Chapitre 4 Stratégies de Recherche pour les SP

0. Introduction

Nous allons donner quelques stratégies de contrôle pour les systèmes de production (SP) dont le rôle est de **sélectionner la règle à déclencher** dans l'étape de sélection de la règle dans la procédure vue dans le chapitre des SP. Pour les SP décomposables, elle aura une tache supplémentaire qui est la sélection de la BDG composante ainsi que la règle composante à déclencher. D'autres taches supplémentaires peuvent lui être affectées:

- Vérifier les conditions d'applicabilité des règles,
- Test de terminaison,
- Mémoriser les règles déjà appliquées, etc…

Une caractéristique de cette sélection est la quantité d'informations (de connaissances) sur le domaine pour faire la meilleure sélection (meilleure règle - la plus prometteuse). La sélection de la règle à appliquer peut être:

- **arbitraire**: Recherche aveugle (non informée). C'est une recherche sans tenir compte d'informations disponibles sur le problème.
- **Stratégie guidée** : on utilise des connaissances disponibles sur le problème pour guider le choix de la bonne règle (recherche informée).

On peut donc affirmer que l'efficacité d'un SP peut dépendre du:

- Coût d'application des règles (nombre de règles pour atteindre le but),
- Coût du contrôle (effort fourni pour avoir une stratégie informée).

On peut observer donc:

- **Avec une stratégie aveugle**, le coût de **controle peut être modeste** (car on ne fournit aucun effort pour faire le choix de la règle, le choix est au hasard), mais elle **entraînera un coût d'application de règles élevé** (on essaye beaucoup de règles avant d'atteindre le but).

- **Informer complètement** un système sur le domaine nécessite une **stratégie de contrôle coûteuse** pour avoir cette information, en revanche on **minimise le coût d'application des règles** car on guide le SP lors de chaque choix (la stratégie guide le système vers le but).

On peut dresser les courbes suivantes des 2 coûts précédents et de déduire le coût global du système

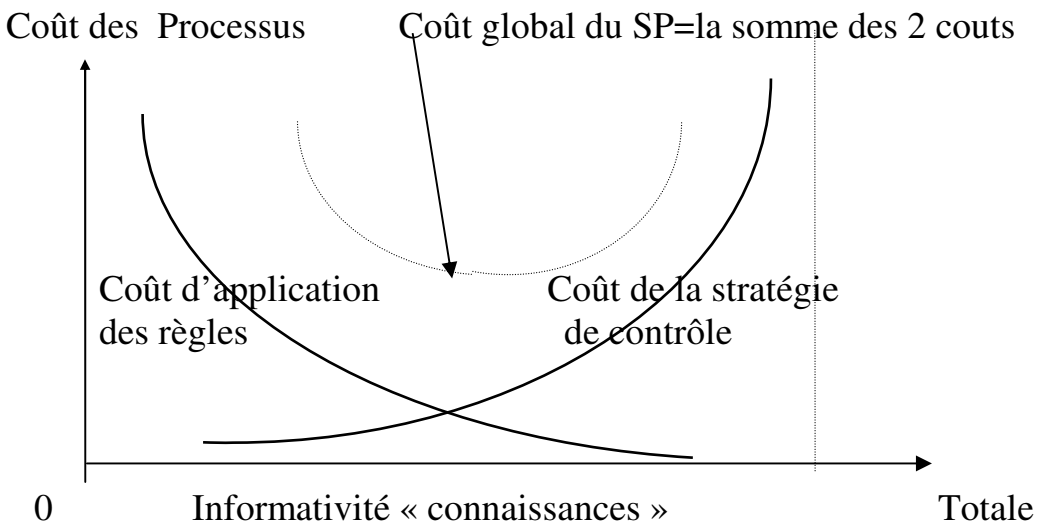

Fig: 1 Coût de calcul des SP

Il serait intéressant de concevoir des SP où on équilibre les 2 coûts. L'efficacité optimale pourrait être obtenue en utilisant des stratégies qui ne soient pas complètement informées mais aussi non nullement informée.

On a vu qu'il existe plusieurs stratégies pour les SP:

- Irrévocable (pour les systèmes commutatifs)
- Par essai successifs:
  - * Retour Arrière (RA) chronologique (simple à implémenter et nécessitant peu d'espace mémoire)
  - * Recherche avec graphe.

On s'intéressera à la 2eme catégorie des stratégies (par essai successifs) (1ere catégorie de stratégie n'est utilisée que pour les SP commutatifs).

1. Stratégie de Retour Arrière:

On donne une procédure récursive **Retour_arrière** qui a en entrée initialement un argument « Donnée » contenant la BDG initiale (état initial) et retourne soit la séquence de règles à appliquer pour atteindre le but soit ECHEC si cette séquence n'existe pas (but ne peut être atteint).

Procédure Récursive Retour_arierre(Donnée)
Début
1) ***Si*** *Terminé(**Donnée**) /*Donnée satisfait la condition d'arrêt - but atteint*/*
Alors *renvoyer(**Nil**)*
2) ***Si*** *Impasse(**Donnée**) /* Donnée n'est pas sur le bon chemin du but */*
Alors *renvoyer(**échec**) /* renvoyer echec à cet appel récursif */*
3) ***Règles*** *← app_regles(**Donnée**) /* une fonction qui calcule les règles applicables */*
*/*à Donnée et les ordonne arbitrairement ou suivant une heuristique */*
4) Boucle: ***si*** *vide(**Règles**)*
Alors *renvoyer(**échec**) /* plus de règles à appliquer, renvoyer echec */*
5) ***R*** *← première(**Règles**) /* Première règle est sélectionnée */*
6) ***Règles*** *← queue(**Règles**) /* On enlève la règle sélectionnée de la liste des règles */*
7) ***RDonnée*** *← R(**Donnée**) /* produire une nouvelle BDG en appliquant R à Donnée*/*
*8)Chemin← retour_arriere(**RDonnée**) /*Appel récursif de la procédure avec rdonnée*/*
9) ***Si*** *Chemin =* ***Echec*** */*si chemin retournée par l'appel récursif est ECHEC */*
alors *Aller à Boucle /* aller essayer une autre règle si il en reste*/*
10) Renvoyer (R.Chemin) / Concaténation de la règle R choisie au Chemin trouvé par*
Fin l'appel récursif/*

Remarques: On peut faire les remarques suivantes sur cette procédure:

- Elle ne se termine avec succès (non échec) que lorsqu'elle produit un état satisfaisant le But (ligne 1).
- La liste des règles à appliquer est construite en ligne 10
- Lorsqu'un échec se produit dans un appel récursif, on fait un Retour arrière au dernier point de choix (aller à boucle – ligne 9).
- En ligne 3, les règles applicables sont ordonnées. A ce niveau toute connaissance peut être utilisée. Par définition, si les règles correctes (on entend par règles correctes : les meilleures règles, règles prometteuses) sont choisies, il n'y aura aucun retour arrière. Par contre si le choix de la règle est arbitraire, la procédure devient inefficace.

Problème: En plus de ces remarques, on peut observer le problème suivant: La procédure peut ne jamais terminer, elle peut:

- Produire de nouvelles BDG indéfiniment,
- Boucler sur une même BDG.

Une solution envisageable à ce problème peut être la suivante:

- Imposer une limite de profondeur des appels récursifs
- Mémoriser les BD déjà produites pour éviter de boucler sur la même Base.

Il faut donc :

- ajouter une variable globale « limite » qui limite la profondeur (le nombre d'appels récursifs).
- que tout le chemin de l'état initial à l'état actuel soit un argument de la procédure.

RA Intelligent : Dans la procédure précédente, lorsqu'on effectue un retour arrière, on le fait au niveau juste inférieur; il existe d'autres systèmes intelligents pour faire des retours arrière directement au bon niveau car en général le meilleur niveau se trouve à un niveau inférieur. C'est ce qu'on appelle un R.A. intelligent.

Dans notre procédure précédente, on ne se souvient pas de toutes les BDG visitées mais uniquement celles se trouvant sur le chemin de l'état initial à l'état courant. Le retour arrière implique que toutes les BDG aboutissant à des échecs sont oubliées. Pour remédier à cet oubli, d'autres stratégies appelées stratégies de recherche avec graphe existent.

2. Stratégie de recherche avec graphe

Dans cette stratégie, on suppose que les **états produits sont représentées comme les noeuds** d'un graphe ou d'un arbre et **les arcs correspondent aux règles**. L'inconvénient est qu'on obtient un graphe volumineux. Une solution est de ne conserver que la BDG initiale et les traces des changements incrémentaux à partir desquels toute autre BDG peut être obtenue. Une stratégie de recherche avec graphe peut être considérée comme un moyen de trouver un chemin qui relie la BDG initiale à un noeud représentant le but.

2.1 rappel sur les graphes: Un graphe peut être décrit :

- explicitement en donnant sous forme d'un tableau, les noeuds du graphe (les BDG/états) avec les arcs (règles) ainsi que les couts associés (méthode inutilisable pour les graphes de taille importante).
- implicitement en donnant le noeud de départ (la BDG initiale) et une fonction de succession qui est appliquée à tout noeud pour donner ses successeurs avec les coûts associés. On introduit un opérateur de succession qui appliqué à un noeud donne tous ses successeurs avec leur couts associés (c'est le développement du nœud).

Le problème consiste à trouver un chemin de l'état initial à l'état But avec des fois une contrainte que le chemin soit de coût minimal.

Définition: Une solution (séquence de règles) est ***optimale*** s'il n'existe pas une autre solution (séquence de règles) de coût strictement inférieur.

2.2 Une procédure générale de recherche avec graphe

Une procédure qui consiste à développer une partie d'un graphe défini implicitement, peut être définie comme suit:
Dans la procédure suivante, la variable *Fermé* contient les nœuds qui ont été déjà développés et la variable *Ouvert* contient les noeuds feuilles qui ne sont pas encore développés.

Procédure recherche_avec_graphe;
1)Créer un graphe de recherche G qui consiste uniquement en nœud de départ d,
Mettre d sur une liste appelée OUVERT. / d est a développer */*
2) Créer une liste appelée FERME qui est initialement vide.
*3) BOUCLE: **SI** OUVERT est vide **alors** ECHEC exit /*plus de noeud à developper*/*
4) Sélectionner le 1er noeud de OUVERT, l'enlever de OUVERT et le
mettre dans FERME, appeler ce noeud n. / ce noud n est à développer */*
*5) **Si** n est un noeud But*
***alors** Terminer avec succès. Renvoyer le chemin obtenu le long des*
pointeurs (construits en phase 7) de n jusqu'à d dans G. exit
6) Développer le noeud n, produisant l'ensemble M de ses successeurs et
les mémoriser comme successeurs de n.
7) Mémoriser un pointeur vers n à partir des éléments de M.
Ajouter ces éléments à OUVERT / ces noeuds peuvent etre développés */*
8)Réordonner OUVERT, soit arbitrairement soit selon une heuristique
9) Aller à Boucle.
Fin

Cette procédure est assez générale pour produire une famille d'algorithmes de recherche avec graphe, **la différence dans cette famille réside dans la manière de réordonner** les nœuds-ligne 8. L'arbre de recherche est défini par ses pointeurs qui sont établis dans l'étape 7. Chaque noeud excepté d a un pointeur orienté vers un seul de ses parents dans G, qui définit son parent unique. Les noeuds de OUVERT sont les noeuds feuilles de l'arbre de recherche (qui n'ont pas encore été sélectionnés pour être développés) et les noeuds de FERME sont les autres nœuds (soit des feuilles qui n'ont pas produits de successeurs soit les nœuds qui ne sont pas des feuilles).

Remarques: Dans cette procédure les noeuds sont ordonnés en phase 8 pour sélectionner les meilleures pour être développées en phase 4. Cet ordre peut être arbitraire ou suivant des heuristiques.

- Chaque fois que le noeud sélectionné peut être développé en un noeud but, le processus se termine avec succès, auquel cas le chemin inverse est parcouru de l'état courant jusqu'à d suivant les pointeurs de la phase 4.
- Au contraire un échec peut arriver lorsque le (les) noeud(s) but(s) sont inaccessibles à partir du noeud de départ (liste OUVERT devient vide sans atteindre le but).

3. Procédures aveugles de recherche avec graphe

Si aucune information sur le domaine n'est disponible dans le classement des noeuds dans « ouvert », un plan arbitraire doit être utilisé à la phase 8 de l'algorithme: une telle procédure est dite aveugle (non informée).
Pour calculer la complexité des stratégies soient les paramètres suivants :

b : facteur de branchement : le nombre max de successeurs à un noeud.
d : la profondeur du noeud but le moins profond.
m : la longueur maximale d'un chemin dans l'espace d'états

3.1 Recherche en profondeur d'abord: Dans ce type de recherche aveugle, les noeuds de « Ouvert » sont ordonnés en ordre **décroissant suivant leur profondeur** dans l'arbre de recherche. **Les plus profonds sont placés en 1er**. Ceux de même profondeur sont classés arbitrairement. On impose une limite de profondeur pour empêcher que le système ne s'égare sur des chemins infinis. Ce type de recherche produit de nouvelles bases dans un ordre similaire à celui qui est produit par un mécanisme de contrôle utilisant le retour arrière (backtracking). En général on préfère le retour arrière de la recherche en profondeur d'abord car il est plus facile à implémenter et nécessite moins d'espace mémoire, car il ne mémorise pas tous les noeuds produits mais uniquement le chemin menant au but.
Complétude: non si profondeur infinie, (des cycles), oui si espaces finis acycliques
Complexité en temps : $O(b^m)$
Complexité en espace : O(bm)
Optimal : Non.

3.2 Recherche en Largeur d'abord: Ce type de procédure aveugle ordonne les noeuds de Ouvert dans l'ordre **croissant de leur profondeur**

dans l'arbre. Il est montré que cette recherche garantit la découverte du chemin le plus court si celui-ci existe. Dans le cas où il n'existe pas elle échouera pour les graphes finis et boucle pour les graphes infinis.
Complétude : Oui
Complexité de temps $1 + b + b^2 + b^3 + \ldots + b^d + (b^{d+1} - b) = O(b^{d+1})$
Complexité d'espace $O(b^{d+1})$ (on garde tous les noeuds en mémoire)
Optimal : - non en général
- oui si le coût des actions est le même pour toutes les actions

3.3 Profondeur limitée

C'est le même algorithme que celui de **profondeur d'abord, mais avec une limite** de l sur la profondeur. Les noeuds de profondeur l n'ont pas de successeurs.

Complétude : oui, si $l \geq d$
Complexité en temps : $O(b^l)$
Complexité en espace : $O(bl)$
Optimal : Non
3 possibilités : solution, échec ou absence de solution dans les limites de la recherche

3.4 Profondeur itérative

Le problème avec la recherche en profondeur limitée est de fixer la bonne valeur de l. La profondeur itérative est de **répéter la méthode de profondeur limitée pour toutes les valeurs possibles de l = 1, 2,** … Elle a les mêmes propriétés que la précédente.

Complétude : oui
Complexité en temps : $O(b^l)$
Complexité en espace : $O(bl)$
Optimal : Oui

3.5 Recherche bidirectionnelle

On exécute deux explorations simultanées: en aval depuis l'état initial et en amont à partir du but. On arrete lorsque les deux explorations se rencontrent au milieu.

Complétude : oui
Complexité en temps : $O(b^{d/2})$
Complexité en espace : $O(b^{d/2})$
Optimal : Oui à condition que les couts sont identiques et on utilise l'exploration en largeur d'abord pour les 2 sens.

4. Procédure Heuristique de recherche avec graphe