

Partie 1 – Résolution de problèmes

1.2 STRATEGIES DE RECHERCHE NON INFORMEE

1.2.1 LA LARGEUR

Recherche en largeur

breadth-first search

■ Principe :

- Les nœuds de profondeur d sont développés avant ceux de profondeur $d+1$
- On utilise une **insertion en queue** dans l'algorithme **Explorer**

La frontière est gérée en file



Recherche en largeur

ExplorerLargeur (p : Problème) : Nœud ou null

Nœud racine = new Nœud(p.étatInitial, null, null, 0) ;

Liste frontière = new Liste() ;

frontière.insérer(racine) ;

tant que non frontière.vide?() faire

 Nœud n = frontière.oterTête() ;

si p.but?(n.état) alors retourner n;

pour toute Action a dans p.actions(n.état) faire

 Nœud sn = new Nœud (a, n, a.résultat(n.état), n.cout+a.cout(n.etat)) ;

 frontière.**insérerQueue**(sn);

fin pour

fin tant que

retourner null;

Complétude et optimalité de la largeur

- Stratégie complète
 - tous les chemins sont étudiés de manière systématique
 - même si l'arbre de recherche est infini et qu'on ne vérifie pas que l'on réexplore plusieurs fois un même état, si une solution existe elle sera trouvée
- Trouve une solution la plus proche de la racine donc optimale seulement si
 1. le critère d'optimalité diminue avec le nombre d'actions effectuées
 2. toutes les opérations ont le même coût

Complexité de la largeur

- En fonction du nombre de nœuds développés
 - Soit **d** la **profondeur** de l'arbre à laquelle la solution est trouvée
 - Soit **b** le **facteur de branchement** (le nombre maximum de nœuds générés par un appel à **développer**)

On génère $1+b+b^2+b^3\dots+b^d+(b^d-1)b$ nœuds

Tous les nœuds générés doivent être mémorisés
- Les complexités temporelle et spatiale sont **bornées par $O(b^{d+1})$**
 - » Ces complexités supposent que les opérations d'expansion et de test d'état but ont des complexités constantes

Quelques chiffres !

- Supposons qu'une machine soit capable de tester et de développer 1000 nœuds par seconde et qu'un nœud nécessite 100 octets de stockage alors pour un facteur de branchement $b=10$

Profondeur	Nb noeuds	Temps	Espace
3	11101	11 s	1 Mo

Ces complexités exponentielles ne permettent que de résoudre des problèmes de « petite taille »

Bilan de la recherche en largeur

- Complète mais complexité spatiale (et temporelle) prohibitive
- Optimale uniquement sous certaines conditions
- Idées d'amélioration :
 - Ne pas maintenir tout l'arbre lors de l'exploration
=> recherche en profondeur
 - Utiliser le critère de coût pour choisir le prochain nœud à développer
=> recherche de coût min

Partie 1 – Résolution de problèmes

1.2 STRATEGIES DE RECHERCHE NON INFORMEE

1.2.2 LA PROFONDEUR

Recherche en profondeur

■ Principe :

- On développe toujours un des nœuds le plus profond
- On ne remonte dans l'arbre que lorsqu'on tombe sur un nœud non but et non développable (pas d'action à appliquer)
- On utilise une **insertion en tête** dans l'algorithme **explorer**

La frontière est gérée en pile



Recherche en profondeur

ExplorerProfondeur (p : Problème) : Nœud ou null

```
Nœud racine = new Nœud(p.étatInitial, null, null, 0) ;  
Liste frontière = new Liste() ;  
frontière.insérer(racine) ;  
tant que non frontière.vide?() faire  
    Nœud n = frontière.oterTête() ;  
    si p.but?(n.état) alors retourner n;  
    pour toute Action a dans p.actions(n.état) faire  
        Nœud sn = new Nœud (a, n, a.résultat(n.état), n.cout+a.cout(n.etat)) ;  
        frontière.insérerTête(sn);  
    fin pour  
fin tant que  
retourner null;
```

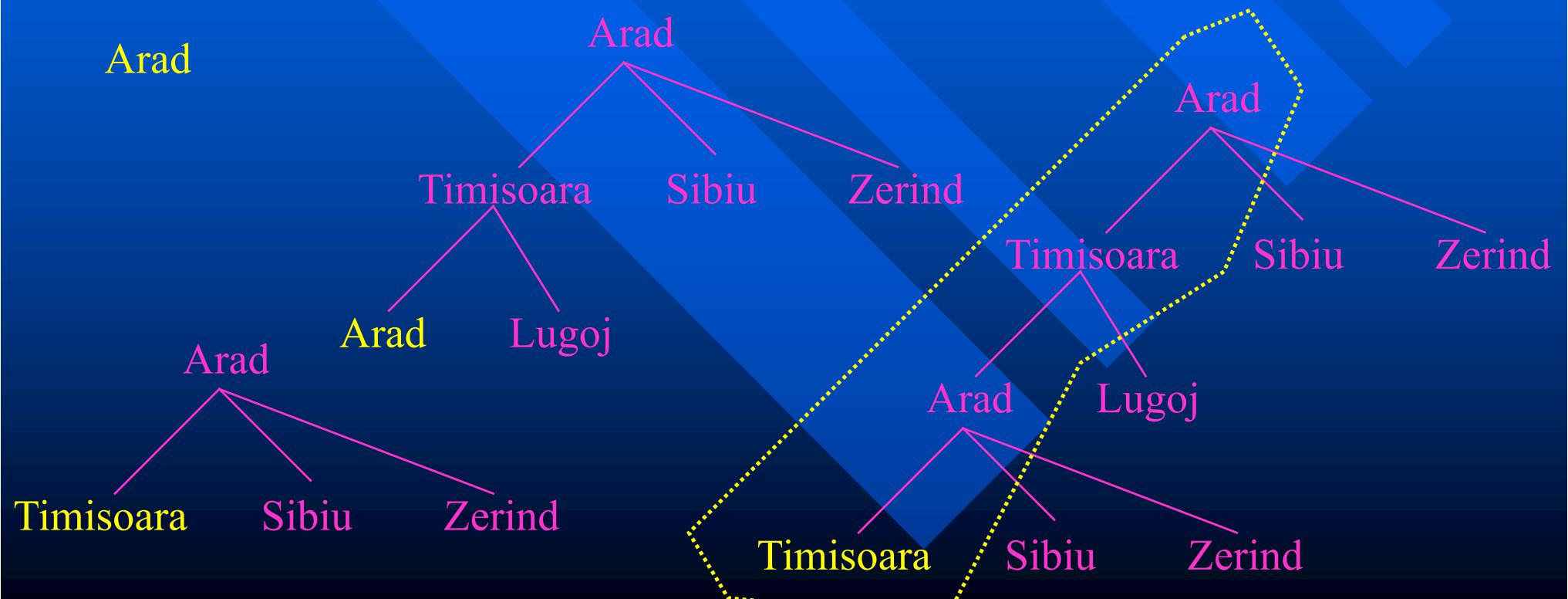
Complexité de la profondeur

- Peu coûteuse en mémoire car on ne mémorise qu'un chemin (plus les nœuds frontières) et non l'arbre entier
 - Autre avantage : implémentable par un algorithme récursif qui gère automatiquement la pile
- Soit **m** la profondeur maximale de l'espace de recherche et **b** le facteur de branchement
 - Complexité temporelle : $O(b^m)$, mais elle peut être rapide en pratique si le problème possède beaucoup de solutions
 - Complexité spatiale : $O(mb)$

*Par comparaison au parcours en largeur,
la profondeur 14 nécessite 14Ko au lieu des 11111To*

Complétude et optimalité de la profondeur

- **Problème** : On insère les nœuds développés en tête de liste ce qui peut conduire à **développer des branches infinies** ou **créer des cycles**



Complétude et optimalité de la profondeur

■ Stratégie non complète :

- à cause des branches infinies potentielles dues aux circuits dans l'espace des états, voire due au fait que le nombre d'états peut être infini

■ Stratégie non optimale :

- car elle retourne la première solution rencontrée sans aucune corrélation avec un critère de coût

■ Rétablir la complétude :

- Déetecter les états répétés si l'espace des états est fini
- Utiliser une borne pour ne pas descendre infiniment dans la même branche => cf TD

Détection des états répétés

■ Objectif :

- Elaguer des branches de l'arbre de recherche dont on sait qu'elles ne conduiront pas à une solution

■ Avantages :

- Restaurer la complétude de la profondeur quand l'espace d'états est fini
- Diminuer la taille de l'arbre de recherche en évitant de redévelopper des branches déjà explorées



Détection des états répétés

- Trois types d'« élagage » sont envisageables
 - Ne pas retourner à l'état d'où l'on vient : nécessite une comparaison au nœud précédent : $O(1)$
 - Ne pas créer de chemin avec des états répétés : nécessite une comparaison avec tous les nœuds prédecesseurs jusqu'à la racine : $O(d)$ => nécessaire pour la complétude de la profondeur si pas de borne et circuits possibles dans l'espace des états
 - Ne pas recréer un **second nœud** pour un même état : nécessite de stocker tous les états du graphe : $O(s)$ où s est la taille de l'espace des états, également borné par $O(b^d)$

Recherche en profondeur complète

ExplorerProfondeurComplet (p : Problème) : Nœud ou null

Nœud racine = new Nœud(p.étatInitial, null, null, 0) ;

Liste frontière = new Liste() ;

frontière.insérer(racine) ;

tant que non frontière.vide?() faire

Nœud n = frontière.oterTête() ;

si p.but?(n.état) alors retourner n;

pour toute Action a dans p.actions(n.état) faire

Etat se = a.résultat(n.état) ;

si se \notin étatsAncêtres(n) alors

Nœud sn = new Nœud (a, n, se, n.cout+a.cout(n.état)) ;

frontière.**insérerTête**(sn);

fin si

fin pour

fin tant que

retourner null;

étatsAncetres(n) :

si n=null, étatsAncêtres(n) = Ø

sinon étatsAncêtres (n) = {n.état} \cup étatsAncêtres (n.parent)

Optimisation de la recherche

- Ne pas **ré-exploré** un état déjà exploré
 - On gère un ensemble **exploré** d'états déjà explorés
 - Avant d'insérer un nœud dans la **frontière**, on vérifie que son état n'a pas déjà été exploré

Ensemble<Etat>
Ensemble() : Ensemble
contient? (e : Etat) : Bool
ajouter (e : Etat) : void

Constructeur retournant un ensemble vide

Retourne vrai si e est dans l'ensemble

Ajoute l'état e à l'ensemble

Profondeur Optimisée

ProfondeurCompletOptimisé (p : Problème) : Nœud ou null

Nœud racine = new Nœud(p.étatInitial, null, null, 0) ;

Liste<Noeud> frontière = new Liste() ;

frontière.insérer(racine) ;

Ensemble<Etat> exploré = new Ensemble();

tant que non frontière.vide?() faire

Nœud n = frontière.oterTête() ;

si non exploré.contient?(n.etat) alors

exploré.ajouter(n.etat);

si p.but?(n.état) alors retourner n;

pour toute Action a dans p.actions(n.état) faire

 Nœud sn = new Nœud (a, n, a.résultat(n.état), n.cout+a.cout(n.etat)) ;

 frontière.insérerTête(sn);

fin pour

fin si

fin tant que

retourner null;

Partie 1 – Résolution de problèmes

1.2 STRATEGIES DE RECHERCHE NON INFORMEE

1.2.2 LA RECHERCHE DE COUT MIN

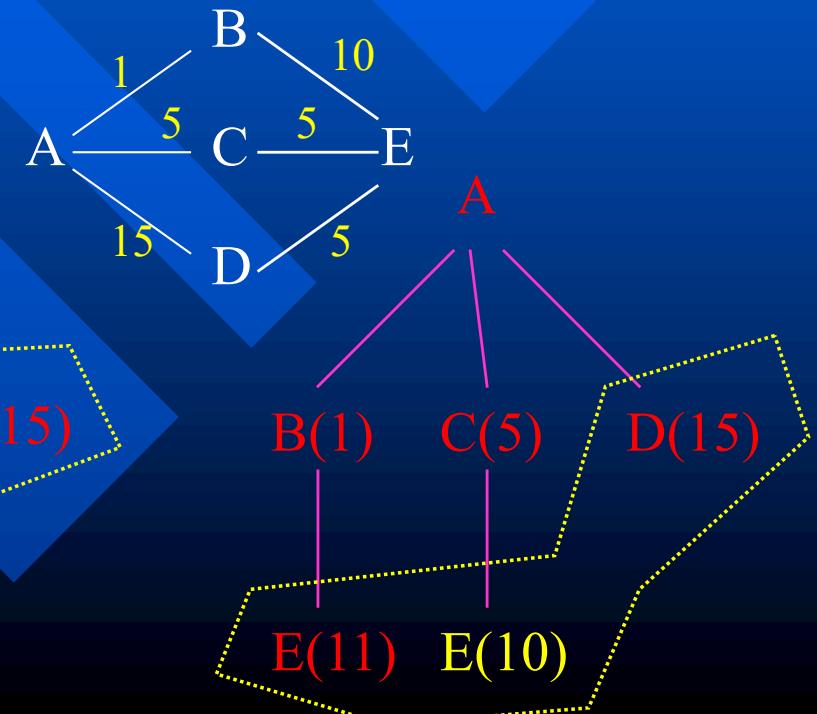
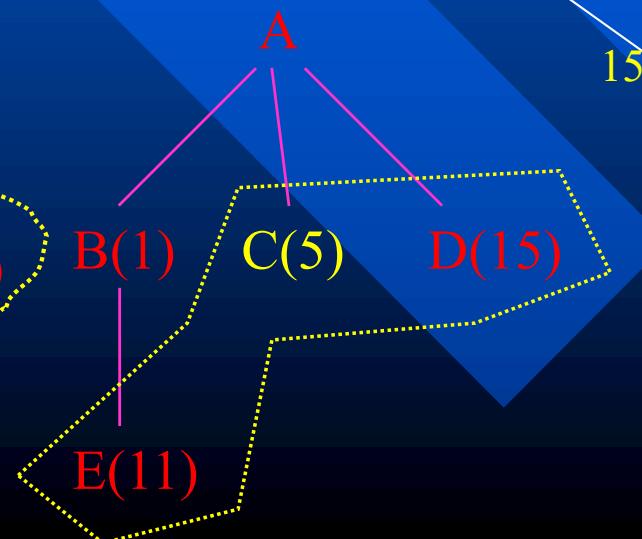
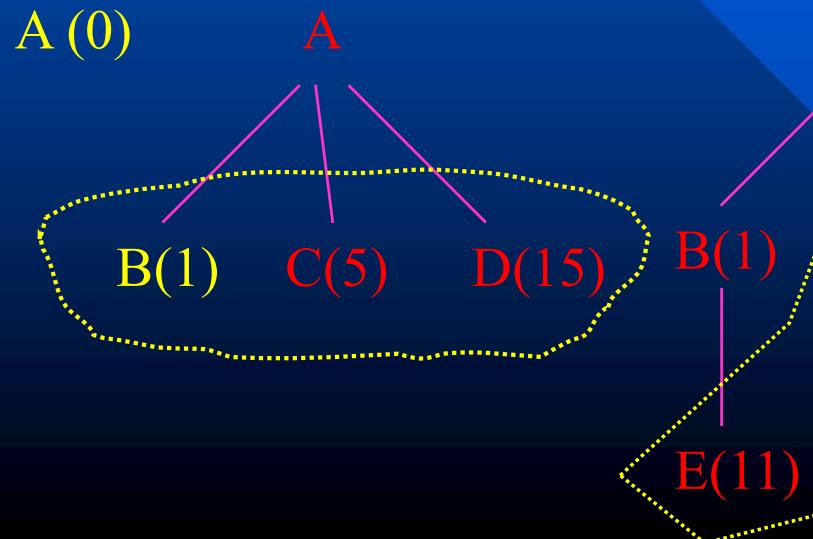
Recherche de coût min

Uniform Cost Search

■ Principe :

- Les nœuds de coût min sont explorés en premier
- On note $g(n)$ le coût du chemin de la racine au nœud n
- On utilise une **insertion par coût g croissant** dans l'algorithme **Explorer**
La frontière est gérée comme une liste triée croissante

- Ex. recherche de route de A à E :



Performances de la recherche coût min

■ Complétude :

Cas où l'espace d'états est fini

=> complète si on détecte les états répétés

Cas où l'espace d'états est infini

=> complète si on ne peut pas générer une infinité de nœuds ayant un coût inférieur au coût de la solution optimale g^*

» Une manière d'assurer cette condition est d'imposer à la fonction de coût g qu'elle augmente au moins d'un coût fixe ε à chaque action :

Il existe $\varepsilon > 0$ tel que pour tout nœud n : $g(fils(n)) \geq g(n) + \varepsilon$

» Si g est défini comme la somme du coût des actions, cela revient à :

Il existe $\varepsilon > 0$ tel que pour toute action a : $cout(a) \geq \varepsilon$

On rappelle qu'on suppose que le nombre d'actions possibles est fini => facteur de branchement fini

Performances de la recherche coût min

- Optimalité :
 - => optimale avec les mêmes conditions que la complétude
 - » On explore les nœuds par coût croissant donc forcément la première solution trouvée sera optimale
- Complexité spatiale et temporelle :
 - La recherche de coût min doit mémoriser tous les nœuds de l'arbre de recherche => complexité semblable à celle de la largeur
 - Soit g^* le coût de la solution optimale et ϵ le pas minimal, la profondeur maximum d'un nœud exploré dans l'arbre de recherche sera de g^*/ϵ , soit une complexité de $O(b^{1+g^*/\epsilon})$

Implémentation générique d'une recherche avec critère de priorité

- On ajoute à la structure de nœud une valeur de priorité

Nœud
état : Etat
parent : Nœud
action : Action
coût : Réel
priorité : Réel
Noeud (e : Etat, p : Nœud, a : Action, c : Réel, p : Réel) : Nœud

La priorité représente l'ordre dans lequel les nœuds seront traités

Constructeur de un nœud à partir d'un état, d'un nœud parent, d'une action (ayant permis d'atteindre cet état), du coût associé au chemin, d'une priorité de sélection du noeud

Recherche par priorité sans détection d'états répétés

Explorer (p : Problème) : Nœud ou null

```
Nœud racine = new Nœud(p.étatInitial, null, null, 0, prioritéInitiale) ;  
Liste frontière = new Liste() ;  
frontière.insérer(racine) ;  
tant que non frontière.vide?() faire  
    Nœud n = frontière.oterTête() ;  
    si p.but?(n.état) alors retourner n;  
    pour toute Action a dans p.actions(n.état) faire  
        Nœud sn = new Nœud (a, n, a.résultat(n.état), cout, priorité) ;  
        frontière.insérerCroissant(sn);  
    fin pour  
fin tant que  
retourner null;
```

Coût min sans détection d'états répétés

Explorer (p : Problème) : Nœud ou null

Nœud racine = new Nœud(p.étatInitial, null, null, 0, 0) ;

Liste frontière = new Liste() ;

frontière.insérer(racine) ;

tant que non frontière.vide?() faire

Noeud n = frontière.oterTête() ;

si p.but?(n.état) alors retourner n;

pour toute Action a dans p.actions(n.état) faire

Nœud sn = new Nœud (a, n, a.résultat(n.état), n.cout+a.cout(n.etat),
n.cout+a.cout(n.etat)) ;

frontière.insérerCroissant(sn);

fin pour

fin tant que

retourner null;

Détection d'états répétés

■ Principe

- à tout moment on ne conserve qu'un nœud par état
- variante de l'algorithme de Dijkstra du plus court chemin qui ne considère qu'une partie des sommets du graphe

■ Dijkstra

- On initialise la frontière avec tous les sommets (avec un coût initial infini)
- à chaque itération on prélève le sommet de coût min et on met à jour les coûts

■ Recherche par priorité avec détection d'états répétés

- On ne met initialement que le sommet source dans la frontière
- à chaque itération on prélève le sommet de coût min, on ajoute dans la frontière les nouveaux sommets atteints et si besoin on met à jour les coûts des sommets déjà présents dans la frontière

=> On doit mémoriser les états explorés pour ne pas les remettre dans la frontière

Recherche par priorité avec détection d'états répétés

ExplorerOptimisé (p : Problème) : Nœud ou null

Nœud racine = new Nœud(p.étatInitial, null, null, 0, prioritéInitiale) ;

Liste<Nœuds> frontière = new Liste() ; frontière.insérer(racine) ;

Ensemble<Etat> exploré = new Ensemble();

tant que non frontière.vide?() faire

 Nœud n = frontière.oterTête() ;

 exploré.ajouter(n.état) ;

si p.but?(n.état) alors retourner n;

pour toute Action a dans p.actions(n.état) faire

 Etat se = a.résultat(n.état) ;

si non exploré.contient?(se) alors

 Nœud sn = new Nœud (a,n,se,n.cout+a.cout(n.etat), priorité) ;

s'il n'y a pas de nœud avec le même état se dans la frontière alors

 frontière.insérerCroissant(sn);

sinon modifier la frontière pour que le nœud de plus forte priorité soit conservé ;

fin si

fin pour

fin tant que

 retourner null;