Fiabilité, maintenance, simulation et combinatoire

Guillaume Thisselin

SOMMAIRE

REMERCIEMENTS

Je tiens à remercier chaleureusement Marie-Claude Portmann, co-responsable scientifique de l'équipe MACSI et professeur à l'Ecole des Mines de Nancy, de m'avoir proposé ce sujet de DEA et de m'avoir encadré dans mon travail tout au long des quatre mois qu'a duré mon stage. Je lui suis tout particulièrement reconnaissant pour sa constante disponibilité, pour les nombreuses heures qu'elle a consacrées à nos réunions de travail, à la relecture et à la correction de ce mémoire, mais surtout pour toutes les idées qui sont ressorties de nos discussions. Je la remercie enfin de m'avoir laissé une très grande liberté de travail, liberté qui j'en suis sûr, m'a permis de travailler plus sereinement et plus efficacement.

INTRODUCTION GÉNÉRALE

1 – Contexte de recherche

Ce mémoire de DEA a été effectué au sein de l'équipe MACSI du LORIA et de l'INRIA-Lorraine. Le sigle MACSI signifie « Modélisation, Analyse et Conduite des Systèmes Industriels ». Les activités de l'équipe s'articulent autour de trois thèmes : la modélisation des systèmes industriels, l'évaluation des performances et le dimensionnement des systèmes de production, la conception et la conduite des systèmes de production. Ce travail s'insère dans ce dernier thème et fait suite à une collaboration avec l'Université Laval de Québec, suite à la venue en 2002 du professeur Daoud Ait-Kadi comme professeur invité à l'Ecole des Mines de Nancy.

Le domaine d'application concerne la fiabilité et la sûreté des systèmes complexes qui peuvent aussi bien être des systèmes de production industriels que des systèmes d'informatique distribués. Dans le cadre de la sûreté de fonctionnement, on peut distinguer en particulier plusieurs grands thèmes de recherche :

- La conception de systèmes complexes répondant à un objectif donné, généralement un compromis entre le coût et la fiabilité du système (par exemple, fiabilité maximale à coût limité ou coût minimal à fiabilité minimale fixée). L'amélioration de la fiabilité passe par l'ajout de composants redondants. L'optimisation consiste à choisir des sous-ensembles de composants redondants et éventuellement à trouver des agencements particuliers de ces composants. Ces problèmes sont en général NP-difficiles et sont abordés par des méthodes approchées relevant de l'optimisation discrète.
- L'utilisation de systèmes complexes en temps réel, leur entretien et leur maintenance. On suppose ici que le système complexe préexiste et que l'on connaît les lois de probabilités de pannes, de remises en service ainsi que tout coût associé (test, réparation, perte de productivité, …). Ces problèmes sont traités par les spécialistes de politique de maintenance industrielle.
- La détection et la prévision des pannes. Cette discipline s'intéresse à la sûreté de fonctionnement et donc à tout outil permettant de surveiller le système complexe et d'apporter de la connaissance sur ses éventuelles dégradations lors de ses défaillances ou mieux avant même l'apparition des défaillances. Une partie de cette discipline s'intéresse à la minimisation des coûts de tests de systèmes complexes, c'est-à-dire comment trouver au moindre coût la cause de la panne ou encore comment obtenir au moindre coût que le système reparte.

Dans le cadre de ce travail, nous nous intéressons de manière privilégiée au dernier thème avec comme objectif principal de faire repartir au moindre coût un système complexe tombé en panne. Ceci induit que le système n'est pas obligatoirement complètement réparé à chaque fois et que son état évolue au cours du temps. Ainsi, pour l'obtention d'un outil industriel intéressant, il faut non seulement considérer le moindre coût ponctuel (ce qui correspond à la partie principale de ce travail), mais également s'intéresser à l'évolution du système au cours du temps et donc également à des politiques de maintenance. Une partie de ce travail a été généralisée pour tenir compte de ce fait.

2 – Le problème considéré et son positionnement scientifique

Le problème consiste à minimiser l'espérance mathématique du coût de remise en marche d'un système complexe qui vient juste de tomber en panne. Le problème modélisé nous conduit à un problème d'optimisation stochastique discrète et à la recherche de politiques optimales. Ce problème combinatoire est très difficile et n'appartient même pas à la classe des problèmes NP. En effet, si on se donne une solution du problème, c'est-à-dire l'ordre dans lequel on teste les éléments, alors la simple évaluation de l'espérance mathématique du coût de la solution proposée est en $0(2^n)$ s'il y a n composants dans le système. En outre, le testeur peut et doit en temps réel changer de politique en fonction de l'état des composants qu'il a déjà testé (aspect dynamique de la politique optimale à adopter). En effet, en cours de procédure de tests, la connaissance partielle de l'état du système permet de déduire des informations complémentaires sur certaines parties du système et d'améliorer l'espérance mathématique du coût de la fin des tests. Il n'est donc pas possible de prévoir a priori une séquence unique des tests à réaliser qui minimiserait l'espérance mathématique du coût de remise en marche du système. La description même d'un solution calculée a priori sera donnée sous forme d'un arbre de tests donnant les tests à réaliser au fur et à mesure des résultats des tests effectués.

Face à la difficulté de ce problème, nous avons adopté deux stratégies complémentaires.

Une première stratégie a consisté à concevoir des méthodes de résolution exacte : une de type procédure par séparation et évaluation utilisant des bornes pour tronquer la recherche arborescente et quelques propagations locales de contraintes spécifiques ; une autre par la programmation dynamique mettant en commun des parties communes et permettant d'apporter des éléments complémentaires utiles à une politique plus générale de maintenance du système au cours du temps. Cette première stratégie ne peut malheureusement être appliquée qu'à des problèmes de tailles relativement petites (une quinzaine de composants).

Une deuxième stratégie a consisté à s'intéresser à des méthodes de résolution approchée : il s'agit cette fois d'obtenir une espérance mathématique aussi bonne que faire se peut tout en maintenant une durée raisonnable pour l'obtention de cette solution. Plusieurs méthodes simples ont déjà été proposées. Nous proposons ici une seule méthode nouvelle construite sur le principe de la simplification des méthodes arborescentes exactes de manière à contrôler plus ou moins bien le compromis entre la qualité du coût de tests et la durée du calcul.

Nous avons cherché dans la littérature des exemples de systèmes complexes comportant des éléments redondants. Néanmoins en l'absence de données numériques précises correspondant aux hypothèses de notre travail, nous avons dû générer aléatoirement les coûts et les probabilités de panne. Des expériences montrent l'intérêt de nos méthodes.

Ce travail n'est qu'une étape dans le cadre de l'intégration de la minimisation des coûts de tests et de réparation pour la remise en marche d'un système complexe au sein d'une politique globale de maintenance. Plusieurs pistes de continuation seront données en conclusion.

I - Présentation du problème et généralités

I.1 - PRÉSENTATION DU PROBLÈME, DU TRAVAIL À EFFECTUER ET DES TRAVAUX EXISTANTS

I.1.1 – Définition d'un système complexe

Un système est un ensemble de composants interactifs qui travaillent ensemble afin de réaliser une fonction spécifique sous un ensemble donné de conditions. On dit qu'un système tombe en panne (ou en état de défaillance) lorsqu'il cesse d'exécuter la fonction pour laquelle il a été conçu [1]. Les causes de défaillance d'un système peuvent être très différentes selon les systèmes : humidité, poussière, corrosion, vibrations, erreur humaine, surtension, …

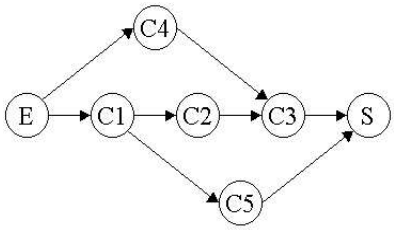
Lorsque le système tombe en panne, sa remise en état de marche a un coût. Il faut d'une part trouver la provenance de la panne (c'est le diagnostic), et d'autre part réparer un ou plusieurs composants (c'est la réparation).

Pour chaque composant, on définit trois coûts : le coût de démontage, le coût de test et le coût de réparation.

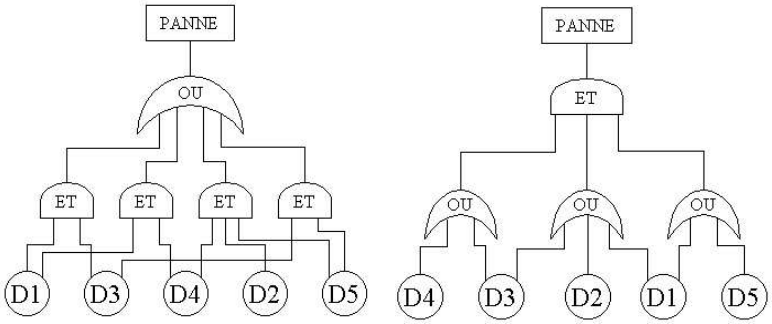
I.1.2 - Représentations d'un système et critères de fonctionnement

a) Diagramme de fiabilité et arbres de défaillance

Il existe principalement deux représentations pour un système complexe : un diagramme de fiabilité et un arbre de défaillance.



Exemple d'un diagramme de fiabilité



Arbres de défaillance possibles correspondant au diagramme de fiabilité précédent

Un diagramme de fiabilité est un graphe sans circuit. Les sommets de ce graphe représentent les composants du système. Le graphe est également doté de deux sommets artificiels : une entrée **E** et une sortie **S**. Le diagramme de fiabilité est tel que le système fonctionne si et seulement s'il existe un chemin de **E** vers **S** tel que tous les composants de ce chemin fonctionnent. De plus, pour un système donné, il n'y a pas unicité du diagramme de fiabilité.

Un arbre de défaillance est une formule booléenne qui traduit la combinaison des événements indésirables qui conduisent à la défaillance du système.

Il existe des cas particuliers de systèmes pour lesquels on est incapable de dessiner un diagramme de fiabilité où chaque composant n'apparaît qu'une fois et où il n'y a pas de composant fictif. C'est le cas par exemple d'un système de 3 composants qui fonctionne si et seulement si au moins 2 des 3 composants fonctionnent.

En revanche, on ne rencontre pas ce problème d'existence avec les arbres de défaillance puisqu'ils représentent une formule booléenne caractérisant le fonctionnement du système et que cette fonction booléenne existe toujours.

b) Chemins de succès minimaux et coupes minimales d'un système complexe

Un **chemin de succès** est une suite de composants reliant l'entrée et la sortie du diagramme de fiabilité. Un chemin de succès est dit **minimal** s'il ne contient pas de chemin de succès comportant strictement moins de composants que lui-même. Le système complexe fonctionne si et seulement si tous les composants d'au moins un chemin de succès minimal fonctionnent (par extension, on dit alors que le chemin fonctionne). On peut construire un arbre de défaillance en utilisant cette propriété. Sur notre exemple, il s'agit du 2ème arbre de défaillance.

Une **coupe** est un ensemble de composants tel que tout chemin de succès comporte au moins un composant de la coupe. Une coupe est dite **minimale** si elle ne contient pas de coupe comportant strictement moins de composants qu'elle-même. Le système complexe est en panne si et seulement s'il existe une coupe minimale dont tous les composants sont en panne (par extension, on dit alors que la coupe est en panne). On peut construire un arbre de défaillance en utilisant cette propriété. Sur notre exemple, il s'agit du 1er arbre de défaillance.

I.1.3 - Hypothèses générales et objectif

a) Objectif

Nous considérons ici un système complexe qui vient de tomber en panne. On suppose que, compte tenu de la probabilité de panne de chacun des composants du système, la probabilité que deux composants tombent en panne simultanément est infinitésimale.

Notre objectif est de payer le moins possible pour faire repartir le système de manière quasi-certaine. C'est-à-dire que si effectivement un seul composant vient de tomber en panne, le système doit repartir à coup sûr. En revanche, si on est dans le cas très rare où deux composants viennent de tomber en panne simultanément, il se peut que le système ne reparte pas.

Nous allons principalement étudier deux sous cas :

- On ne peut réparer un composant que s'il a été testé au préalable.
- On peut réparer un composant même s'il n'a pas encore été testé.

b) Hypothèses sur les coûts

Nous allons expliquer dans cette partie, que, sauf cas exceptionnel le coût de démontage peut être inclus soit au coût de test, soit au coût de réparation. Pour cela, nous devons distinguer 4 cas de figures :

- On peut tester sans démonter et on peut réparer sans démonter.
- On doit démonter pour tester, mais on peut réparer sans démonter.
- On peut tester sans démonter, mais on doit démonter pour réparer.
- On doit démonter pour tester et pour réparer.

On suppose bien entendu que le composant « démonté » n'est pas remonté après son test lorsqu'il ne fonctionne pas et qu'il reste démonté dans l'optique d'une éventuelle réparation. Ceci sous-entend que l'on démonte un composant au plus une fois.

<table>
<tr><th></th><th>On doit tester avant de réparer</th><th>On peut réparer sans tester</th></tr>
<tr><td>On peut tester sans démonter et on peut réparer sans démonter</td><td colspan="2">Pas de démontage.</td></tr>
<tr><td>On doit démonter pour tester, mais on peut réparer sans démonter</td><td colspan="2">Le démontage n'ayant lieu que pour effectuer un test, le coût de démontage peut-être inclus dans le coût de test.</td></tr>
<tr><td>On peut tester sans démonter, mais on doit démonter pour réparer</td><td colspan="2">Le démontage n'ayant lieu que pour effectuer une réparation, le coût de démontage peut-être inclus dans le coût de réparation.</td></tr>
<tr><td>On doit démonter pour tester et pour réparer</td><td>Comme le test est obligatoire avant la réparation, on inclut le coût de démontage dans le coût de test.</td><td>cas critique</td></tr>
</table>

Dans le cas critique, on ne peut inclure le coût de démontage ni au coût de test, ni au coût de réparation, car on peut être amené à effectuer seulement un test ou seulement une réparation. On peut cependant décider de l'inclure au coût de test et utiliser deux coûts de réparation : un coût de réparation lorsque le composant est déjà démonté et un coût lorsqu'il ne l'est pas.

Nous supposerons pour la suite, que le coût de démontage est soit nul, soit inclus à l'un des deux autres coûts. Ainsi, nous ne manipulerons que deux coûts : test et réparation.

I.1.4 – Travaux existants

En abordant ce problème, nous avons voulu rester le plus général possible, en considérant des systèmes complexes quelconques, en supposant qu'il existait plusieurs coûts (test, réparation, démontage), en utilisant conjointement la connaissance des chemins de succès minimaux et des coupes minimales pour détecter les composants en panne et les composants à réparer. Pour cette raison, nous n'avons pas trouvé dans la bibliographie de travaux correspondant au problème tel que nous l'avons énoncé, et ce, malgré les nombreuses

recherches effectuées dans INPEC, et les prises de contacts avec Eric Chatelet, Daoud Ait-Kadi et Yves Dutuis.

Nous avons trouvé de nombreux articles concernant des politiques de maintenance. Mais, même si notre problème s'insère à terme dans un processus plus complexe où la politique de maintenance est primordiale, ces articles ne nous concernaient pas directement pour le stage de DEA.

Nous avons également trouvé de nombreux articles traitant du cas des systèmes séries dont le plus cité est [2]. Seul Jinsheg Gao et Daoud Ait-Kadi se sont intéressés à un problème relativement « proche » du notre dans [4]. La différence vient du fait qu'il n'y a que des coûts de tests et que seules les coupes minimales du système sont utilisées. Dans ce papier, ils nous présentent une méthode de résolution exacte permettant de trouver au moindre coût une coupe en panne dans le cas où les coupes minimales sont indépendantes. Dans [5], Marie-Claude Portmann propose une méthode de résolution approchée dans le cas où les coupes ne sont plus indépendantes. Pourtant, la méthode exacte présentée dans [4] n'est plus exacte dès lors que l'on décide de tenir compte des chemins de succès (démonstration en Annexe I).

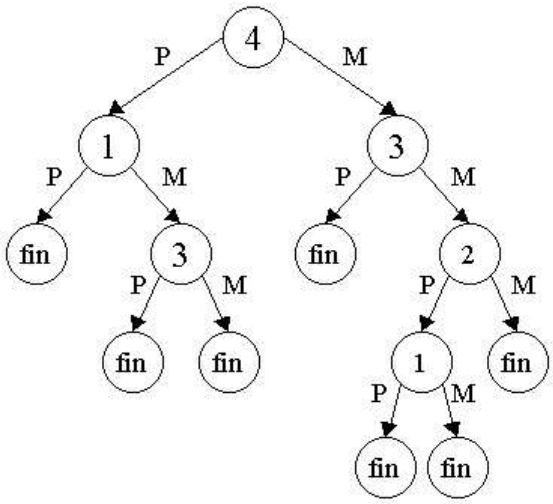
I.2 - ARBRE DE TESTS ET ESPÉRANCE MATHÉMATIQUE

I.2.1 - Organisation de la remise en marche du système complexe.

La remise en marche du système s'organise en deux temps. Il y a tout d'abord la phase de diagnostic ou « phase de tests ». Durant cette phase, on teste successivement plusieurs composants jusqu'à pouvoir déterminer un ensemble des composants à réparer pour faire repartir le système. La séquence des composants à tester n'est pas connue à l'avance et se détermine au fur et à mesure, en fonction de l'état de fonctionnement réel des composants déjà testés. Après la phase de diagnostic, il y a la phase de réparation pendant laquelle on répare l'ensemble des composants trouvé lors de la phase de diagnostic, et ce, sans faire aucun test supplémentaire.

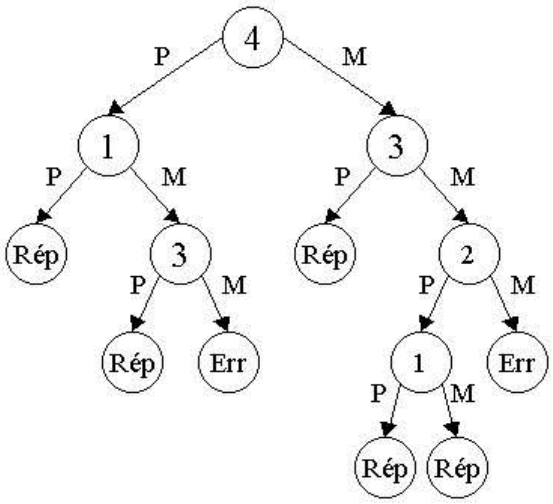
I.2.2 - Arbre de tests

La séquence des éléments à tester au cours de la phase de diagnostic est dynamique. C'est-à-dire que lorsque l'on effectue notre $i^{ème}$ test, on ne connaît pas encore forcément le $i+1^{ème}$ composant à tester. Comme l'état de fonctionnement d'un composant est soit en marche, soit en panne, on peut voir la phase de diagnostic comme la descente dans un arbre binaire.



Allure de la phase de diagnostic

Sur cet exemple, on voit que si l'on suit cet arbre de tests, il faut commencer par tester le composant n°4. Si ce composant est en panne, il faudra alors tester le composant n°1, sinon, il faudra tester le composant n°3. On descend ainsi dans l'arbre selon les résultats obtenus au cours des différents tests, jusqu'à arriver à une feuille marquant la fin des tests. Certaines feuilles de fin sont inaccessibles dans la pratique (ce sont les feuilles auxquelles on ne peut arriver que si le système fonctionne). Pour les autres, la fin des tests est suivie de la phase de réparation. Au final, l'arbre de tests a l'allure suivante :



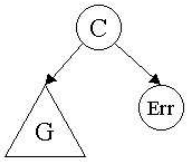
Allure d'un arbre de tests

I.2.3 - Espérance mathématique

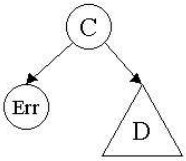
Comme la phase de diagnostic se fait par une descente dans l'arbre de tests et que cette descente est stochastique (suivant les probabilités de panne ou de fonctionnement des composants), on ne peut pas minimiser le coût de remise en marche du système, mais seulement l'espérance mathématique de ce coût (i.e. espérance mathématique de l'arbre de tests). Notre objectif sera donc de construire un arbre de tests dont l'espérance mathématique est minimale.

L'espérance mathématique d'un arbre de tests se calcule récursivement, en suivant les règles de calcul présentées ci-dessous.

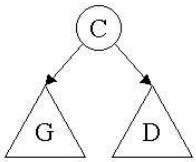
EM = coût de la meilleure réparation



EM = coût de test de C + EM(G)



EM = coût de test de C + EM(D)



EM = coût de test de C + $p(C).EM(G) + \overline{p(C)}.EM(D)$

II - Les méthodes de résolution exacte

II.1 – PRÉAMBULE

L'objectif de notre travail est de préparer « hors ligne », des algorithmes (i.e. les arbres de tests à explorer) que l'on exécutera en utilisant des données supposées acquises « en ligne » (les résultats des tests successifs). Notre but est trouver le meilleur arbre dans l'ensemble des arbres de tests possibles. Il s'agit donc d'un problème de recherche combinatoire.

En Recherche Opérationnelle, pour les problèmes de ce type, on distingue deux grandes familles de méthodes de résolution exacte :

- L'exploration explicite (ou énumération en extension) qui consiste simplement à énumérer toutes les solutions possibles pour ne conserver que la meilleure.
- L'exploration implicite (ou énumération en compréhension) qui consiste à ne pas faire certains calculs soit parce qu'ils ont déjà été effectués (élimination des parties communes), soit parce qu'on peut prouver qu'ils n'aboutiront pas (élimination des parties inutiles). Parmi les méthodes d'exploration implicite, on distingue principalement :
    - Les procédures par séparation et évaluation (PSE) [7]
    - La programmation dynamique [7]
    - La programmation par contraintes [8]

Dans cette partie, nous présentons trois méthodes de résolution exacte. La première s'appuiera sur une exploration explicite de l'ensemble des arbres de tests solutions. La seconde sera une exploration implicite s'apparentant à une PSE, mais se différenciant d'une PSE classique par l'utilisation de bornes supérieures localement globale. Cette modification est imposée par l'environnement stochastique du problème considéré. Enfin, la dernière méthode utilisera une approche de type programmation dynamique.

II.2 - CONSTRUCTION DE L'ARBRE DE TESTS OPTIMAL PAR UNE EXPLORATION EXPLICITE DE L'ENSEMBLE DES SOLUTIONS

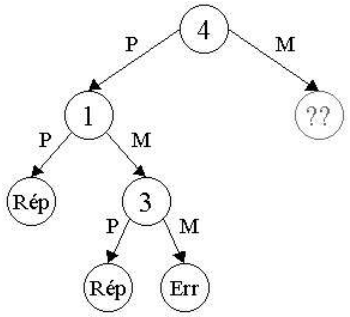
II.2.1 – Grandes lignes de la construction de l'arbre de tests

a) Rappel sur la structure d'un arbre de tests.

Nous rappelons qu'un arbre de tests est un arbre binaire dont les nœuds sont les composants à tester et dont les feuilles correspondent soit à un état inaccessible, soit à une réparation à effectuer.

b) Nœuds virtuels et nœuds définitifs

L'arbre de tests optimal se construit de manière récursive. Lors de cette construction récursive, à un nœud quelconque de l'arbre qui n'est pas une feuille erreur, on est amené à choisir entre réparer (lorsque cela est possible) ou tester un des composants encore non testés. Dans ce dernier cas, c'est l'optimisation stochastique en espérance mathématique qui doit décider quel composant tester.



Que faire dans le cas où 4 fonctionne ?

A un point de choix quelconque, on va donc créer plusieurs sous-arbres virtuels :

- Un sous-arbre « réparation ». Il ne s'agit en fait que d'une feuille réparation. Bien entendu, ce sous-arbre n'existe que si la réparation du système est possible au vu des tests déjà effectués.
- Des sous-arbres de tests. On aura autant de sous-arbres que de composants encore non testés. C'est d'ailleurs au niveau de la création de ces sous-arbres virtuels qu'a lieu la récursivité.

Puis, on évalue l'espérance mathématique de chaque sous-arbre virtuel, et on transforme le sous-arbre virtuel dont l'espérance mathématique est la plus petite en sous-arbre définitif.

c) Algorithme de construction

```
Arbre construireArbreTestsExplicite() {
   // vérification de l'état de fonctionnement du système
   si (système fonctionne) alors
      return Feuille(erreur)
   fsi
   // construction des sous-arbres virtuels
   listeSousArbresVirtuels = vide
```

```
        si (réparation possible) alors
           listeSousArbresVirtuels += Feuille(réparation)
        fsi
        E = ensemble des composants non encore testés
        Pour tout c ∈ E faire
           mettre c à l'état de fonctionnement « panne »
           arbreGauche = construireArbreTestsExplicite()
           mettre c à l'état de fonctionnement « marche »
           arbreDroit = construireArbreTestsExplicite()
           listeSousArbresVirtuels += <arbreGauche,c,arbreDroit>
           mettre c à l'état de fonctionnement « indéterminé »
        finpour
        // choix d'un sous-arbre définitif
        return meilleurSousArbreVirtuel(listeSousArbresVirtuels)
    }
```

d) Complexité temporelle de l'algorithme

Soit un système complexe à n composants. On note $C_T(p)$ la complexité temporelle nécessaire pour construire, dans le pire des cas, un sous-arbre de tests, sachant que p composants du système sont encore dans un état indéterminé.

Si on regarde le fonctionnement de l'algorithme, pour chaque composant non testé, on doit construire deux sous-arbres correspondant aux deux états possible du composant après son tests. Donc on a naturellement : $C_T(p) = O(1) + 2.p.C_T(p-1)$. Par récurrence, on peut voir que $C_T(p) = O(2^p.p!)$. Donc, pour notre système à n composants, la complexité temporelle dans le pire des cas est en $O(2^n.n!)$.

II.2.2 – Procédures détaillées liées à l'efficacité de la construction de l'arbre de tests.

a) Les appels récurrents à certaines fonctions.

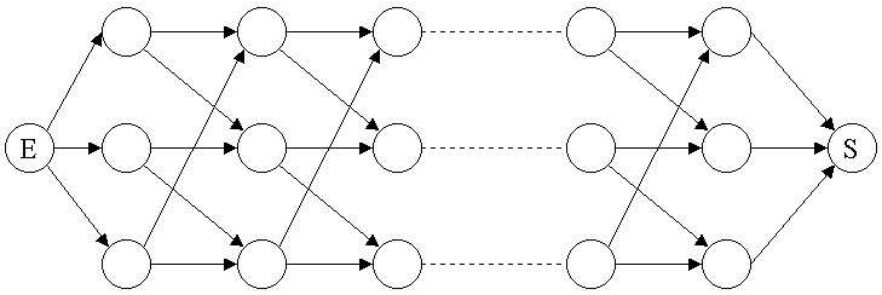
Lors de la construction de l'arbre de tests optimal, on se rend compte qu'à chaque nœud, avant même de construire nos sous-arbres virtuels, on a besoin de savoir d'une part si le système fonctionne, d'autre part si une réparation est possible et si oui à quel prix. Nous allons définir précisément ces notions, ainsi que les méthodes mises en œuvre.

Est-ce que le système fonctionne ? : L'idée est de savoir si compte tenu des résultats déjà obtenus aux différents tests, le système fonctionne. Nous avons vu précédement qu'il y avait deux approches différentes permettant de dire si le système fonctionne : soit un des chemins de succès minimal fonctionne, soit toutes les coupes minimales possèdent au moins un composant en état de marche.

Est-ce que la réparation est possible ? : La première réparation que l'on peut envisager est une réparation que nous qualifierons de « certaine ». Cette réparation consiste à réparer tous les composants en panne d'un chemin de succès minimal. Cette réparation est qualifiée de « certaine » dans le sens où quel que soit le nombre de composants venant de tomber en panne, on est certain que le système repartira puisque l'on aura remis un chemin de succès en état de marche. La seconde réparation que l'on peut envisager est une réparation « quasi-certaine ». Cette réparation consiste à réparer les composants communs à toutes les coupes minimales détectées en panne. Cette réparation est qualifiée de « quasi-certaine » puisque dans le cas très rare où deux composants viennent de tomber en panne, le système peut ne pas repartir.

b) Cardinalité de l'ensemble des chemins de succès minimaux et de l'ensemble des coupes minimales

Pour le moment, pour savoir si le système fonctionne ou si une réparation est possible, on évoque toujours des propriétés sur les coupes minimales et les chemins de succès minimaux. Mais on est alors amené à se demander combien de chemin de succès minimaux et de coupes minimales peut posséder un système de n composants. En réalité, le nombre de chemins de succès minimaux et de coupes minimales peut être exponentiel dans les pires cas. Prenons l'exemple suivant :



3.p composants,
3.2^{p-1} chemins de succès minimaux,
$\sum_{j=1}^{p}(1+3\sum_{k=2}^{j}2^{k-2})$ coupes minimales

On comprend aisément que chercher à savoir si le système fonctionne ou si une réparation est possible, en utilisant les coupes minimales et les chemins de succès minimaux, ne permet pas d'obtenir des procédures polynomiales à coup sûr. Il faut donc réfléchir à une autre méthode.

c) Utilisation du diagramme de fiabilité du système

Le diagramme de fiabilité d'un système à n composants est un graphe à n sommets. En travaillant directement sur cette structure de graphe, pour déterminer si le système fonctionne, si une réparation est possible et si oui à quel coût, on utilise simplement la recherche de chemin et de plus court chemin dans un graphe. Nous avons donc à coup sûr des méthodes polynomiales (disponibles dans [6]).

Mais, comme nous l'avons vu précédement, il existe des systèmes pour lesquels on ne sait pas représenter de diagramme de fiabilité sans dupliquer certains composants. Donc cette nouvelle approche sera limitée à un sous-ensemble spécifique de systèmes.

Les différentes méthodes polynomiales utilisant le diagramme de fiabilité sont présentées en annexes :

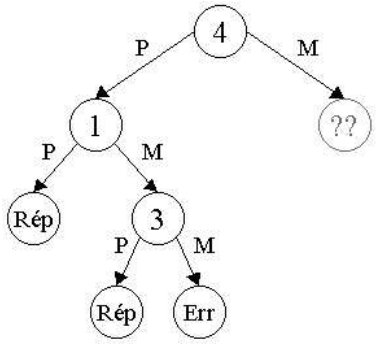
	Test obligatoire avant réparation	Test facultatif avant réparation
Est-ce que le système fonctionne ?	Annexe A	
Une réparation certaine est-elle possible ?	Annexe B	toujours
Une réparation quasi-certaine est-elle possible ?	Annexe C	Annexe D
Quel est le coût de la meilleure réparation certaine possible ?	Annexe E	Annexe F
Quel est le coût de la meilleure réparation quasi-certaine possible ?	Annexe G	Annexe H

II.3 - CONSTRUCTION DE L'ARBRE DE TESTS OPTIMAL PAR UNE EXPLORATION IMPLICITE DE L'ENSEMBLE DES SOLUTIONS

II.3.1 – Présentation des bornes utilisées

a) Idée générale.

Reprenons l'exemple précédent :



Que faire dans le cas où le composant 4 fonctionne ?

En utilisant l'exploration explicite, nous devons construire autant de sous-arbres virtuels qu'il y a de composants encore non testés, puis choisir le meilleur (i.e. celui dont l'espérance mathématique est la plus petite). Or, construire un sous-arbre virtuel est très coûteux et si nous pouvions être sûr à l'avance qu'un sous-arbre virtuel ne sera pas le meilleur, nous n'aurions pas besoin de le calculer.

b) Borne supérieure localement globale et bornes inférieures locales.

Nous allons utilisé :

- une borne supérieure localement globale BS qui est un majorant de l'espérance mathématique du sous-arbre de tests optimal cherché localement.
- une borne inférieure local BI_c pour chaque composant c non testé. Cette borne BI_c représente un minorant de l'espérance mathématique du sous-arbre de tests optimal cherché lorsque l'on force le composant c à en être la racine.

Maintenant, supposons que pour un composant c, on ait $BI_c > BS$. Ceci signifie, que si l'on choisit de mettre c à la racine de notre arbre de tests, l'espérance mathématique sera dans le meilleur des cas, strictement supérieure à l'espérance mathématique de l'arbre de tests optimal. En conclusion, on est sûr que le composant c n'est pas la racine de l'arbre de tests optimal.

c) Algorithme de construction

```
Arbre construireArbreTestsImplicite() {
   // vérification de l'état de fonctionnement du système
   si (système fonctionne) alors
      return Feuille(erreur)
   fsi
   // construction des sous-arbres virtuels
   borneSup = calculerBorneSup()
   meilleurSousArbreVirtuel = null
   si (réparation possible) alors
      meilleurSousArbreVirtuel += Feuille(réparation)
     mettre à jour borneSup
   fsi
   E = ensemble des composants non encore testés
   Pour tout c ∈ E faire
      borneInf = calculerBorneInf(c)
      si (borneInf ≤ borneSup) alors
         mettre c à l'état de fonctionnement « panne »
         arbreGauche = construireArbreTestsImplicite()
         mettre c à l'état de fonctionnement « marche »
         arbreDroit = construireArbreTestsImplicite()
         mettre c à l'état de fonctionnement « indéterminé »
         sousArbre = <arbreGauche,c,arbreDroit>
         si (sousArbre est meilleur que meilleurSousArbreVirtuel) alors
            meilleurSousArbreVirtuel = sousArbre
            mettre à jour borneSup
         fsi
      fsi
   finpour
   return meilleurSousArbreVirtuel
}
```

II.3.2 - Calcul de la borne supérieure localement globale

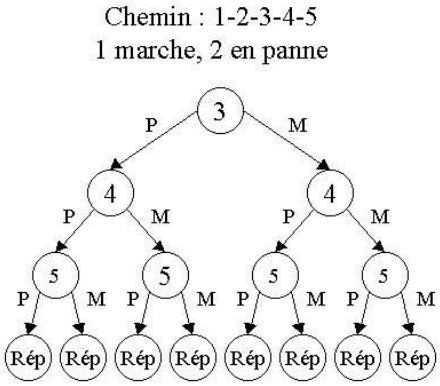
Avant tout, rappelons que, pour une instance donnée, toute solution « faisable » constitue une borne supérieure.

Si on regarde le corps de l'algorithme, on peut remarquer qu'il y a deux calculs différents pour la borne supérieure localement globale : une initialisation (avant la construction des sous-arbres virtuels) et une mise à jour après la construction de chaque sous-arbre virtuel (cette mise à jour consiste simplement à remplacer notre borne supérieure par la valeur du sous-arbre virtuel construit si celle-ci est meilleure). Nous allons détailler l'initialisation qui se base elle aussi sur une solution faisable.

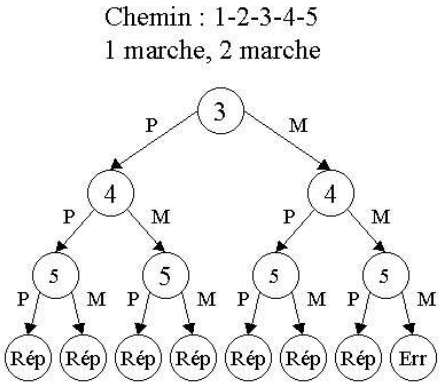
Pour faire repartir le système, il suffit de « remettre en marche » un chemin de succès minimal. On décide de prendre pour borne supérieure la plus petite espérance mathématique parmi les chemins de succès minimaux, où l'on définit l'espérance mathématique d'un chemin comme l'espérance mathématique du coût de la « remise en marche » de ce chemin. Quelque soit les hypothèses de travail, nous supposerons que la remise en marche consiste à tester tous les composants encore non testés et à réparer tous les composants en panne. Il est clair que dans le cas où le test n'est pas obligatoire nous aurons une surestimation. Mais comme il ne s'agit que de calculer une borne supérieure, ceci n'est pas fondamentalement gênant, d'autant plus qu'il ne s'agit que de l'initialisation.

Pourquoi avoir choisi cette borne supérieure ? Pour une question de rapidité de calcul. En effet, on peut calculer l'espérance mathématique d'un chemin sans construire aucun arbre de tests. Il faut distinguer deux cas :

- cas 1 : parmi les composants déjà testés du chemin, il y en a au moins un en panne. Il faut donc tester tous les composants dont l'état est encore indéterminé, puis réparer tous les composants en panne. L'arbre de tests fictif associé pourrait avoir l'allure suivante :



- cas 2 : parmi les composants déjà testés du chemin, tous fonctionnent. Ici, il faut également tester tous les composants dont l'état est encore indéterminé et réparer ceux qui sont en panne. Mais contrairement au cas 1, on est certain qu'à l'issue des tests, tous les composants encore non testés ne pourront pas tous être en état de marche. Il faut donc compenser ce cas de figure virtuel par la réparation d'un composant (le moins cher en l'occurrence). L'arbre de tests fictif associé pourrait avoir l'allure suivante (en supposant que 5 est le composant qui coûte le moins cher à réparer) :



Les espérances mathématiques associées à chaque se calculent directement en utilisant les formules suivantes :

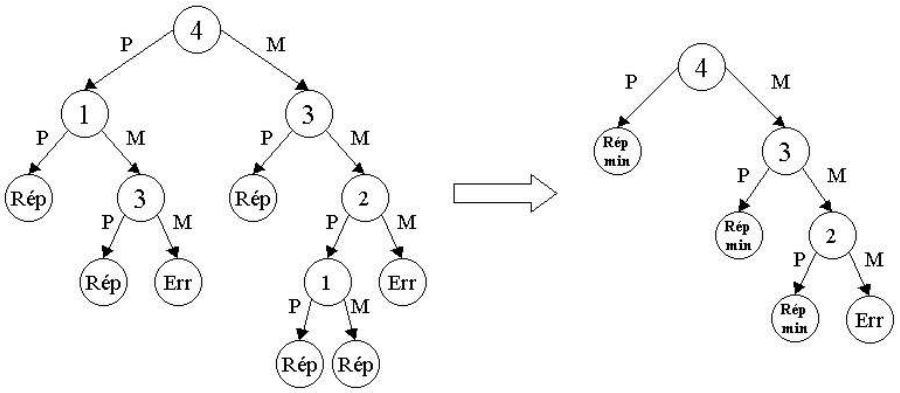
$$\text{cas 1 : EM} = \sum_{\substack{\text{composants} \\ \text{non testés cnt}}} (\text{coutT}_{\text{cnt}} + \text{probaP}_{\text{cnt}} * \text{coutR}_{\text{cnt}}) + \sum_{\substack{\text{composants} \\ \text{en panne cp}}} \text{coutR}_{\text{cp}}$$

$$\text{cas 2 : EM} = \sum_{\substack{\text{composants} \\ \text{non testés cnt}}} (\text{coutT}_{\text{cnt}} + \text{probaP}_{\text{cnt}} * \text{coutR}_{\text{cnt}}) + \prod_{\substack{\text{composant} \\ \text{non testés}}} \text{probaM}_{\text{cnt}} * \underset{\substack{\text{composants} \\ \text{non testés}}}{\text{Min}} (\text{coutR}_{\text{cnt}})$$

II.3.3 - Les bornes inférieures locales.

a) Lorsque le test est obligatoire avant la réparation

Plaçons nous à un instant quelconque de la phase de tests. Soit A* le sous-arbre de tests optimal à cet instant donné. Soit A_d l'arbre obtenu en ne gardant que la branche droite de A* et en remplaçant les branches éliminées par la réparation du composant le moins coûteux.



Passage de A à A_d*

Comme le système est en panne, on est sûr de réparer au moins un composant. Donc, l'espérance mathématique des branches éliminées est au moins égale au coût de réparation du composant le moins coûteux. Donc A_d est clairement une borne inférieure.

Si maintenant, on décide de construire A_d*, le meilleur arbre de tests de type « branche droite », alors par transitivité, A_d* sera également une borne inférieure. Comme précédement, l'avantage est que l'on peut calculer l'espérance mathématique de A_d* sans avoir à construire explicitement cet arbre. En effet, on peut tout d'abord remarquer que tous les composants de A_d* appartiennent à un même chemin car on ne s'intéresse qu'au cas où ces composants marchent et donc à une réparation certaine. On a donc :

$$EM(A_d^*) = \min_{\text{chemins ch}} (EM(A_d^*(ch))$$

Il ne reste plus qu'à définir EM(A_d*(ch)) pour un chemin ch donné. Comme le test est obligatoire avant la réparation, on doit donc tester tous les composants encore non testés. Il faut trouver l'ordre de tests qui minimise l'espérance mathématique. On est dans un cas similaire à celui de Gao lorsqu'il cherche une coupe défectueuse [4]. Il faut donc trier les composants par ordre de coutT/probaP croissant. Si on note [k] le $k^{ème}$ composant testé, l'espérance mathématique est alors :

$$EM(A_d^*(ch)) = \min_{\text{composants c}} (coutR_c) + \sum_{k=1}^{n} c_{[k]} * \prod_{i=1}^{k-1} probaM_{[i]}$$

b) Lorsque le test avant la réparation est facultatif

Dans le cas où le test n'est pas obligatoire, nous n'avons pour le moment pas trouvé de formule donnant directement l'espérance mathématique. Nous sommes donc obligés de construire la branche (temps de calcul en p! où p est le nombre de composants d'un chemin).

On peut aussi envisager d'effectuer une recherche partielle de profondeur 2 (la complexité est alors en $O(n^2)$) et de remplacer les branches coupées par la valeur 0. Cette

méthode de type « Look Ahead » est généralisée et expliquée en détails dans la partie consacrée à la résolution approchée.

II.3.4 – Complexité en temps du calcul des bornes

On considère un système à n composants et p chemins.

a) Pour la borne supérieure localement globale

Le calcul de l'espérance mathématique d'un chemin se fait dans le pire des cas en O(n). Et comme on a p chemins, le calcul de la borne supérieure se fera en O(n.p)

b) Pour la borne inférieure locale.

Sous les hypothèses de travail n°1, le calcul de la borne inférieure local se fait également en O(n.p), pour les mêmes raisons que précédemment. Sous les hypothèses de travail n°2, selon l'approche choisie, le calcul se fera en O(n!) ou en $O(n^2)$

II.3.5 – Ordre de construction des nœuds virtuels

Nous avons vu que la valeur de la borne supérieure localement globale était réactualisée après chaque calcul d'un nœud virtuel. Donc, il est légitime de penser que l'ordre de construction des nœuds virtuels a une importance. Cette piste, bien qu'elle n'ait pas été considérée comme prioritaire lors du stage de DEA, méritera qu'on s'y attarde lors de futurs travaux.

II.4 – RÉSOLUTION D'UN PROBLÈME PLUS GÉNÉRAL : APPROCHE DE TYPE PROGRAMMATION DYNAMIQUE

II.4.1 – Définition du problème étendu

a) Problématique.

Supposons que notre système vient de tomber en panne pour la première fois. Nous ne connaissons, a priori, l'état de fonctionnement d'aucun composant. Lors de la phase de tests, nous allons obtenir des connaissances sur l'état de fonctionnement de certains composants. Mais comme notre objectif est de faire repartir le système au moindre coût, il se peut que, dans la phase de réparation, on ne répare pas tous les composants testés en panne lors de la phase de tests. En conséquence, lors de la prochaine panne, il se peut que l'on connaisse déjà l'état de panne de certains composants. Or l'arbre de tests utilisé lors de la 1ère réparation n'est pas forcément optimal compte tenu de ces connaissances. Il faut donc recalculer un nouvel arbre de tests.

L'algorithme de construction d'un arbre de tests est exponentiel. Il ne semble donc pas raisonnable de recalculer un arbre de tests à chaque nouvelle panne. L'idée est donc de calculer directement tous les arbres de tests possibles, chaque arbre correspondant à une configuration du système. Comme chaque composant est soit en panne, soit en marche, soit dans un état indéterminé, il y a 3^n configurations possibles pour notre système.

b) Parenthèse technique : numérotation des configurations du système

On va attribuer un numéro unique à chaque configuration supposée connue du système. Pour ce faire, on crée un nombre de n chiffres en base 3 en utilisant les états de fonctionnements de nos n composants : 0 pour panne, 1 pour marche, 2 pour indéterminé. Puis, pour plus de facilité, on convertit ce nombre en base 10.

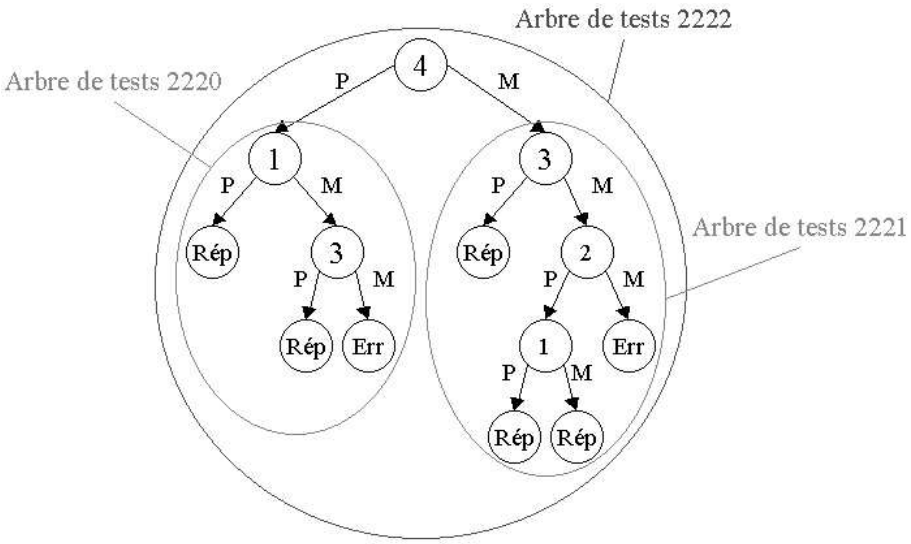
Ainsi, la configuration 0 correspond au fait que tous les composants du système sont en panne et la configuration 3^n-1 correspond au fait que tous les composants du système sont dans un état indéterminé.

De manière naturelle, l'arbre de tests n°i que nous calculerons correspondra à la configuration n°i du système.

II.4.2 – Mise sous forme récursive

a) Construction des arbres de tests

Pour construire les 3^n arbres de tests, on ne va bien entendu pas utiliser 3^n fois la méthode exposée précédemment. On va utiliser la programmation dynamique. Pour comprendre le principe, regardons le schéma suivant :



Cet exemple montre que l'arbre de test n°2222_3 (i.e. tous les composants sont dans un état indéterminé) est en réalité constitué d'un composant à tester (la racine) et des deux arbres de tests correspondant aux deux configurations possibles après le test de la racine (2220_3 : le composant 4 en panne ; 2221_3 : le composant 4 en état de marche).

Ensuite, on peut remarquer que, pour construire l'arbre de test n°i, on ne fera appel qu'à des configurations de n° strictement plus petits que i. En effet, seule la valeur de l'élément à tester change. Et comme on ne teste que des composants indéterminés, cette valeur est 2 et elle ne peut que diminuer après le test (0 ou 1).

On peut donc aisément utiliser la programmation dynamique et construire les configurations par ordre croissant de n°.

b) Nœuds virtuels et nœuds définitifs

Pour déterminer l'arbre de tests n°i, on va, comme pour la méthode précédente, utiliser des sous-arbres virtuels :

- Un sous-arbre « réparation ». Comme pour la méthode précédente, ce sous-arbre n'existe que si la réparation du système est possible au vu des tests déjà effectués.
- Des sous-arbre de tests. On aura autant de sous-arbres que de composants encore non testés.

Enfin, comme précédemment, on évalue l'espérance mathématique de chaque sous-arbre et on garde le meilleur.

c) Implémentation

```
construireArbresDeTestsDynamique() {
   Pour tout i de 0 à 3^n-1 faire
      // construction de l'arbre de tests n°i
      placer le système dans la configuration n°i
      si (système fonctionne) alors
         arbreTests[i]=Feuille(erreur)
      sinon
         listeSousArbresVirtuels = vide
         si (réparation possible) alors
```

```
            listeSousArbresVirtuels += Feuille(réparation)
         fsi
         E = ensemble des composants non encore testés
         Pour tout c ∈ E faire
            k ← n° de configuration tel que c soit en panne
            arbreGauche = arbreTests[k]
            k ← n° de configuration tel que c soit en marche
            arbreDroit = arbreTests[k]
            listeSousArbresVirtuels += <arbreGauche,c,arbreDroit>
         finpour
         arbreTests[i] = meilleurSousArbreVirtuel(listeSousArbresVirtuels)
      fsi
   finpour
}
```

II.4.3 - Complexités

a) Complexité temporelle de l'algorithme

Soit un système complexe à n composants. On doit calculer 3^n arbres de tests. Pour chaque calcul, on doit construire autant d'arbres virtuels que de composants non testés. Donc la complexité de la méthode est en $O(n.3^n)$.

b) Complexité spatiale de l'algorithme

Soit un système complexe à n composants. Nous devons stocker 3^n arbres de tests. Quand on sait qu'un arbre de tests peut avoir jusqu'à 2^n-1 éléments, on peut se dire que l'espace de stockage nécessaire va être immense. Mais en fait, comme on l'a vu, un arbre est composé d'une racine et de deux sous-arbres plus « petits ». Donc, on ne stocke que la racine et deux pointeurs vers les sous-arbres. Donc, si tous les arbres sont stockés de cette manière, on voit que l'on a besoin de place seulement pour stocker 3^n racines et 2.3^n pointeurs.

II.4.4 - Avantage et inconvénient de la méthode

L'avantage incontestable de cette méthode vient du fait que tous les calculs peuvent être effectués une fois pour toutes avant même que le système ne soit mis en marche. Après, à chaque nouvelle panne, il suffit de déterminer dans quelle configuration le système se trouve pour avoir « instantanément » l'arbre optimal de tests correspondant. Cette méthode est donc idéale du point de vue de l'utilisateur, qui, quelle que soit la situation, obtient une solution optimale instantanément.

Mais, à cet avantage, s'oppose un inconvénient de taille. Le temps de calcul et l'espace de stockage nécessaires sont exponentiels. Cette méthode sera donc limitée à des systèmes de « petite taille » (au maximum 15 composants).

III - Heuristique de résolution approchée

III.1 – PRÉAMBULE : RAPPEL SUR LES MÉTHODES DE RÉSOLUTION APPROCHÉE

En Recherche Opérationnelle, les principales familles de méthodes de résolution approchée sont :

- Les approches par construction.
- Les approches par voisinage.
- Les approches par décomposition, qui sont de deux types :
    - Déviance de schémas optimaux existants : il s'agit d'élaborer une méthode approchée par transformation d'une méthode exacte existante.
    - Conception d'autres types de décomposition qui ne reposent pas sur des théorèmes assurant l'optimalité

Dans cette partie, nous allons présenter une approche par décomposition obtenu par transformation de l'exploration explicite présentée précédemment.

III.2 – MÉTHODE DE RÉSOLUTION APPROCHÉE UTILISANT LE PRINCIPE DU LOOK AHEAD

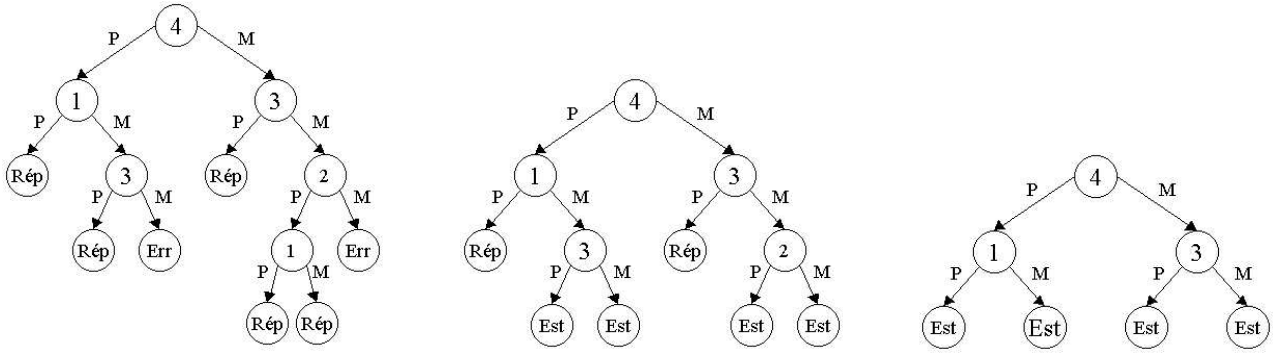
III.2.1 - Problématique.

Nous avons vu que les méthodes de résolution exactes, bien qu'elles présentent l'avantage de résoudre le problème de manière optimale, ont l'inconvénient d'être de complexité exponentielle. L'utilisation de telles méthodes est donc limitée aux systèmes de petites tailles (au maximum une quinzaine de composants) et il est indispensable de chercher une méthode de résolution approchée pour les systèmes plus conséquents.

III.2.2 - Le principe du « Look Ahead »

Pour comprendre ce que fait réellement le look ahead, introduisons tout d'abord la notion d'arbre de tests de profondeur k. On appelle « arbre de tests de profondeur k » un arbre de tests dont la profondeur maximale est majorée par k. Il s'agit donc de couper toutes les branches situées à une profondeur supérieures à k et de les remplacer par une feuille « estimation ». Cette feuille, comme son nom l'indique, est une estimation de l'espérance mathématique de la branche coupée.

L'exemple suivant montre un arbre de tests et les arbres de tests de profondeur 3 et 2 que l'on obtient à partir de lui.



Arbres de tests complet, de profondeur 3, et de profondeur 2

Le principe du look ahead peut s'exprimer comme suit : « la racine de l'arbre de tests optimal et la racine du meilleur arbre de tests de profondeur k ont d'autant plus de chance d'être identiques que k est grand et que la fonction d'estimation est performante »

On comprend donc implicitement que le Look Ahead de profondeur k sera utilisé pour calculer les tests à effectuer au fur et à mesure et non pour déterminer un arbre de tests global. Bien entendu, cet arbre de tests global sera toutefois construit lors des expérimentations, et ce, pour valider la méthode.

III.2.3 - La fonction d'estimation

Comme nous l'avons mentionné précédemment, la performance de la fonction d'estimation joue un rôle très important. Il faut donc que cette fonction renvoie une valeur très proche de l'espérance mathématique de la branche coupée.

Comme fonction d'estimation, nous utiliserons la borne supérieure localement globale présentée dans la partie consacrée à la résolution exacte par exploration implicite.

III.2.4 - L'algorithme du « Look Ahead »

L'algorithme utilisé est très proche de celui utilisé dans la première méthode de résolution exacte. La seule différence vient du fait que l'on mémorise à quelle profondeur de l'arbre on se trouve.

```
Arbre LookAhead(entier k) {
   // vérification de l'état de fonctionnement du système
   si (profondeur==0) alors
      return Feuille(estimation)
   fsi
   si (système fonctionne) alors
      return Feuille(erreur)
   fsi
   // construction des sous-arbres virtuels
   listeSousArbresVirtuels = vide
   si (réparation possible) alors
      listeSousArbresVirtuels += Feuille(réparation)
   fsi
   E = ensemble des composants non encore testés
   Pour tout c ∈ E faire
      mettre c à l'état de fonctionnement « panne »
      arbreGauche = construireArbreTests(profondeur-1)
      mettre c à l'état de fonctionnement « marche »
      arbreDroit = construireArbreTests(profondeur-1)
      listeSousArbresVirtuels += <arbreGauche,c,arbreDroit>
      mettre c à l'état de fonctionnement « indéterminé »
   finpour
   // choix d'un sous-arbre définitif
   return meilleurSousArbreVirtuel(listeSousArbresVirtuels)
}
```

L'algorithme pour construire l'arbre de tests en entier en se basant sur le Look Ahead est le suivant :

```
Arbre construireArbreLA(entier k) {
   // construction de la racine
   arbreLookAhead = LookAhead(k)
   si (arbreLookAhead = Réparation ou Erreur) alors
      return arbreLookAhead
   sinon
      c = racine(arbreLookAhead)
      mettre c à l'état de fonctionnement « panne »
      arbreGauche = construireArbreLA(k)
      mettre c à l'état de fonctionnement « marche »
      arbreDroit = construireArbreLA(k)
      mettre c à l'état de fonctionnement « indéterminé »
   fsi
}
```

III.3 – RECHERCHE D'UNE BORNE INFÉRIEURE GLOBALE DE L'ESPÉRANCE DU COÛT DE REMISE EN MARCHE DU SYSTÈME

III.3.1 – Problématique et objectif

Le Look Ahead est une méthode de résolution approchée. Il faut donc valider cette approche en comparant les résultats obtenus avec les résultats optimaux. Mais, comme nous l'avons déjà évoqué précédemment, les méthodes de résolution exactes ne peuvent résoudre que de petits systèmes. On ne peut donc évaluer les performances du Look Ahead que sur de petites instances. Or, cette méthode est conçue pour être utilisée sur de grandes instances, et c'est justement sur ces grandes instances que l'on souhaiterait la valider.

Nous allons donc utiliser une autre approche de validation : la borne inférieure. En effet, en utilisant une borne inférieure globale de l'espérance mathématique du coût de remise en marche du système, on peut alors majorer l'écart entre la solution approchée et la solution optimale. L'objectif est de trouver une borne inférieure offrant le meilleur compromis temps / pertinence (une borne inférieure sera d'autant plus pertinente que sa valeur sera proche de la valeur optimale).

III.3.2 – Quelle borne inférieure utiliser ?

Dans la partie consacrée à la résolution exacte par exploration implicite, nous avions présenté une borne inférieure possible. En raison d'un grand nombre d'appels, le temps de calcul était privilégié par rapport à la pertinence. Mais ici, la pertinence nous intéresse plus que la rapidité. C'est pourquoi, nous allons utiliser une autre borne inférieure.

Dans la partie précédente, nous avons introduit la notion d'arbre de tests de profondeur k. L'idée était de couper toutes les branches situées à une profondeur supérieure à k et de les remplacer par une feuille « estimation ». Si pour fonction d'estimation, on utilise une borne inférieure de la branche coupée et non plus une borne supérieure, alors l'arbre obtenu constitue une borne inférieure globale.

Pour obtenir une borne inférieure globale, il suffit donc de réutiliser l'algorithme du Look Ahead en utilisant comme fonction d'estimation un calcul de borne inférieure (ici, on utilisera celui présenté dans la partie de l'exploration implicite).

Comme précédemment, la borne inférieure sera d'autant plus pertinente que la profondeur k sera grande.

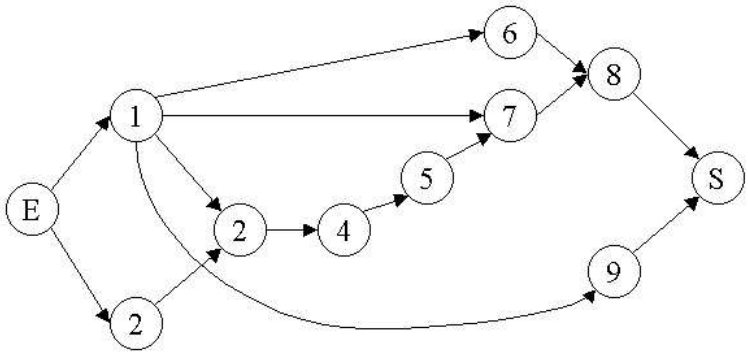
IV - Tests

IV.1 – PLAN D'EXPÉRIMENTATION

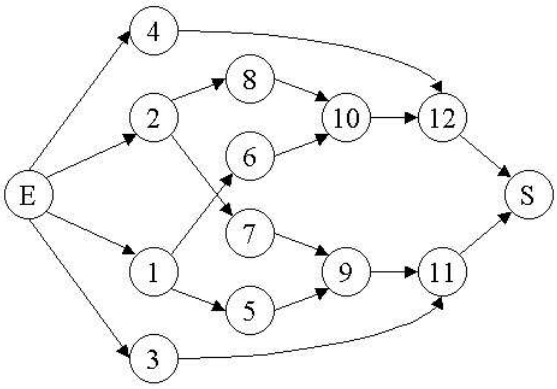
IV.1.1 – Les systèmes utilisés

Bien que nous ayons testé nos différentes méthodes sur de nombreux systèmes, allant du plus simple (systèmes séries, systèmes parallèles, …) au plus compliqués (systèmes complexes pour lesquels les coupes minimales ne sont pas indépendantes), nous ne présenterons les résultats que pour 3 systèmes concrets issus de [3]. Ici, nous ne présenterons que les diagrammes de fiabilité obtenus après une analyse complète de ces systèmes. Pour plus de précisions, se reporter à [3]

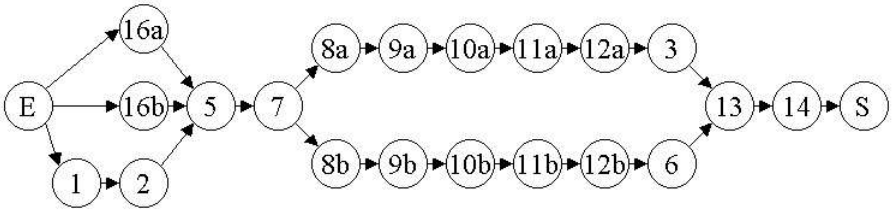
a) Un système de protection contre les survitesses d'une turbine



b) Une alimentation électrique



c) Chaîne cinématique de levage d'un pont roulant



IV.1.2 – Coûts de test, coûts de réparation et probabilités de panne.

Pour chaque composant, nous allons générer aléatoirement son coût de test, son coût de réparation et sa probabilité de panne selon des lois uniformes. Pour les expériences données ici, nous considérons que ces trois données sont indépendantes. Dans la pratique, ce n'est souvent pas le cas. En effet, un composant très cher à réparer est souvent très fiable, et, à l'inverse, un composant tombant fréquemment en panne ne coûtera pas trop cher à remettre en marche.

IV.1.3 – Critères retenus

Pour chaque système, nous avons généré plusieurs jeux de données. Pour chaque jeu de données, nous avons calculé :

- la fiabilité du système,
- l'espérance mathématique optimale de remise en marche du système obtenue par la programmation dynamique (cette méthode étant la plus rapide des 3),
- l'espérance mathématique approchée obtenue par la technique du Look Ahead Estimation de profondeur k (les valeurs de k varient selon l'exemple traité),
- la meilleure borne inférieure globale obtenue par la technique du Look Ahead Borne Inférieure de profondeur k (les valeurs de k varient selon l'exemple traité).

Enfin, pour chaque système, nous tracerons les courbes moyennes de la méthode approchée et de la borne inférieure globale en fonction de k.

IV.2 – RÉSULTATS ET COMMENTAIRES

IV.2.1 – Synthèse de résultats

Pour chaque système complexe considéré, nous avons généré 50 jeux de données différents. Les résultats détaillés sont donnés en annexes (annexes J, K et L). Nous présentons, ici, la synthèse de ces résultats.

a) La programmation dynamique

Pour le système de protection contre la survitesse d'une turbine, le temps de calcul de la méthode est inférieure à une seconde. Pour l'alimentation électrique, le temps de calcul va de 9.78 secondes à 17.85 secondes. Enfin, pour la chaîne cinématique de levage d'un pont roulant, le calcul a toujours été interrompu par manque de mémoire avant d'atteindre 2 minutes.

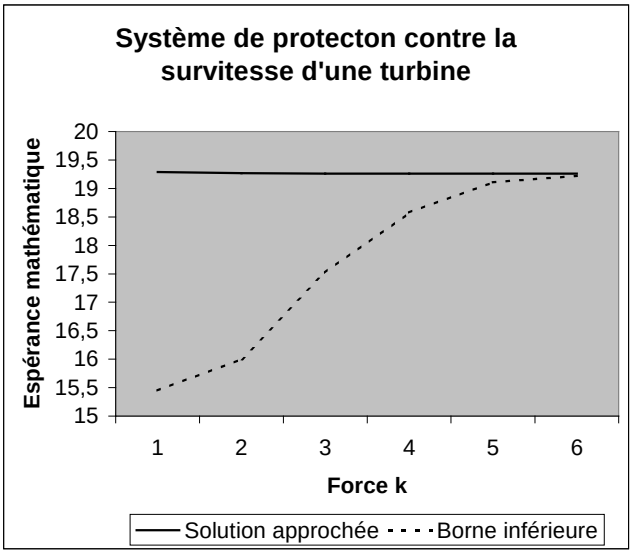
b) La résolution approchée par le Look Ahead

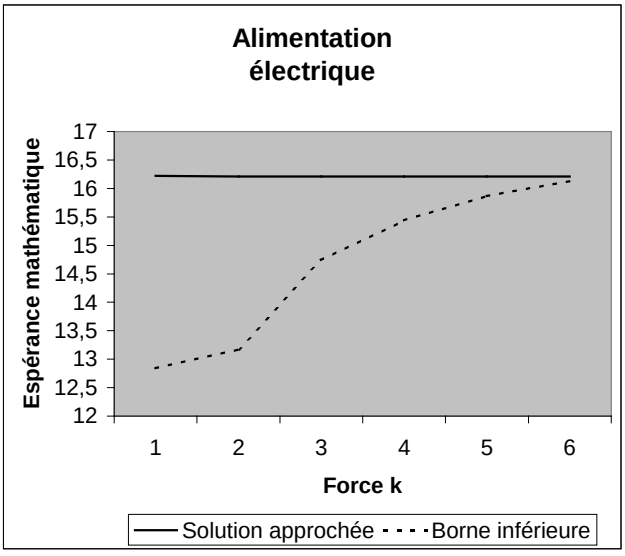
Le premier résultat qui ressort de nos tests, c'est que, dans la majorité des cas (4 exceptions seulement sur l'ensemble des tests), le résultat fourni par le Look Ahead de profondeur k+1 est plus pertinent que le résultat fourni par le Look Ahead de profondeur k. Dans les cas exceptionnels où cette propriété n'est pas vérifié, l'écart entre les deux espérances mathématiques est majoré par 0,003 %.

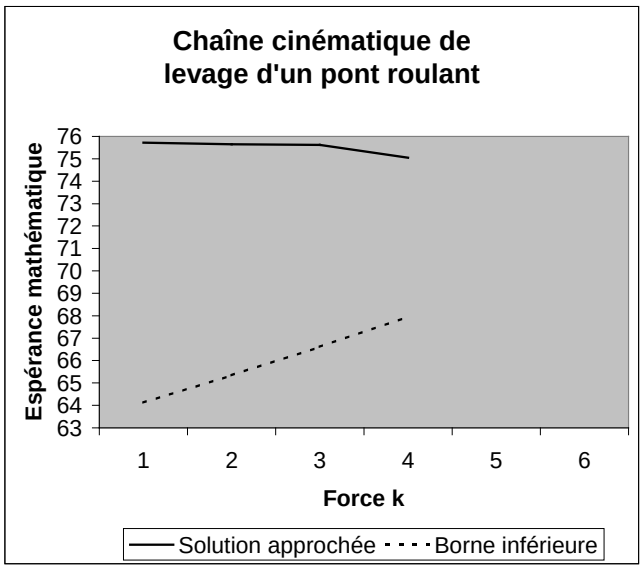
Au point important : à chaque fois que la méthode exacte de programmation dynamique a aboutit, le résultat obtenu était identique à celui obtenu par le Look Ahead.

Lorsque programmation dynamique n'était pas utilisable (cas de la chaîne cinématique de levage), la majoration de l'erreur moyenne est de xx,xx % et la majoration maximale de yy,yy %. Nous avons également remarqué une corrélation entre la majoration de l'erreur et la fiabilité du système.

c) Evolution de la méthode approchée et de la borne inférieure en fonction de k







IV.2.2 – Commentaires

a) La programmation dynamique

Il apparaît clairement la programmation dynamique est essentiellement limitée par l'espace mémoire. Mais ce qui ressort des tests, c'est que lorsque la programmation dynamique est utilisable (systèmes de moins de 15 composants), cette dernière est très rapide. De plus, rappelons que cette méthode permet de résoudre toutes les configurations du système une fois pour toutes.

b) La résolution approchée par le Look Ahead

Le fait que l'espérance mathématique obtenue par la méthode de résolution approchée soit, dans la quasi totalité des cas, une fonction croissante de k n'est pas un résultat anodin. Le fait que cette propriété soit vérifiée tend à prouver que la fonction d'estimation utilisée est pertinente.

Concernant la chaîne cinématique de levage, la majoration de l'erreur dépasse dans le pire cas les xx %. Comment doit-on interpréter ce résultat. Tout d'abord, on peut remarquer que nous n'avons pas pu utiliser de grandes valeurs de k en raison du grand nombre de composants. Or, si on regarde les courbes d'évolution de la méthode approchée et de la borne inférieure en fonction de k, on peut extrapoler que la borne inférieure n'est pas encore prête à converger quand k=5, et qu'en revanche, le Look Ahead semble vite se stabiliser (on rappelle que la limite du Look Ahead correspond à la solution optimale).

CONCLUSION ET PERSPECTIVES

Durant ce stage de DEA, nous avons eu l'occasion de concevoir trois méthodes de résolution exacte. La première méthode (exploration explicite) est une méthode servant de base. Etant explicite, elle n'est, dans la pratique, presque jamais utilisable, même sur des systèmes de taille raisonnable. La seconde méthode (exploration implicite de type PSE) reprend le principe de la première méthode, mais n'effectue pas les calculs « inutiles ». Cette méthode, bien que plus rapide, reste cependant limitée à des systèmes de taille raisonnable. De plus, sa performance dépend fortement des coûts de test, de réparation et des probabilités de pannes des composants du système. Plus les coûts et les probabilités seront homogènes, moins elle sera efficace. Enfin, la troisième et dernière méthode conçue (exploration implicite de type Programmation Dynamique) a consisté à généralisé le problème en calculant les arbres de tests optimaux pour tous les états possibles du système. Contrairement aux deux première méthodes, cette méthode n'est pas limitée par le temps mais par l'espace mémoire. En effet, au delà de 15 composants, la méthode n'a plus assez d'espace mémoire pour effectuer ses calculs. En revanche, le point fort de cette méthode, c'est d'une part sa rapidité (moins de deux minutes lorsqu'elle est utilisable), et d'autre part le fait que l'on résolve en une seule fois tous les états possibles du système.

Nous avons également travailler sur une méthode de résolution approchée et sur une borne inférieure globale basées sur le principe du Look Ahead. L'idée générale de ces méthodes était de stopper la recherche dès lors qu'une profondeur maximale fixée à l'avance était atteinte et de remplacer la branche inexplorée soit par une estimation de cette branche, soit par une borne inférieure locale. Le premier point fort de ces méthodes vient du fait que l'on peut accélérer la résolution (donc le temps de calcul nécessaire) en jouant sur la profondeur maximale fixée. Mais surtout, à chaque fois que nous avons pu calculer une solution exacte, il s'est avéré que la solution approchée fournie par le Look Ahead était, elle aussi, exacte. Malheureusement, pour les systèmes de grande taille, aucune méthode exacte n'est utilisable et seule la borne inférieure globale permet de majorer l'erreur de la solution approchée. Et dans ce cas, la majoration de l'erreur peut atteindre 33%.

Nous avons vu qu'il y avait corrélation entre la majoration de l'erreur et la fiabilité du système. Pour cette raison, il y a de fortes chances pour que les « mauvais résultats » soient le fait d'une borne inférieure peu pertinente. Mais, nous ne pouvons pas le prouver. C'est pourquoi, il faudrait, dans de futurs travaux, développer d'autres méthodes approchées radicalement différentes de celle exposée ici, puis confronter les différentes méthodes approchées. Une piste pourrait consister à regrouper les éléments du système complexe en macro-éléments, macro-éléments sur lesquels nous pourrions appliquer la méthode de résolution exacte de type programmation dynamique et ainsi obtenir les coûts et la probabilité de panne de chaque macro-élément. Il ne resterait plus qu'à appliquer une méthode de résolution exacte sur le système formé de ces macro-éléments.

Enfin, notre méthode consiste à faire repartir, au moindre coût, un système tombé en panne. Comme nous l'avons déjà dit précédemment, ceci induit que l'on ne réparer pas forcément tous les composants en panne. Il faudrait maintenant essayer d'insérer nos méthodes sur un horizon à moyen et long terme, où le nombre de fois où le système tombe en panne a son importance. Il faudrait alors revoir le but fixé qui ne serait plus faire repartir au moindre coût le système, mais peut-être faire repartir le système et lui assurer une fiabilité minimum au moindre coût.

V - Annexes

ANNEXE A : EST-CE QUE LE SYSTÈME FONCTIONNE ?

On sait que le système fonctionne si et seulement si au moins un chemin de succès fonctionne. Il s'agit donc de prouver l'existence ou la non-existence d'un chemin de succès dont tous les composants fonctionne. En utilisant le diagramme de fiabilité, ce problème revient simplement à prouver qu'il existe un chemin dont tous les composants fonctionnent allant de l'entrée E à la sortie S.

Pour ce faire, on valide les arcs sortants de tout composant en état de marche, puis on cherche l'existence d'un chemin allant de E à S. Si un tel chemin existe, alors le système fonctionne. Réciproquement, si le système fonctionne, on est sûr qu'un tel chemin sera trouvé.

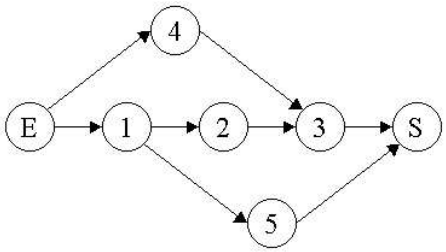
ANNEXE B : UNE RÉPARATION CERTAINE EST-ELLE POSSIBLE ?

Le problème est de savoir si tous les composants d'un des chemins ont été testés. Pour ce faire, on va travailler sur le graphe associé à notre système complexe et on va simplement valider les arcs sortants pour tout composant testé ou pour tout composant dont on sait qu'il fonctionne. Ainsi, s'il existe un chemin de E vers S, c'est la preuve qu'un chemin de succès est réparable.

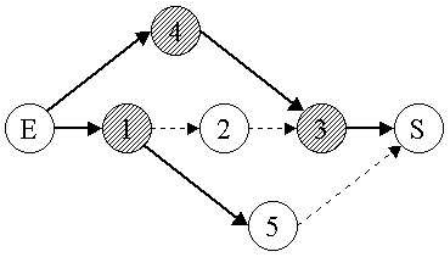
Comme il s'agit d'une recherche de chemin dans un graphe, cette méthode est polynomiale.

Exemple :

Considérons le système complexe dont le graphe associé est le suivant :



Supposons qu'à l'instant t, on ait déjà testé les composants 1,3 et 4.



Il existe un chemin de E vers S : la réparation est possible.

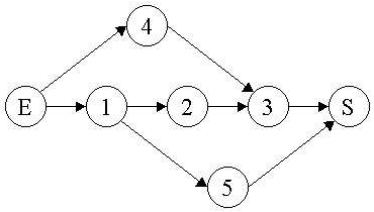
ANNEXE C : UNE RÉPARATION QUASI-CERTAINE EST-ELLE POSSIBLE SACHANT QUE LE TEST EST OBLIGATOIRE AVANT LA RÉPARATION ?

Cette fois, on veut savoir si au moins un composant par chemin est testé et en panne. On considère toujours le graphe associé au système complexe. Mais contrairement au cas précédent, on va faire le raisonnement inverse. Ici, on considère qu'au départ, tous les arcs du graphe sont validés (et donc qu'il existe un chemin de E vers S). Puis pour tout composant testé en panne, on invalide les arcs sortant du sommet correspondant. Si on ne parvient plus à trouver un chemin de E vers S, c'est que tout chemin de succès comporte au moins un composant testé en panne (car dans le cas contraire, un chemin existerait de E à S).

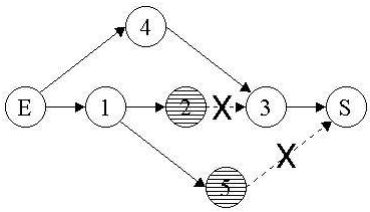
Comme précédement, recherche de chemin → méthode polynomiale.

Exemple :

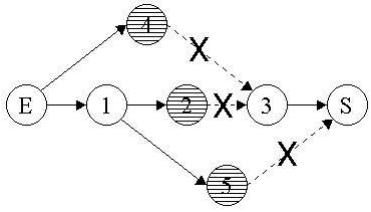
Considérons le système complexe dont le graphe associé est le suivant :



Supposons qu'à l'instant t, on ait testé 2 et 5 en panne. On a donc :



On voit qu'il existe encore un chemin : « E-4-3-S ». Supposons maintenant que 4 soit également testé en panne :



Cette fois, il n'existe plus de chemin de E vers S. Donc chaque chemin a au moins un composant testé en panne.

ANNEXE D : UNE RÉPARATION QUASI-CERTAINE EST-ELLE POSSIBLE SACHANT QUE LE TEST N'EST PAS OBLIGATOIRE AVANT LA RÉPARATION ?

Contrairement au cas précédent, on se moque de savoir si le composant en panne est testé ou non. Donc on utilise la même méthode que celle exposée dans la fiche 3.a, sauf qu'au lieu d'invalider les arcs pour les composants testés en panne, on invalide les arcs pour les composants dont on sait qu'ils sont en panne (testé ou non).

ANNEXE E : COMMENT DÉTERMINER, EN TEMPS POLYNOMIAL, LA MEILLEURE RÉPARATION CERTAINE POSSIBLE, LORSQUE LE TEST EST OBLIGATOIRE AVANT LA RÉPARATION ?

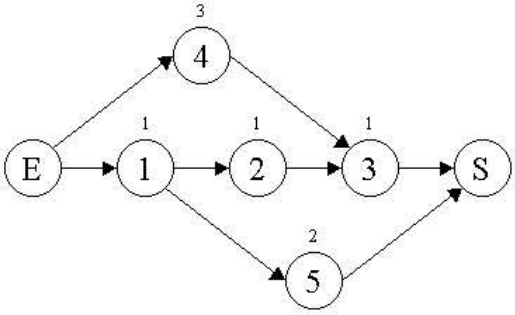
Comme nous l'avons déjà dit précédemment, lorsqu'une réparation certaine est possible, il se peut que plusieurs chemins soient candidat à la réparation. Il faut donc trouver, parmi tous ces chemins candidats, celui dont la réparation coûtera le moins cher.

Une fois encore, on travaille sur le graphe associé au système complexe. On valide tous les arcs sortants des composants déjà testés. On attribue ensuite une valeur à chaque sommet : 0 si le composant fonctionne et son coût de réparation sinon. Et on cherche alors le chemin de coût minimum allant de E vers S.

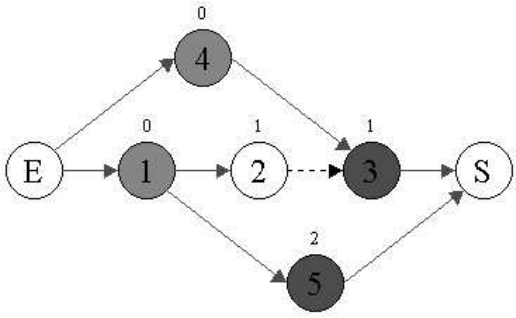
La recherche d'un chemin de moindre coût dans un graphe est, elle aussi, polynomiale.

Exemple :

Considérons le système complexe dont le graphe associé est le suivant :



Supposons qu'à l'instant t, on ait testé 1, 3, 4 et 5. On suppose aussi que 1 et 4 fonctionnent. On a donc :



Dans cette situation, deux chemins sont candidats à la réparation : « E-4-3-S » et « E-1-5-S ». La recherche du plus court chemin nous dira de réparer « E-4-3-S » qui ne coûte que 1 à réparer contre 2 au chemin « E-1-5-S ».

ANNEXE F : COMMENT DÉTERMINER, EN TEMPS POLYNOMIAL, LA MEILLEURE RÉPARATION CERTAINE POSSIBLE, LORSQUE LE TEST N'EST PAS OBLIGATOIRE AVANT LA RÉPARATION ?

Nous avons vu que dans le cadre des hypothèses de travail n°2, la réparation déterministe était toujours possible. Ceci tient au fait que l'on peut réparer les composants indéterminés d'un chemin sans les tester, même s'ils marchent. Du coup, tout chemin est candidat à la réparation.

La méthode est sensiblement la même que précédemment. Cette fois tous les arcs sont validés (vu que tout chemin est candidat à la réparation) et la valeur associée à un composant est : 0 s'il fonctionne, son coût de réparation sinon.

ANNEXE G : COMMENT DÉTERMINER, EN TEMPS POLYNOMIAL, LA MEILLEURE RÉPARATION QUASI-CERTAINE POSSIBLE, LORSQUE LE TEST EST OBLIGATOIRE AVANT LA RÉPARATION ?

Nous abordons ici la partie la plus délicate. Nous allons procéder par étapes successives.

1. Quand on sait qu'une réparation quasi-certaine est possible, alors le composant qui vient de tomber en panne appartient à l'ensemble des composants déjà testés en panne. Pourquoi ? Il existe un chemin tel que ce composant soit le seul en panne. Or, si on reprend la méthode de la fiche 3.a en supposant que ce composant n'est pas testé, on comprend aisément que le chemin en question sera validé et donc que la réparation quasi-certaine ne sera pas possible.

2. On sait aussi que chaque chemin possède au moins un composant en panne, et qu'il existe au moins un chemin possédant (pour le moment) exactement un composant en panne. Donc, d'après la propriété établie en 1., si on répare les composants testés en panne présents dans un chemin possédant (pour le moment) exactement un composant en panne, on est sûr de réparer le composant qui vient de tomber en panne. Cette condition suffisante est également une condition nécessaire.

3. Il faut donc déterminer quels sont les composants présents dans un chemin contenant (pour le moment) exactement un composant testé en panne. La méthode est assez simple. On valide tous les arcs du graphe associé au système. Puis on invalide tous les arcs partant d'un composant testé en panne. Un composant est présent dans un chemin contenant exactement un composant testé en panne, dès lors que le supposer en état de marche permet de trouver un chemin de E vers S. Il est donc très facile d'isoler ces composants à réparer.

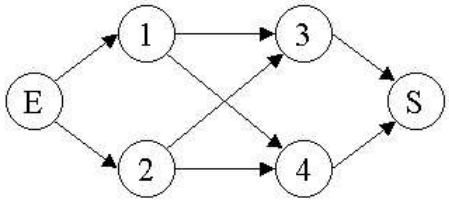
ANNEXE H : COMMENT DÉTERMINER, EN TEMPS POLYNOMIAL, LA MEILLEURE RÉPARATION QUASI-CERTAINE POSSIBLE, LORSQUE LE TEST N'EST PAS OBLIGATOIRE AVANT LA RÉPARATION ?

1. Quand on sait qu'une réparation quasi-certaine est possible, alors le composant qui vient de tomber en panne appartient à l'ensemble des composants en panne (testés ou non). Pourquoi ? Il existe un chemin tel que ce composant soit le seul en panne. Or, si on reprend la méthode de la fiche 3.b en supposant que l'état de fonctionnement de ce composant soit encore indéterminé, on comprend aisément que le chemin en question sera validé et donc que la réparation quasi-certaine ne sera pas possible.

2. On sait aussi que chaque chemin possède au moins un composant en panne, et qu'il existe au moins un chemin possédant (pour le moment) exactement un composant en panne. Donc, d'après la propriété établie en 1., si on répare les composants en panne présents dans un chemin possédant (pour le moment) exactement un composant en panne, on est sûr de réparer le composant qui vient de tomber en panne.

3. Il faut donc déterminer quels sont les composants présents dans un chemin contenant (pour le moment) exactement un composant en panne. La méthode est assez simple. On valide tous les arcs du graphe associé au système. Puis on invalide tous les arcs partant d'un composant en panne. Un composant est présent dans un chemin contenant exactement un composant en panne, dès lors que le supposer en état de marche permet de trouver un chemin de E vers S. Il est donc très facile d'isoler ces composants à réparer.

ANNEXE I : EST-CE QUE LA SEULE UTILISATION DES COUPES MINIMALES ASSURE L'OPTIMALITÉ DE LA SOLUTION TROUVÉE ?



	Coût	Proba
1	1	p
2	x	p
3	1	p
4	x	p

Dans cet exemple, tous les composants ont la même probabilité de panne p. Les coupes minimales sont C_1={1,2} et C_2={3,4} (on remarquera au passage qu'elles sont indépendantes). En appliquant la méthode, on va commencer par examiner la coupe C_1, et plus précisément le composant 1. Deux cas de figures se présentent : le composant 1 marche ou ne marche pas. S'il marche, la procédure s'arrête et l'ensemble de composants cherché est {3,4}. Si le composant 1 est en panne, on doit tester le composant 2 et quel que soit le résultat de ce test, on s'arrête (soit parce que la coupe C_1 est en panne, soit que l'on sait par déduction que la coupe C_2 est en panne). Donc l'espérance mathématique de cette méthode est EM = p + (1-p).x

Or on peut remarquer qu'en testant systématiquement les composants 1 et 3, on peut faire repartir le système à coup sûr. Ce test systématique induit une espérance mathématique EM = 2.

Il apparaît assez évident qu'il existe des valeurs de p et de x tels que 2 < p + (1-p)x.

BIBLIOGRAPHIE

[1] E. E. Lewis, 1987. *Introduction to Reliability Engineering*, John Wiley & sons, New York

[2] J. A. Nachlas, S. R. and B. A. Biney. *Diagnostic-strategy selection for a series systems*, IEEE Transaction on Reliability, vol.39, 1990, pp 273-280

[3] A. Pagès et M. Gondran. *Fiabilité des systèmes*. Eyrolles, 1980.

[4] J. Gao. *Surveillance et diagnostic de pannes*. PhD thesis, Université Laval, Québec

[5] M.C. Portmann, D. Ait-Kadi. *Minimisation des coûts de détection de pannes à partir des coupes minimales du diagramme de fiabilité*. MOSIM'01, avril 2001, Troyes

[6] M. Gondran, M. Minoux. *Graphes et algorithmes*. 3 éd. Paris, Eyrolles, 1995.

[7] C. H. Papadimitriou, K. Steiglitz. *Combinatorial optimization : Algorithms and complexity*, 1982.

[8] K. Marriott and P. J. Stuckey. *Programming with constraints*. MIT Press, 1998.