre de stratégie ;
- `Repair` : classe qui représente un nœud de réparation ;
- `Observation` : classe qui représente un nœud d'observation ;
- `StrategyTree` : classe qui représente un arbre de stratégie.

Toutes ces classes sont fournies dans le module `StrategyTree.py`. On remarque qu'il n'y a pas de classe séparée pour l'action du type « observation-réparation » car il suffit d'utiliser `Observation` pour ces cas-là. Ces classes auxiliaires nous permettent de gérer assez facilement les objets concernés. Par exemple, la classe `StrategyTree` dispose des méthodes `get_sub_tree` pour obtenir un sous-arbre quelconque, `connect` pour connecter l'arbre avec un autre, ou encore `visualize` pour sauvegarder l'arbre dans un fichier `.pdf`.

Afin de lancer un des algorithmes de force brute, la classe `TroubleShootingProblem` dispose d'une méthode dévouée `brute_force_solver`, qui est une *wrapper* permettant de lancer les deux versions de l'algorithme, le *dénombrement complet* ou la *programmation dynamique*. Par ailleurs, cette méthode prend également en argument des configurations importantes des algorithmes :

- `obs_rep_couples` : si cette variable booléenne vaut `True`, il n'y a qu'une seule action pour les composants $c \in \mathcal{C}_o$, correspondant à un couple « observation-réparation ». Sinon, on aura deux actions séparées, c_o, qui est une observation locale, et c_r, la réparation. Dans ce dernier cas, il est alors possible de réaliser les deux de manière indépendante, ce qui peut améliorer l'espérance de coût dans certaines situations.
- `obs_obsolete` : si cette variable booléenne vaut `True`, on remet en cause les nœuds d'observation globale après chaque réparation. Plus précisément, on considère qu'une réparation peut rendre obsolètes les résultats des observations, nécessitant ainsi de rechercher quelles sont les observations impactées et de supprimer les évidences associées.

Les autres paramètres sont plus techniques et détaillés dans la documentation.

Les versions différentes de l'approche de force brute s'effectuent en appelant les deux méthodes récursives : `_evaluate_all_st` pour le dénombrement complet et `dynamic_programming_solver` pour la programmation dynamique. On remarque que c'est possible d'appeler `dynamic_programming_solver` sans utiliser la *wrapper* ci-dessus tandis qu'appeler `_evaluate_all_st` n'est pas recommandé. Pour calculer l'espérance du coût d'une stratégie dans `_evaluate_all_st`, on utilise la méthode `expected_cost_of_repair`, qui est également récursive (en fait, plutôt sa partie interne `_expected_cost_of_repair_internal`) alors que `dynamic_programming_solver` calcule cette valeur directement.

On utilise un objet du type `StrategyTree` pour stocker la solution optimale trouvée, que l'on retourne dans la fonction `brute_force_solver` avec la valeur du meilleur coût espéré.

6.7 Calcul de l'espérance de coût par une méthode de Monte-Carlo

Suivant les idées présentées dans la Section 5.1, nous avons implémenté plusieurs méthodes permettant d'approcher $\mathrm{EC}(E_0, S)$ pour les stratégies S obtenues avec les algorithmes décrits dans les Sections 6.2 à 6.6. Ces méthodes permettent aussi d'obtenir des statistiques sur le déroulement de la résolution du problème de *Troubleshooting* avec les stratégies S. Elles ont toutes le suffixe `_tester` dans leur nom.

Un de leurs arguments est `true_prices`, un dictionnaire contenant les prix réels de réparation de chaque composant. En effet, un des intérêts de ces méthodes est de tester les algorithmes dans le cas où les prix réels de réparation des composants ne sont pas connus et on applique les algorithmes soit avec les espérances de prix de réparation, soit en faisant des questions d'élicitation pour réduire l'incertitude sur ces prix. Ainsi, la séquence S est toujours calculée avec les espérances de prix de réparation de chaque composant ou avec l'élicitation, mais la valeur du prix total de réparation retourné par les algorithmes `_tester` est calculée en utilisant `true_prices`.

Les autres arguments communs des méthodes `_tester` sont la valeur de ε à être utilisée pour le critère d'arrêt (6) et le nombres minimal et maximal d'itérations à être réalisées, `nb_min` et `nb_max`. Remarquons que `nb_min` se fait nécessaire car l'algorithme peut trouver au début quelques fois de suite la même valeur de coût de réparation, auquel cas $s_n = 0$ et le critère d'arrêt (6) est satisfait, alors qu'on n'est pas encore à la convergence.

Les méthodes `_tester` retournent toujours un booléen indiquant si l'algorithme s'est terminé après avoir atteint `nb_max` itérations (cas `False`) ou à cause du critère d'arrêt (6) (cas `True`). Il retourne aussi les coûts trouvés, le nombre de réparations faites et, le cas échéant, le nombre d'observations globales faites ou de questions d'élicitation posées.

Pour le testeur de la méthode d'élicitation, nous avons décidé de, avant chaque prise d'action, faire le calcul de tous les EVOI et répondre à toutes les questions disponibles (en ayant connaissance des coûts réels) avant de déterminer quelle est la meilleure action à prendre.

6.8 Interface graphique

L'interface graphique a été implémentée en utilisant le module `PyQt5`. Elle permet de sélectionner un des algorithmes décrits dans les Sections 6.2 à 6.6 et de l'utiliser pour résoudre le problème de *Troubleshooting* dans le réseau présenté à la Section 6.1. Alors que la sélection des algorithmes des Sections 6.2 et 6.3 donne juste la séquence d'actions à réaliser, les autres algorithmes utilisent un mode interactif qui, à chaque pas, donne l'action recommandée. Les algorithmes myopes (avec ou sans élicitation) permettent à l'utilisateur de choisir une autre action, alors que l'algorithme exact oblige le choix de l'action optimale donnée, mais ils attendent tous de l'utilisateur qu'il fournisse le résultat des actions. Un manuel d'utilisateur donne des explications détaillées de son utilisation et est fourni avec le code (voir annexe).

7 Résultats

7.1 Tests avec les algorithmes approchés

Les résultats présentés ci-dessous concernent le réseau Bayésien de la Figure 10 de l'Annexe A. Le *problem defining node* de ce réseau est le nœud `car.carWontStart`, le nœud d'appel au service est `callService`. L'ensemble des composants choisi est

$$\mathcal{C} = \{\texttt{car.batteryFlat}, \texttt{oil.noOil}, \texttt{tank.Empty}, \texttt{tank.fuelLineBlocked}, \texttt{starter.starterBroken}\},$$

le sous-ensemble de composants observables contient tous sauf `tank.tankEmpty`. L'ensemble des observations globales est

$$\mathcal{O} = \{\texttt{car.lightsOk}, \texttt{car.noOilLightOn}, \texttt{oil.dipstickLevelOk}\}.$$

Les coûts utilisés sont donnés dans la Table 1, dans laquelle la donnée d'un intervalle indique que nous considérons un coût aléatoire de loi uniforme sur cet intervalle, et les algorithmes qui ne prennent pas en compte cette information utilisent comme coût correspondant la moyenne de cet intervalle.

Table 1 – Coûts d'observation et de réparation utilisés dans les simulations.

	Coût
`car.batteryFlat` (observation)	20
`oil.noOil` (observation)	50
`tank.fuelLineBlocked` (observation)	60
`starter.starterBroken` (observation)	10
`car.lightsOk`	2
`car.noOilLightOn`	1
`oil.dipstickLevelOk`	7
`car.batteryFlat` (réparation)	[100, 300]
`oil.noOil` (réparation)	[50, 100]
`tank.Empty`	[40, 60]
`tank.fuelLineBlocked` (réparation)	150
`starter.starterBroken` (réparation)	[20, 60]
`callService`	500

Nous avons appliqué les algorithmes `simple_solver_tester`, `simple_solver_obs_tester`, `myopic_solver_tester` et `elicitation_solver_tester` à ce problème en utilisant $\varepsilon = 0{,}025$ et $n_0 = 200$ pour le critère d'arrêt (6), avec les coûts réels donnés dans la Table 2. L'algorithme exacte n'a pas été testé sur cette instance car son temps d'exécution est trop long, ses résultats seront donnés dans la suite sur une instance plus petite. Les histogrammes de coûts de réparation obtenus sont représentés sur la Figure 4 et les coûts moyens obtenus sont donnés dans la Table 3, qui présente aussi les temps d'exécution correspondants.

On remarque sur la Figure 4 que, comme attendu, les résultats s'améliorent lorsque l'on utilise des algorithmes plus avancés. L'algorithme simple conduit à des situations où, dans plus de 10% des cas, on trouve des coûts supérieurs à 700, et ne donne jamais des coûts inférieurs à 100, alors que les autres algorithmes donnent toujours des coûts inférieurs à 700 et conduisent dans plusieurs situations à des coûts inférieurs à 100. En effet, en tournant `simple_solver` sur cette instance, la séquence d'actions retournée consiste dans les réparations, dans l'ordre, de `car.batteryFlat`, `oil.noOil` et `tank.Empty` suivies d'un appel au service. Ainsi, les seuls coûts possibles sont 120, $120 + 90 = 210$, $120 + 90 + 55 = 266$ et $120 + 90 + 55 + 500 = 766$.

Table 2 – Coûts réels utilisés pour les réparations avec coût aléatoire.

	Coût
car.batteryFlat	120
oil.noOil	90
tank.Empty	55
starter.starterBroken	50

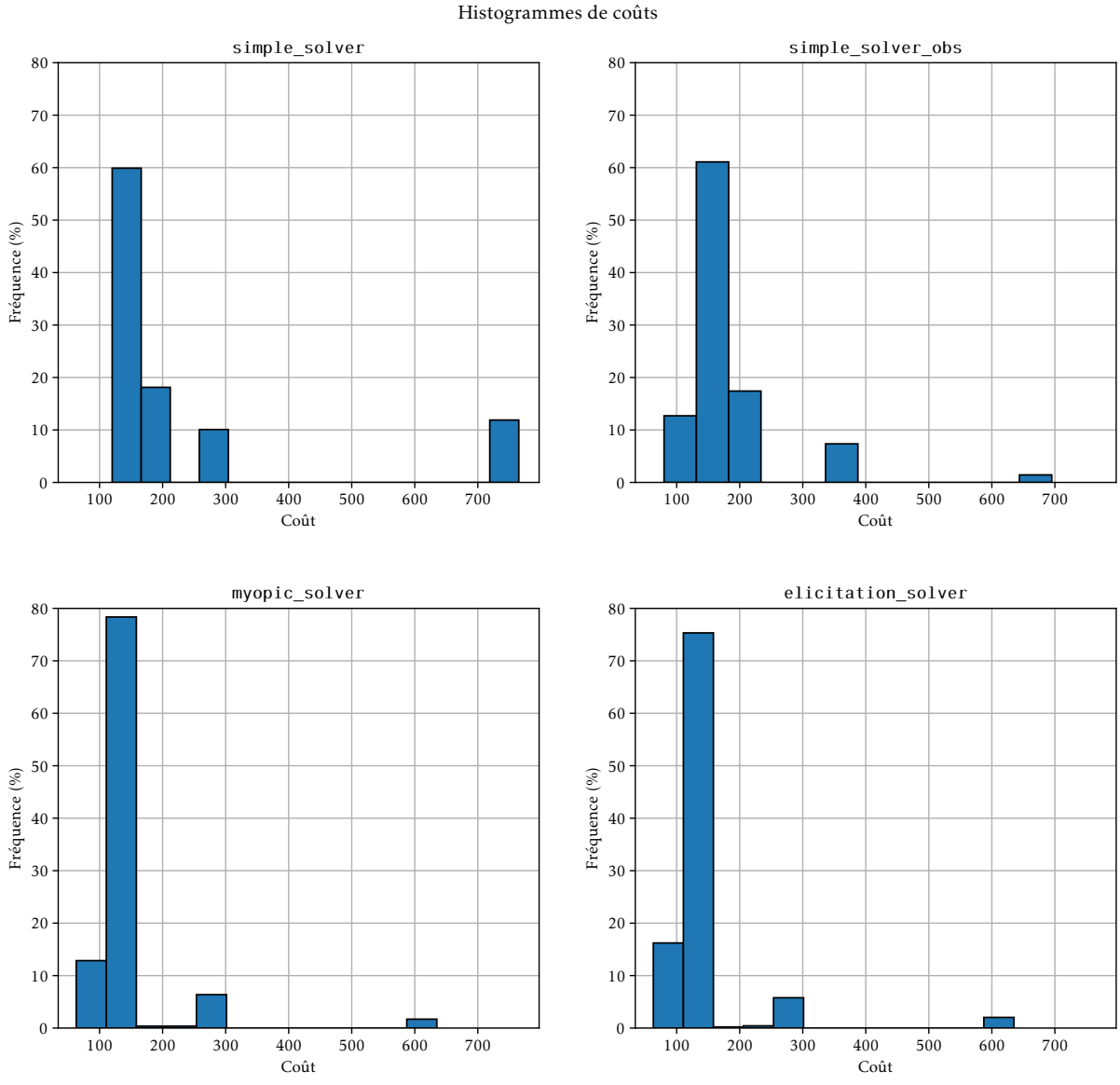

Figure 4 – Histogrammes de coûts de réparation obtenus avec les algorithmes approchés

L'algorithme simple_solver_obs constitue ainsi une bonne amélioration déjà par rapport à simple_solver. Sa séquence d'actions est les réparations ou observations-réparations de car.batteryFlat, starter.starterBroken, tank.Empty, oil.noOil et tank.fuelLineBlocked suivies d'un appel au service. Il se peut ainsi que, après avoir réalisé l'observation de car.batteryFlat pour un coût de 20, on observe que ce composant est déjà en marche et qu'on passe ensuite à starter.starterBroken, dont l'observation a un coût de 10. Si ce composant est défectueux, sa

Table 3 – Coûts de réparation moyens obtenus avec les algorithmes approchés et temps d'exécution.

Algorithme	Coût moyen	Temps d'exécution
`simple_solver`	227,62	20,8s
`simple_solver_obs`	170,66	8,5s
`myopic_solver`	150,72	18min 34s
`elicitation_solver`	149,03	1h 33min 24s

réparation coûte 50, et il est alors possible que le dispositif marche après cette réparation, auquel cas le coût total de réparation sera 80, qui a effectivement été le plus petit coût de réparation observé pour cet algorithme.

Les algorithmes `myopic_solver` et `elicitation_solver` sont beaucoup mieux que les autres, avec des coûts de réparation inférieurs à 160 dans la quasi-totalité des cas (plus de 90%), notamment à cause des observations globales de faible coût, dont les résultats permettent de meilleur choisir les prochaines actions. Les deux sont très proches, avec un coût légèrement inférieur pour `elicitation_solver`. En effet, `myopic_solver` résout le problème en supposant que les coûts dont les valeurs exactes sont inconnues sont égales aux espérances des variables aléatoires correspondantes, ce qui peut conduire à des choix non-optimaux. De son côté, `elicitation_solver` pose des questions pour améliorer les connaissances des vrais coûts, ce qui tend à donner une stratégie plus proche de l'optimale pour les vrais coûts, expliquant ainsi l'amélioration observée des résultats.

Pour l'algorithme avec élicitation, il est intéressant à noter que, comme rappelé dans la Section 3.6, l'élicitation a très souvent un coût pour l'utilisateur. Nous avons choisi de ne pas représenter ce coût ici, mais plutôt de suivre la quantité de questions répondues. Nous avons alors constaté que, dans la simulation réalisée, l'algorithme a répondu toujours à une seule question. Cela suggère ainsi que ce coût de l'élicitation pourrait être assez faible, car le nombre de questions répondues est assez limitée. Finalement, nous croyons que les bénéfices de l'élicitation peuvent être plus évidents lorsque les réseaux sont plus grands et avec plus d'incertitudes sur les coûts, auquel cas on pourrait s'attendre à des différences statistiquement significatives.

Nous avons aussi testé ces algorithmes en remplaçant les valeurs réelles de coût de réparation de la Table 2 par des valeurs tirées aléatoirement selon les intervalles de la Table 1. Bien que les valeurs précises changent, l'allure générale et les conclusions présentées dans l'analyse ci-dessus restent valables pour ces autres coûts aléatoires.

7.2 Tests avec l'algorithme exact

Puisque l'algorithme exact prend trop de temps à s'exécuter sur l'instance considérée dans la Section 7.1 (les calculs n'étaient toujours pas fini au bout de 6h), nous l'avons testé sur une instance plus petite, avec le même réseau Bayésien, le même *problem defining node*, le même nœud d'appel au service, mais avec

$$\begin{aligned}\mathcal{C} &= \{\texttt{car.batteryFlat}, \texttt{tank.Empty}, \texttt{oil.noOil}\},\\ \mathcal{C}_o &= \{\texttt{car.batteryFlat}, \texttt{oil.noOil}\},\\ \mathcal{O} &= \{\texttt{oil.dipstickLevelOk}\}.\end{aligned}$$

Les coûts utilisés sont les mêmes que ceux de la Table 1 mais, puisque l'on ne fait pas de comparaison avec l'algorithme avec l'élicitation dans cette section, on a choisit comme coût réel la moyenne de l'intervalle de coût pour le cas des coûts incertains. Les histogrammes de coûts sont représentés sur la Figure 5 et les valeurs des coûts moyens sont données dans la Table 4. Les Figures 6 et 7 montrent, sous la forme d'un arbre, les stratégies obtenues avec les algorithmes myope et exact, respectivement.

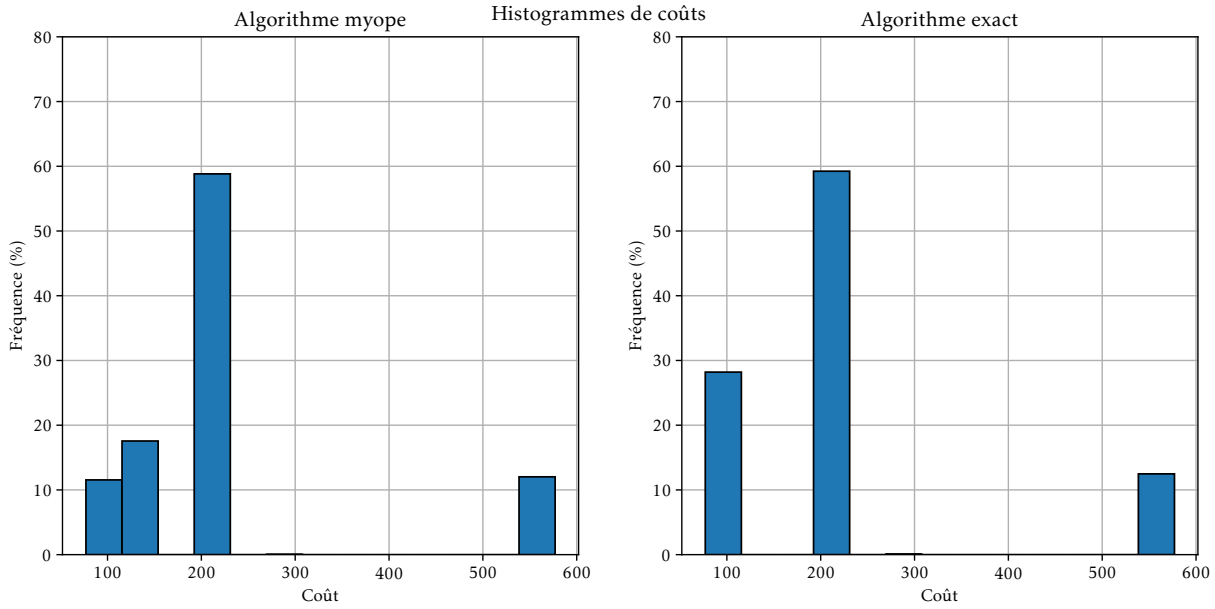

Figure 5 – Histogrammes de coûts de réparation obtenus avec les algorithmes et exact

Table 4 – Coûts de réparation moyens obtenus avec les algorithmes approchés

Algorithme	Coût moyen
Myope	235,10
Exact	229,31

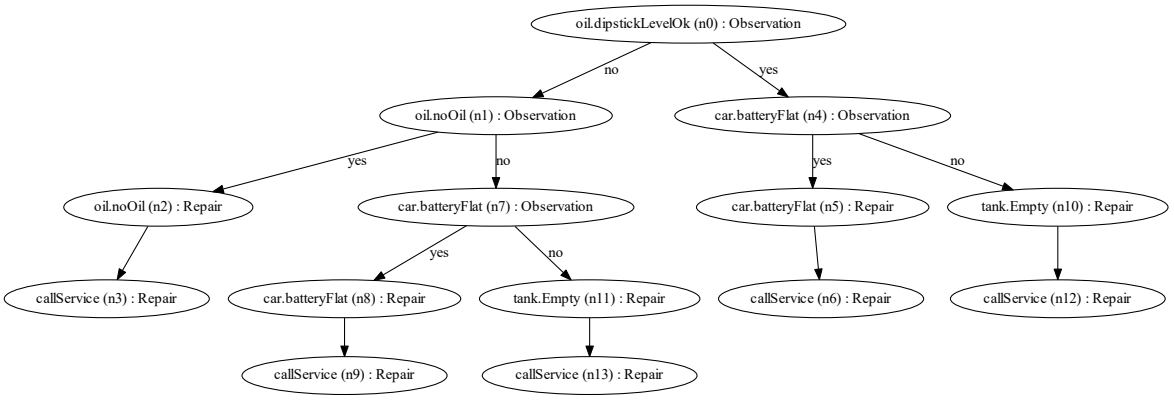

Figure 6 – Stratégie obtenue avec l'algorithme myope.

On observe que, comme attendu, malgré son temps d'exécution plus long, l'algorithme exact trouve bien une espérance de coût de réparation plus petite que celle trouvée par l'algorithme myope. Les stratégies optimales montrées dans les Figures 6 et 7 sont très similaires, avec une seule différence : lorsque l'observation `oil.dipstickLevelOk` indique qu'il y a un problème, l'algorithme myope, suivant son principe, fait d'abord l'observation de `oil.noOil` avant de le réparer si nécessaire, alors que l'algorithme exact, probablement à cause du fait que la probabilité que oil.noOil soit le composant présentant un problème est grande dans ce cas, fait directement sa réparation sans l'observer au préalable. Dans le cas où c'est effectivement ce nœud qui présente un problème (ce qui est le cas avec une probabilité d'environ 18% dans le réseau Bayésien utilisé), cela diminue le coût de réparation de 50 (soit le coût de l'observation de `oil.noOil`) : il est

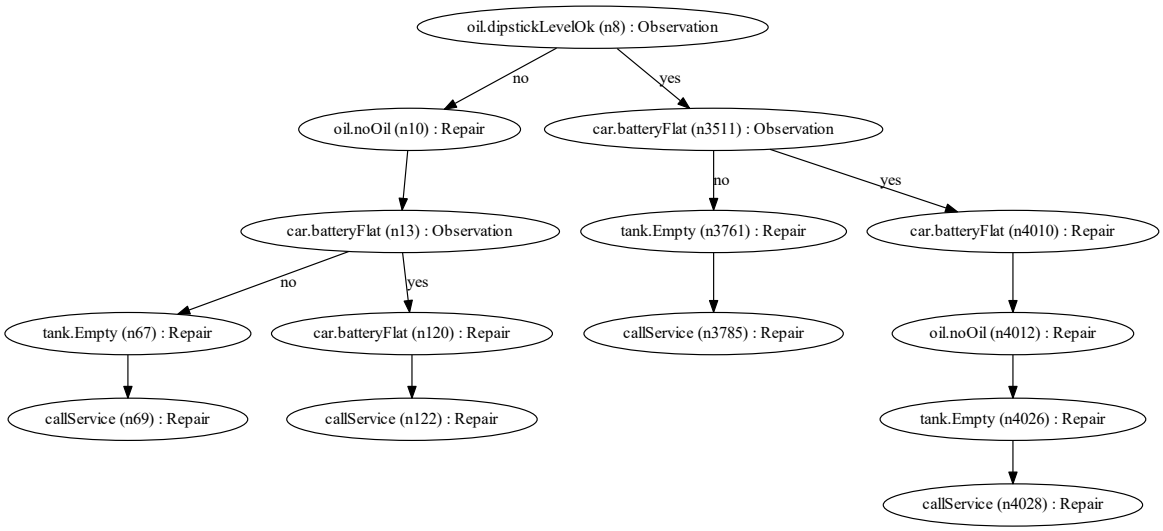

Figure 7 – Stratégie obtenue avec l'algorithme exact.

de 132 pour la stratégie de l'algorithme myope mais de 82 pour la stratégie optimale, ce qui explique les différences dans l'histogramme et dans les espérances de coût. Cela montre aussi que, sur cette instance, l'algorithme myope donne un coût espéré de réparation très proche de celui de l'algorithme exacte.

8 Ouverture sur l'élicitation des coûts

Comme présenté dans la Section 7.1, la technique d'élicitation des coûts proposée dans la Section 5.3 donne souvent des meilleurs résultats que l'algorithme myope sans élicitation de coûts, mais très proches de celui-ci. En général, en tirant les coûts incertains de façon aléatoire et en tournant l'algorithme, on observe que peu de questions sont répondues : il y a presque toujours une question répondue, et très rarement ce nombre dépasse 2 questions.

Un des paramètres de l'algorithme avec élicitation est le choix des α utilisés pour poser les questions, que nous avons fixés de façon heuristique à l'espérance de la loi uniforme du coût de réparation correspondant. Ainsi, une piste d'amélioration possible pour l'algorithme avec élicitation serait d'utiliser des stratégies plus avancées pour le choix de α.

Cependant, la méthode proposée dans la Section 5.3 avec une question d'élicitation à réponse binaire se heure à un obstacle : dans ce cadre, il est possible que, suite à une question d'élicitation, la valeur de coût de réparation utilisée dans l'algorithme après l'élicitation soit plus loin de la valeur réelle qu'avant l'élicitation.

Pour l'illustrer, considérons, pour simplifier, un dispositif dont le coût réel est dans l'intervalle $[0,1]$. Sans questions d'élicitation, l'algorithme considérera que son coût est $\frac{1}{2}$, et l'erreur commise en fonction du coût réel c est $\left|c - \frac{1}{2}\right|$, représentée en bleu sur la Figure 8. Après la question d'élicitation « est-ce que le coût réel est plus petit que $\frac{1}{2}$? » et en supposant une réponse correcte à cette question, l'erreur commise en fonction du coût réel est la courbe représentée en orange sur la Figure 8. On observe ainsi que, pour $c \in \left[\frac{3}{8}, \frac{5}{8}\right]$, l'erreur commise augmente après une réponse correcte à la question d'élicitation. Ce problème est intrinsèque aux questions d'élicitation binaires : indépendamment de la valeur de $\alpha \in (0,1)$ choisie pour la question d'élicitation, il y aura toujours un sous-intervalle de $[0,1]$ pour lequel l'erreur entre coût réel et coût considéré par l'algorithme aura augmenté.

Cette augmentation sur l'erreur commise sur le coût de réparation d'un composant a un impact indirect sur l'espérance de coût de réparation du dispositif. Cela peut ainsi expliquer pourquoi,

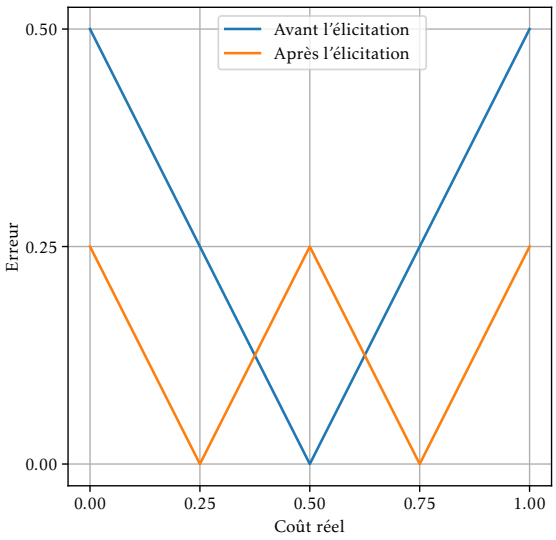

FIGURE 8 – Distance entre le coût utilisé par l'algorithme et le coût réel avant et après une question d'élicitation à réponse binaire.

dans les résultats présentés dans la Section 7.1, l'amélioration observée sur l'espérance de coût avec l'élicitation est petite.

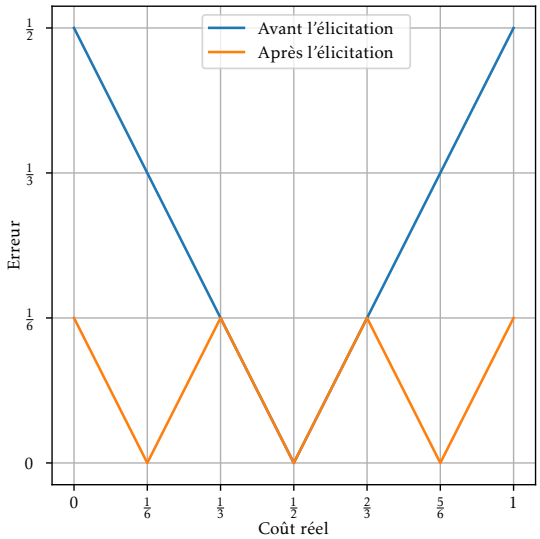

FIGURE 9 – Distance entre le coût utilisé par l'algorithme et le coût réel avant et après une question d'élicitation à réponse ternaire.

Une possible solution à ce problème est de passer à des questions d'élicitation avec plus de deux réponses possibles. Considérons, par exemple, toujours dans le cas de coûts réels dans $[0,1]$, une question de la forme « le coût réel est dans lequel des intervalles $[0,\beta)$, $[\beta,\gamma)$, ou $[\gamma,1]$? » pour des valeurs $0 < \beta < \gamma < 1$. Afin de ne pas avoir une augmentation d'erreur en aucun point après la réponse à la question, il faut que β et γ soient symétriques, c'est-à-dire $\gamma = 1 - \beta$, car dans le cas contraire l'erreur commise pour le coût réel $\frac{1}{2}$ est forcément strictement positive après l'élicitation, alors qu'elle est nulle avant. Dans ce cas, le choix de β qui minimise l'erreur maximale après l'élicitation est $\beta = \frac{1}{3}$, auquel cas on obtient les courbes d'erreur de la Figure 9. Bien évidemment, les techniques décrites pour l'élicitation dans la Section 5.3 peuvent être adaptées au cas de cette

question, et il serait intéressant, dans des travaux futurs, de tester expérimentalement l'apport de cette modification de la question. Dans des situations pratiques, il faut en plus prendre en compte le fait qu'une question sur les coûts avec trois réponses possibles peut être plus difficile à répondre pour l'utilisateur.

Références

Craig Boutilier. 2002. A POMDP Formulation of Preference Elicitation Problems. In *Eighteenth National Conference on Artificial Intelligence*. American Association for Artificial Intelligence, USA, 239–246.

Craig Boutilier, Relu Patrascu, Pascal Poupart, and Dale Schuurmans. 2006. Constraint-based optimization and utility elicitation using the minimax decision criterion. *Artificial Intelligence* 170, 8-9 (2006), 686–713. DOI:http://dx.doi.org/10.1016/j.artint.2006.02.003

Darius Braziunas and Craig Boutilier. 2005. Local Utility Elicitation in GAI Models. In *Proceedings of the Twenty-First Conference on Uncertainty in Artificial Intelligence (UAI'05)*. AUAI Press, Arlington, Virginia, USA, 42–49.

Darius Braziunas and Craig Boutilier. 2008. Elicitation of Factored Utilities. *AI Magazine* 29, 4 (dec 2008), 79. DOI:http://dx.doi.org/10.1609/aimag.v29i4.2203

Urszula Chajewska, Daphne Koller, and Ronald Parr. 2000. Making Rational Decisions Using Adaptive Utility Elicitation. In *Proceedings of the Seventeenth National Conference on Artificial Intelligence and Twelfth Conference on Innovative Applications of Artificial Intelligence*. AAAI Press, 363–369.

Christophe Gonzales, Lionel Torti, and Pierre-Henri Wuillemin. 2017. aGrUM: A Graphical Universal Model Framework. In *Advances in Artificial Intelligence: From Theory to Practice*. Springer International Publishing, 171–177. DOI:http://dx.doi.org/10.1007/978-3-319-60045-1_20

David Heckerman, John Breese, and Koos Rommelse. 1994. *Troubleshooting under uncertainty*. Technical Report. Microsoft Research Technical Report MSR-TR-94-07.

David Heckerman, John S. Breese, and Koos Rommelse. 1995. Decision-theoretic troubleshooting. *Commun. ACM* 38, 3 (mar 1995), 49–57. DOI:http://dx.doi.org/10.1145/203330.203341

Ronald Howard. 1966. Information Value Theory. *IEEE Transactions on Systems Science and Cybernetics* 2, 1 (1966), 22–26. DOI:http://dx.doi.org/10.1109/tssc.1966.300074

Vijay S. Iyengar, Jon Lee, and Murray Campbell. 2001. Evaluating Multiple Attribute Items Using Queries. In *Proceedings of the 3rd ACM Conference on Electronic Commerce (EC '01)*. Association for Computing Machinery, New York, NY, USA, 144–153. DOI:http://dx.doi.org/10.1145/501158.501174

Finn V. Jensen, Uffe Kjærulff, Brian Kristiansen, Helge Langseth, Claus Skaanning, Jiří Vomlel, and Marta Vomlelová. 2001. The SACSO methodology for troubleshooting complex systems. *Artificial Intelligence for Engineering Design, Analysis and Manufacturing* 15, 4 (sep 2001), 321–333. DOI:http://dx.doi.org/10.1017/s0890060401154065

Finn V. Jensen and Thomas D. Nielsen. 2007. *Bayesian Networks and Decision Graphs*. Springer New York, New York, NY. DOI:http://dx.doi.org/10.1007/978-0-387-68282-2

H Langseth. 2003. Decision theoretic troubleshooting of coherent systems. *Reliability Engineering & System Safety* 80, 1 (apr 2003), 49–62. DOI:http://dx.doi.org/10.1016/s0951-8320(02)00202-8

Václav Lín. 2014. Decision-theoretic troubleshooting: Hardness of approximation. *International Journal of Approximate Reasoning* 55, 4 (jun 2014), 977–988. DOI:http://dx.doi.org/10.1016/j.ijar.2013.07.003

D. North. 1968. A Tutorial Introduction to Decision Theory. *IEEE Transactions on Systems Science and Cybernetics* 4, 3 (1968), 200–210. DOI:http://dx.doi.org/10.1109/tssc.1968.300114

Bob Price and Paul R. Messinger. 2005. Optimal Recommendation Sets: Covering Uncertainty over User Preferences. In *Proceedings of the 20th National Conference on Artificial Intelligence - Volume 2 (AAAI'05)*. AAAI Press, 541–548.

Paolo Viappiani and Craig Boutilier. 2010. Optimal Bayesian Recommendation Sets and Myopically Optimal Choice Query Sets. In *Proceedings of the 23rd International Conference on Neural Information Processing Systems - Volume 2 (NIPS'10)*. Curran Associates Inc., Red Hook, NY, USA, 2352–2360.

Marta Vomlelová. 2003. Complexity of decision-theoretic troubleshooting. *International Journal of Intelligent Systems* 18, 2 (jan 2003), 267–277. DOI:http://dx.doi.org/10.1002/int.10087

John von Neumann and Oskar Morgenstern. 1953. *Theory of games and economic behavior* (third ed.). Princeton University Press, Princeton, N.J.

A Réseau bayésien utilisé

Le réseau bayésien utilisé pour les tests et implémenté dans la version du logiciel avec l'interface graphique est celui donné dans la Figure 10 ci-après.

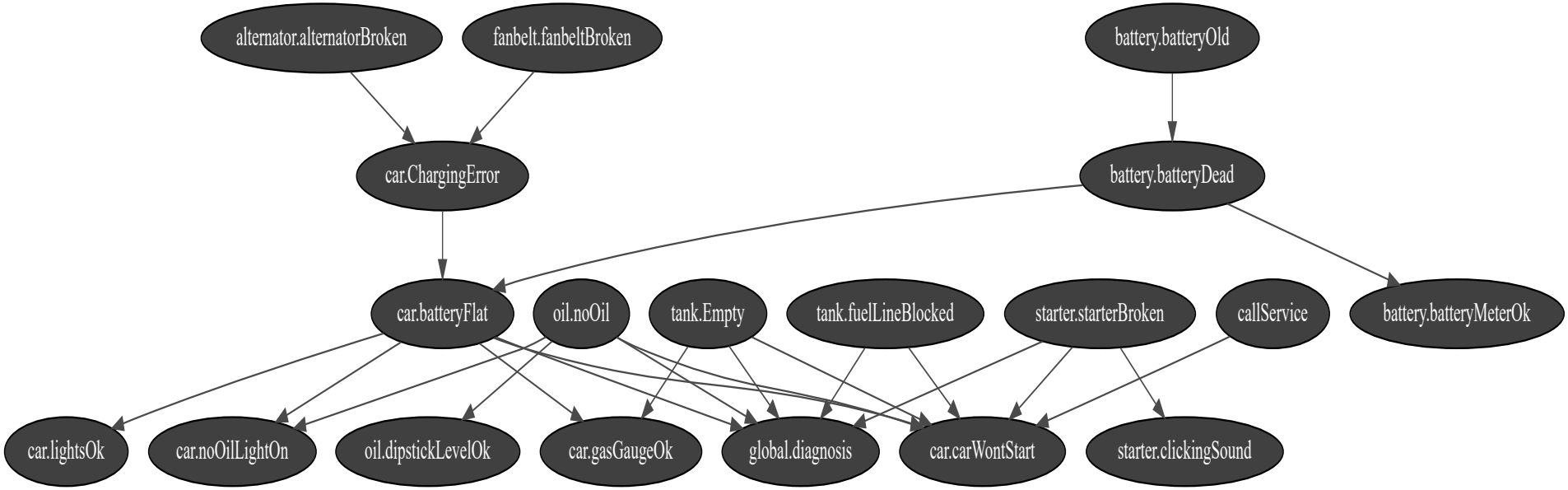

Figure 10 – Réseau Bayésien utilisé pour tester les méthodes implémentées