**Chapitre 2 : Résolution de problèmes**

Le but de ce chapitre est d’introduire quelques algorithmes de recherche pour la résolution des problèmes. Nous présentons d’abord la notion de problèmes et nous donnons quelques exemples de problèmes connus. Suite à quoi nous détaillons quelques algorithmes de résolution.

**2.1 Définition formelle d’un problème :**

Un problème est défini par la donnée de cinq éléments qui sont

1. Un état initial
2. Un ensemble d’actions
3. Une fonction successeur qui définit le résultat de l’exécution d’une action dans un état
4. Un ensemble d’états

On peut voir un problème comme **un graphe orienté** où les nœuds sont les états accessibles depuis l’état initial et les arcs sont des actions. **Une solution est un chemin** de l’état initial vers un état but. Une solution est dite **optimale** si son cout est **le minimum des couts des autres solutions**.

**2.2 Exemples de problèmes**

1. **Problèmes de satisfaction de contraintes** : un problème de satisfaction de contraintes est défini par la donnée d’un ensemble de variables, des ensembles de valeurs permises pour chaque variable appelés domaines des variables, et un ensemble de contraintes sur les combinaisons de valeurs des variables. L’objectif est de trouver des valeurs pour toutes les variables de telle sorte que toutes les contraintes soient simultanément satisfaites.

**Voici une formalisation de ce problème**

***Etats****:* solutions partielles (affectation de quelques variables)

***Etat initial***: valuation vide (aucune variable n’est affectée)

***Actions*** : choisir une valeur de son domaine pour une des variables restantes non encore instanciée.

***Fonction successeur***; ajouter la nouvelle valeur à la valuation actuelle

***Test de but*** : un état but est un état où toutes les variables sont instanciées et toutes les contraintes sont satisfaites

***Coût de la solution*** : donner le même coût pour chaque action

1. **Problème des 8 reines :**  Le but de ce problème est de placer les 8 reines sur un échiquier de telle sorte qu’aucune reine ne peut attaquer une autre reine. C’est-à-dire il n’y a pas deux reines sur une même colonne, ni sur la même ligne ni sur la même diagonale.

**Voici une formalisation de ce problème**

***Etats*** : toute configuration de 0 à 8 reines sur la grille.

***Etat initial*** : la grille vide.

***Actions*** : ajouter une reine sur n’importe quelle case vide de la grille.

***Fonction successeur*** : la configuration qui résulte de l’ajout d’une reine à une case spécifiée à la configuration courante.

***Test de but*** : une configuration de huit reines avec aucune reine sous attaque.

***Coût des actions*** : ce pourrait être 0, ou un coût constant pour chaque action - nous nous intéressons pas du chemin, seulement l’état but obtenu.

Contraintes du problème des 8 reines

* les reines doivent être sur des lignes différentes  
  *Clig = {Xi ≠ Xj / i et j sont 2 entiers différents compris entre 1 et n}*
* les reines doivent être sur des diagonales montantes et descendantes différentes  
  Cij : | Xi - Xj |≠|i-j|

**2.3 Structure générale d’un algorithme de résolution**

La plus part des algorithmes de recherche suivent à peu près le même schéma : on commence toujours de l’état initial et puis on exécute les actions suivantes itérativement jusqu’à atteindre le but.

* **S’il** n’y a plus d’états à traiter alors renvoyer échec
* **Sinon** choisir un des états à traiter
* **Si** l’état est un état but alors renvoyer la solution correspondante
* **Sinon** retirer cet état de l’ensemble des états à traiter et le remplacer par l’ensemble de ses états successeurs

**2.3 .1 Evaluation des algorithmes de recherche**

Les métriques de comparaison des algorithmes sont :

* *Complexité en temps* : combien de temps prend l’algorithme pour trouver une solution ?
* *Complexité en espace* : de combien d’espace mémoire a besoin l’algorithme pour trouver une solution ?
* *Complétude* : est ce que l’algorithme trouve toujours la solution quand il existe une solution ?
* *Optimalité* : Est-ce que l’algorithme retourne des solutions optimales.

**3. Un algorithme de recherche générique**

Dans ce paragraphe, nous présentons un algorithme de recherche générique, dont le pseudo-code est comme suit :

fonction **recherche**(état initial, successeurs, test but, coût)

nœuds\_ à\_ traiter ← créer\_liste(créer\_noeud(état initial, [ ], 0))

**boucle**

**si** vide ?(nœuds\_à\_traiter) **alors** renvoyer échec

nœud ← enlever premier\_nœud (nœuds\_à\_traiter )

**si** test\_but(état(nœud)) = vrai

**alors** renvoyer chemin(nœud), état(nœud))

**pour tout** (action, état) dans successeurs(état(nœud))

chemin← [action, chemin(nœud)]

coût du chemin ← coût du chemin(nœud) + coût(état)

s← créer\_nœud(état, chemin, coût\_ du\_ chemin)

insérer(s, nœuds \_à traiter)

Cet algorithme générique prend en entrée la description du problème à résoudre:

* un état initial,
* une fonction de successeurs,
* un test de but,
* une fonction de coût.

**La première étape** de cet algorithme est d’initialiser une liste de nœuds à traiter, un nœud étant composé d’un état, d’un chemin, et du coût du chemin. Nous commençons avec un seul nœud correspondant à l’état initial.

**A chaque itération de la boucle**, nous vérifions si la liste de nœuds à traiter est vide. Si c’est le cas, nous avons examiné tous les chemins possibles sans pour autant trouver une solution, donc l’algorithme renvoie échec. Si la liste contient encore des nœuds, nous sortons le premier nœud de la liste. Si l’état de ce nœud est un état but, c’est gagné, et nous renvoyons l’état et le chemin qui permet d’accéder à ce nœud but. Dans le cas contraire, la recherche se poursuit : nous produisons les successeurs du nœud et les insérons dans la liste de nœuds à traiter.

Dans ce qui suit nous allons présenter quelques algorithmes de recherche.

**4. Recherche non-informée**

Les algorithmes de recherche non informée ne disposent pas d’informations pour distinguer les états prometteurs ; ces algorithmes font une recherche exhaustive de tous les chemins possibles depuis l’état initial. (ce n’est pas le cas par exemple des programmes joueurs d’échecs qui ne peuvent explorer toutes les possibilités, et se concentrent donc à chaque étape sur un petit nombre de coups qui leur semblent être les meilleurs).

**Parcours en largeur**

Le parcours en largeur est un algorithme de recherche très simple : nous traitons d’abord l’état initial, puis tous ses successeurs, puis les successeurs des successeurs, etc. Tous les nœuds d’une certaine profondeur sont examinés avant les nœuds de profondeur supérieure. Pour implémenter cet algorithme, il suffit de placer les nouveaux nœuds systématiquement à la fin de la liste des nœuds à traiter.

*B C*

*D E F G*

Le parcours en largeur est un algorithme de recherche complet à condition que le nombre de successeurs des états soit toujours fini (ce qui est très souvent le cas dans les problèmes courants). Pour voir pourquoi, soit p le nombre minimal d’actions nécessaires pour atteindre un état but depuis l’état initial. Comme nous examinons les nœuds profondeur par profondeur, et qu’il n’y a qu’un nombre fini de nœuds à chaque profondeur, nous atteindrons forcément le niveau p.

Le parcours en largeur trouve toujours un état but de plus petite profondeur possible. Donc si nous cherchons une solution quelconque, ou une solution avec le moins d’actions possibles, le parcours en largeur est optimal.

La complexité de cet algorithme est comme suit. Supposons que chaque état possède s successeurs et que p soit le nombre minimal d’actions pour relier l’état initial à un état but. Dans le pire des cas, nous allons examiner tous les nœuds de profondeur au plus p afin de trouver l’état but qui se trouve à cette profondeur. Nous allons alors produire 1 + s + s2 + s3 + ... + sp + (sp+1 − s) nœuds. sp+1 − s correspond au nombre de nœuds de profondeur p+1 qui ont été produits lors de l’examen des nœuds.

**Parcours en profondeur**

Le parcours en profondeur suit le chemin courant le plus longtemps possible. Il est facile à implémenter : il faut tout simplement mettre des successeurs du nœud courant au début de la liste de nœuds à traiter.

Le parcours en profondeur n’est pas complet parce que l’algorithme peut **continuer sur un chemin infini**, ignorant complètement un état but qui se trouve sur un autre chemin. Si par contre, nous n’avons qu’un nombre fini de chemins possibles (ce qui n’est pas souvent le cas), le parcours en profondeur sera complet. L’algorithme n’est pas optimal: il n’y a rien qui garantie que le premier état but trouvé sera le bon.

Pour la complexité. Si nous avons s successeurs et une profondeur maximale de m actions, dans le pire des cas, nous aurions besoin d’examiner tous les sm nœuds pour trouver une solution.

Le principal avantage du parcours en profondeur reste sa faible complexité en espace. Pour un problème ayant s successeurs et une profondeur maximale de m actions, il nous faut garder au plus 1+(m\*(s−1)) nœuds en mémoire (correspondant au chemin actuellement en développement plus les successeurs de profondeur 1 que nous n’avons pas encore traités, les successeurs de profondeur 2 que nous n’avons pas encore traités, etc.). La complexité en espace est donc de O(s\*m). Même quand s et m sont grands, la mémoire nécessitée par le parcours en profondeur reste raisonnable, ce qui nous permet de traiter des problèmes qui ne sont pas abordables par le parcours en largeur.

**Parcours en profondeur limitée**

Nous venons de voir que le parcours en profondeur n’est pas très bien adapté aux problèmes où la longueur des chemins n’est pas bornée parce que nous risquons de suivre un chemin inﬁni qui ne mène pas à un état but. Une façon naïve d’éviter ce problème **serait de ﬁxer une profondeur maximale** et de ne pas considérer les chemins de profondeur supérieur à cette limite. L’algorithme résultant, nommé **parcours en profondeur limitée**, va toujours terminer (à condition bien sûr que le nombre de successeurs soit ﬁni) ce qui n’était pas le cas pour le parcours en profondeur simple. Le parcours en profondeur limitée est **complet** si la profondeur maximale est supérieure à la profondeur minimale des solutions. Mais comme nous ne savons pas en général quelle profondeur sera suffisante, nous ne saurons pas si l’algorithme est complet ou non pour une profondeur donnée. Comme c’était le cas pour le parcours en profondeur classique, le parcours en profondeur limitée n’est pas optimal en général. Pour une profondeur maximale ﬁxée à m et s successeurs par état, la complexité en temps sera sm (dans les pire des cas il faut examiner chacun des sm chemins de profondeur au plus m). La complexité en espace est d’ordre O(s\*m).

**Parcours en profondeur itérée**

Dans le dernier paragraphe, nous avons introduit le parcours en profondeur limitée dont l’inconvénient principal est la difficulté de bien choisir la borne de la profondeur. Le parcours en profondeur itérée permet de remédier à cet inconvénient : comme nous ne savons pas quelle borne de profondeur choisir, nous allons les essayer les unes après les autres. Nous effectuons donc un parcours en profondeur limitée avec une borne de 1, puis un parcours de profondeur avec une borne de 2, et nous continuons à augmenter la borne jusqu’à ce que l’on trouve une solution.

Comme le parcours en largeur, le parcours en profondeur itérée est complet — à condition que le nombre de successeurs soit toujours ﬁni — et optimal quand le coût d’un chemin ne dépend que du nombre d’actions. Pour un problème ayant s successeurs par état et une solution de profondeur p, nous allons examiner p+1 fois le nœud unique de profondeur 0, p fois les nœuds de profondeur 1, p-1 fois les nœuds de profondeur 2, ..., et une seule fois les nœuds de profondeur p. Le nombre de nœuds généré lors de la recherche est donc (p + 1) + (p)s +(p-1)s2 +(p-2)s3 + ... + (1)sp ce qui donne une complexité en temps en O(sp). Remarquons que même s’il semble inefficace de traiter les mêmes nœuds plusieurs fois, la complexité en temps du parcours en profondeur itérée est en fait moins importante que celle du parcours en largeur (où la complexité est O(sp+1) parce que nous produisons des nœuds de profondeur p+1 lors de l’examen des nœuds de profondeur p) et que celle du parcours en profondeur classique (où nous pouvons examiner les nœuds de profondeur supérieure à p). Quant à la complexité en espace, nous gardons au plus 1 + (p\*(s-1)) nœuds en mémoire lors de l’examen des nœuds de profondeur p, donc la complexité en espace est en O(s\*p). Le parcours en profondeur itérée combine donc les avantages du parcours en largeur (complétude et optimalité) et du parcours en profondeur (faible complexité en espace). C’est pour cette raison que le parcours en profondeur itérée est considéré aujourd’hui comme le meilleur algorithme de recherche non-informée pour les problèmes de grande taille où la profondeur des solutions est inconnue.

**Etats redondants**

Les algorithmes que nous venons de présenter peuvent visiter plusieurs fois le même état pendant la même recherche. Il y a deux types de redondances possibles : d’une part, il y a des états qui peuvent être visités deux fois sur le même chemin (par exemple, au taquin, si vous jouez droite puis gauche, vous revenez à l’état initial), et d’autre part, il peut y avoir plusieurs chemins différent qui amènent au même état. Dans ce paragraphe, nous considérons les façons d’éviter de telles redondances.

Il est relativement simple d’éviter le premier type de redondance. Il suffit d’examiner au fur et à mesure les chemins des nœuds à insérer et de ne jamais ajouter à la liste de nœuds à traiter des nœuds dont les chemins contiennent plusieurs fois le même état. Il est possible de modifier les différents algorithmes de telle façon que cette modification ne change pas significativement la complexité en espace. Par contre, si nous voulons éviter le deuxième type de redondance, cela signifie qu’il faut se rappeler de tous les états déjà visités et de ne garder qu’un seul chemin par état. Plus spécifiquement, il faut gérer une liste des états déjà visités et y ajouter les états quand ils sont visités pour la première fois. Avant de générer les successeurs d’un nœud, il faut vérifier si l’état successeur est dans la liste des états déjà visités (et si c’est le cas, nous ne générons pas de successeurs). Cette modification ne changera pas trop la complexité en espace pour le parcours en largeur (qui souffre déjà d’une haute complexité en espace). Par contre, pour le parcours en profondeur et le parcours en profondeur itérée qui gardent normalement peu de nœuds en mémoire cette modification va augmenter très significativement la complexité en espace.

**5. Recherche Heuristique**

Les algorithmes que nous avons vus font une recherche exhaustive de tous les chemins possibles, ce qui les rend inefficaces voire inutilisables sur les problèmes de grande taille. Dans cette section, nous présentons les algorithmes de recherche heuristiques qui utilisent des informations supplémentaires pour pouvoir mieux guider la recherche. Tout algorithme de recherche heuristique dispose d’une fonction d’évaluation f qui détermine l’ordre dans lequel les nœuds sont traités : la liste de nœuds à traiter est organisée en fonction des valeurs des nœuds, avec les nœuds de plus petite valeur en tête de liste. A priori, il n’y a pas vraiment de restriction sur la nature de la fonction d’évaluation, mais souvent elle a comme composante une fonction heuristique h où h(n) = coût estimé du chemin de moindre coût reliant n à un état but.

Notons que la fonction heuristique prend un nœud en entrée mais sa valeur ne dépend que de l’état associé au nœud. Et bien sûr, h(n) = 0 si n est un état but.

**Best-ﬁrst search (meilleur d’abord)**

Le principe de l’algorithme “best-ﬁrst search” est d’examiner les nœuds qui semblent les plus proches d’un état but, dans l’espoir d’aboutir plus vite à une solution. Un peu plus formellement, la stratégie employée par best-ﬁrst search consiste à utiliser la fonction heuristique h comme fonction d’évaluation (c’est-à-dire qu’on prend f(n)=h(n)).

Pour une profondeur maximale de p et pour s successeurs, la complexité en temps et en espace de best-ﬁrst search sont toutes les deux en O(sp) dans les pire des cas, même si la complexité dépend en pratique de la qualité de la fonction heuristique.

**Recherche A\***

Best-ﬁrst search donne la préférence aux nœuds dont les états semblent les plus proches d’un état but, mais il ne prend pas en compte les coûts des chemins reliant l’état initial à ces nœuds. Néanmoins, c’est une information très pertinente car le coût d’un chemin passant par un nœud n est la somme du coût de chemin entre l’état initial et n et le coût du chemin reliant n à un état but. C’est cette idée qui est à la base de la recherche A\*. Si nous appelons g(n) le coût du chemin entre l’état initial et n, la fonction d’évaluation utilisée par la recherche A\* est donnée par la formule suivante :

(n)=g(n)+h(n)

Comme g(n) est le coût réel associé au chemin entre l’état initial et n et que h(n) est une estimation du coût du chemin entre n et un état but, la fonction d’évaluation f donne une estimation du coût de la meilleure solution passant par le nœud n.

L’algorithme de recherche A\* est complet et optimal s’il y a un nombre ﬁni de successeurs (on commence à avoir l’habitude....) et si nous plaçons une certaine restriction sur la fonction heuristique h. Il faut que la fonction h soit admissible, c’est à dire que la valeur h(n) ne doit jamais être supérieure au coût réel du meilleur chemin entre n et un état but.

La complexité de la recherche A\* dépend de la fonction heuristique en question. En général, la complexité en temps et en espace est grande, ce qui rend la recherche A\* mal adaptée pour les problèmes de grande taille. Pour pallier cet inconvénient, plusieurs autres algorithmes heuristiques moins gourmands en mémoire ont été proposés.

.

**Recherche Locale**

Nous avons vu que pour certains problèmes, en particulier les problèmes de satisfaction de contraintes, nous ne nous intéressons pas au chemin reliant l’état initial à l’état but mais seulement à l’état but lui-même. Pour les problèmes de ce type, il existe une autre stratégie possible : générer les états successivement sans s’intéresser aux chemins jusqu’à ce que nous trouvions un état but. Cette idée est à la base des algorithmes de recherche locale, qui sacrifient la complétude pour gagner du temps et de la mémoire.

L’algorithme local le plus simple s’appelle la recherche locale gloutonne. Nous commençons avec un état choisi aléatoirement. Si l’état est un état but, nous arrêtons la recherche. Sinon nous générons ses successeurs et leurs valeurs heuristiques. S’il n’existe pas de successeur avec une meilleure valeur que la valeur heuristique de l’état actuel, nous pouvons plus améliorer la situation, donc nous arrêtons la recherche. Sinon, nous choisissons l’état successeur ayant la meilleure valeur heuristique, et nous continuons ainsi.

L’avantage de la recherche locale gloutonne est qu’elle a une très faible consommation en mémoire (il ne faut garder que l’état actuel en mémoire) et aussi en temps. Par exemple, pour le problème des huit reines, l’algorithme termine après 4 étapes quand il trouve une solution et 3 étapes quand il est bloqué (en moyenne).

L’inconvénient principal de la recherche locale gloutonne est son faible taux de succès qui résulte du fait que nous arrêtons la recherche si jamais nous ne pouvons plus améliorer directement la valeur heuristique. Pour le problème des huit reines, cet algorithme ne réussit que 14% du temps.

Il y a plusieurs améliorations possibles de la recherche locale gloutonne. Une idée simple serait de permettre à l’algorithme de visiter les successeurs ayant la même valeur que l’état actuel (il faut mettre une borne sur le nombre de déplacements consécutifs de ce type pour ne pas tourner en boucle). Cette modification augmente significativement le taux de succès. Pour le problème des huit reines, si nous permettons au plus 100 déplacements consécutifs qui n’améliorent pas la valeur heuristique, le taux de succès passe de 14% à 94%. Une autre possibilité est d’ajouter de l’aléatoire : au lieu de choisir toujours le meilleur voisin, nous pouvons choisir un voisin aléatoirement (où la probabilité de sélectionner un état est défini en fonction de sa valeur heuristique). Finalement, nous pouvons tout simplement recommencer la recherche locale gloutonne à partir d’un autre état choisi au hasard, et continuer ainsi jusqu’à obtenir un état but. Cette technique pourrait sembler un peu simpliste, mais elle s’avère très efficace : elle permet de résoudre le problème des 3 millions de reine en moins d’une minute sur un ordinateur de bureau! Grace à leur très faible complexité en espace, les algorithmes locaux peuvent être utilisés pour résoudre des problèmes de taille importante qui ne peuvent pas être résolus avec des algorithmes classiques.