

RÉSOLUTION DE PROBLÈMES

Problème

- Un problème est une collection d'informations que l'agent utilise pour décider quelle(s) action(s) accomplir.
- Définir un problème c'est choisir une abstraction de la réalité en termes :
 - Identification d'un **état initial** (donc choix d'un langage de description d'états du problème)
 - Identification des **actions possibles** par définition d'opérateurs de changement d'état (donc définition de l'ensemble des états possibles du problème)
 - » *Les opérateurs ont parfois des conditions d'applications*

Espace des états

- On appelle **espace des états** (ou **espace de recherche**) d'un problème l'ensemble des états atteignables depuis l'état initial par n'importe quelle séquence d'actions
- Un espace de recherche peut être représenter par un graphe orienté
 - » Les sommets sont les états
 - » Les arcs sont les actions

Résoudre un problème

- Un problème est défini pour un objectif particulier. On doit donc également définir :
 - Une **fondation de test de but atteint** qui détermine si un état du problème correspond à un état but du problème
 - » Par liste d'états but
 - » Par la donnée d'une propriété pour un tel état
- Une **solution** est une séquence d'actions permettant de passer de l'état initial vers un état but
 - Donc un **chemin** dans l'espace des états de l'état initial vers un état but
- Résoudre un problème c'est trouver une/toutes les solutions
 - Pour certains problèmes, une **fondation de coût de chemin** permet de sélectionner une solution préférée parmi l'ensemble des solutions

Formalisation d'un problème

Type de données

Composants

InitialState

Operators

Opérations

$Bool \leftarrow GoalTest(State)$

Fonction qui teste si un état est un but

$Real \leftarrow PathCost(Sequence\ of\ Operator)$

Fonction de coût d'un chemin
(souvent définie comme la somme du coût des opérateurs utilisés)

Problem

InitialState, Operators

Etat(s) initial(aux) de l' agent

Actions possibles de l' agent

GoalTest, PathCost

Connaissance complète vs. incomplète

■ Problèmes à état simple

- On connaît l'état dans lequel on est
- On connaît précisément l'effet des actions
- **On peut calculer à tout moment l'état dans lequel on se trouvera après une action**

■ Problèmes à états multiples

- On ne sait pas exactement dans quel état on se trouve
=> seulement un ensemble d'états possibles
- On ne connaît pas précisément l'effet des actions
- **On ne peut que caractériser par un ensemble d'états la situation où l'on est**

Connaissances complètes vs. incomplètes

- Problèmes non complètement prévisibles
 - Le choix d'une action nécessite de tester l'environnement durant l'exécution
 - » La recherche d'une solution se fait durant la phase d'exécution de la solution en alternance (ex. les problèmes de jeux à 2 joueurs)
 - Le calcul des états atteints par une action est paramétré par les événements extérieurs pouvant modifier l'environnement

Nature du problème

- Problèmes jouets : concis et bien défini
 - » Le taquin, les reines...
 - » La modélisation est facile !
 - » Intéressant pour comparer les différentes stratégies de résolution
- Problèmes du monde réel : complexe et très ouvert
 - » Calcul de routes, voyageur de commerce, navigation de robots...
 - » Difficiles à résoudre dans le cas général (trop de paramètres) d' où importance de la modélisation !

Algorithme de résolution

- Résoudre un problème consiste à trouver une séquence d'actions permettant de passer de l'état initial à un état but : **une solution**
- Il s'agit donc d'effectuer une **recherche à travers l'espace des états**
- L'idée est de maintenir et d'étendre un **ensemble de solutions partielles** : des séquences d'actions qui amènent à des états intermédiaires « plus proche » de l'état but.

Génération des solutions partielles

- Un cycle en 3 phases
 1. Tester si l'état actuel est un état but
 2. Générer un nouvel ensemble d'états à partir de l'état actuel et des actions possibles
 3. **Sélectionner** un des états générés (à cette étape ou précédemment) et recommencer...
- Le choix de l'état à considérer (la sélection) est déterminé par la **stratégie de recherche**

Processus de résolution

- Le processus de résolution de problème consiste donc à construire un **arbre de recherche** qui se superpose à l'espace des états du problème.
- Chaque nœud de l'arbre correspond soit à l'état initial du problème, soit à un développement du sommet parent par un des opérateurs du problème.

Arbres de recherche

- La racine de l'arbre correspond à l'état initial du problème.
- Les feuilles de l'arbre sont :
 - soit des nœuds associés à des états sans action permettant de poursuivre
 - soit des nœuds non encore développés
- Un chemin est une séquence de sommets partant du sommet à une feuille.
 - Son coût est défini par la somme des coûts des opérateurs de chacun de ses sommets.

Nœuds d'un arbre de recherche

Type de données

Node

Composants

State, ParentNode, Operator, Depth, PathCost

State

Etat dans l'espace des états auquel le nœud correspond

ParentNode

Le nœud ayant généré ce nœud

Operator

Opérateur utilisé pour générer ce nœud

Depth

Le nombre de nœuds -1 du chemin de la racine à ce nœud

PathCost

Le coût de ce chemin

Opérations

MakeNode, Expand

Node \leftarrow *MakeNode(State)* Fabrique un nœud à partir d'un état (utilisé pour l'état initial)

Set of Node \leftarrow *Expand(Node, Set of Operator)*

Calcule l'ensemble des nœuds générés par l'application des opérateurs au nœud spécifié

Nœuds d'un arbre de recherche

- On appelle **frontière** l'ensemble des nœuds non encore développés (explorés) de l'arbre de recherche
- Dans cette présentation des algorithmes de recherche la frontière est une liste
 - On sélectionne toujours le nœud en tête de liste
 - La stratégie de la recherche est reportée sur la procédure d'insertion des nœuds dans la liste

Nœuds d'un arbre de recherche

Type de données

Queue

Opérations

MakeQueue, Empty?, RemoveFront, QueuingFn

Queue \leftarrow *MakeQueue(Node)*

Construit une liste à un nœud

Bool \leftarrow *Empty?(Queue)*

Retourne vrai si la liste est vide

Node \leftarrow *RemoveFront(Queue)*

Extrait le nœud en tête

QueuingFn(Set of Node, Queue)

Insère des nœuds dans la liste selon une stratégie particulière

Fonction générale de Recherche

```
Node or nil ← GeneralSearch(Problem p, QueuingFn strategy)
```

// retourne une solution ou un échec

```
Queue frontier←MakeQueue(MakeNode(p.InitialState));
```

```
Loop do
```

```
    if Empty?(frontier) then return nil;
```

```
    Node n←RemoveFront(frontier);
```

```
    if GoalTest(n.state) then return n;
```

```
    frontier←strategy(Expand(n,p.operators),frontier)
```

```
End
```

Performance d'une stratégie de résolution

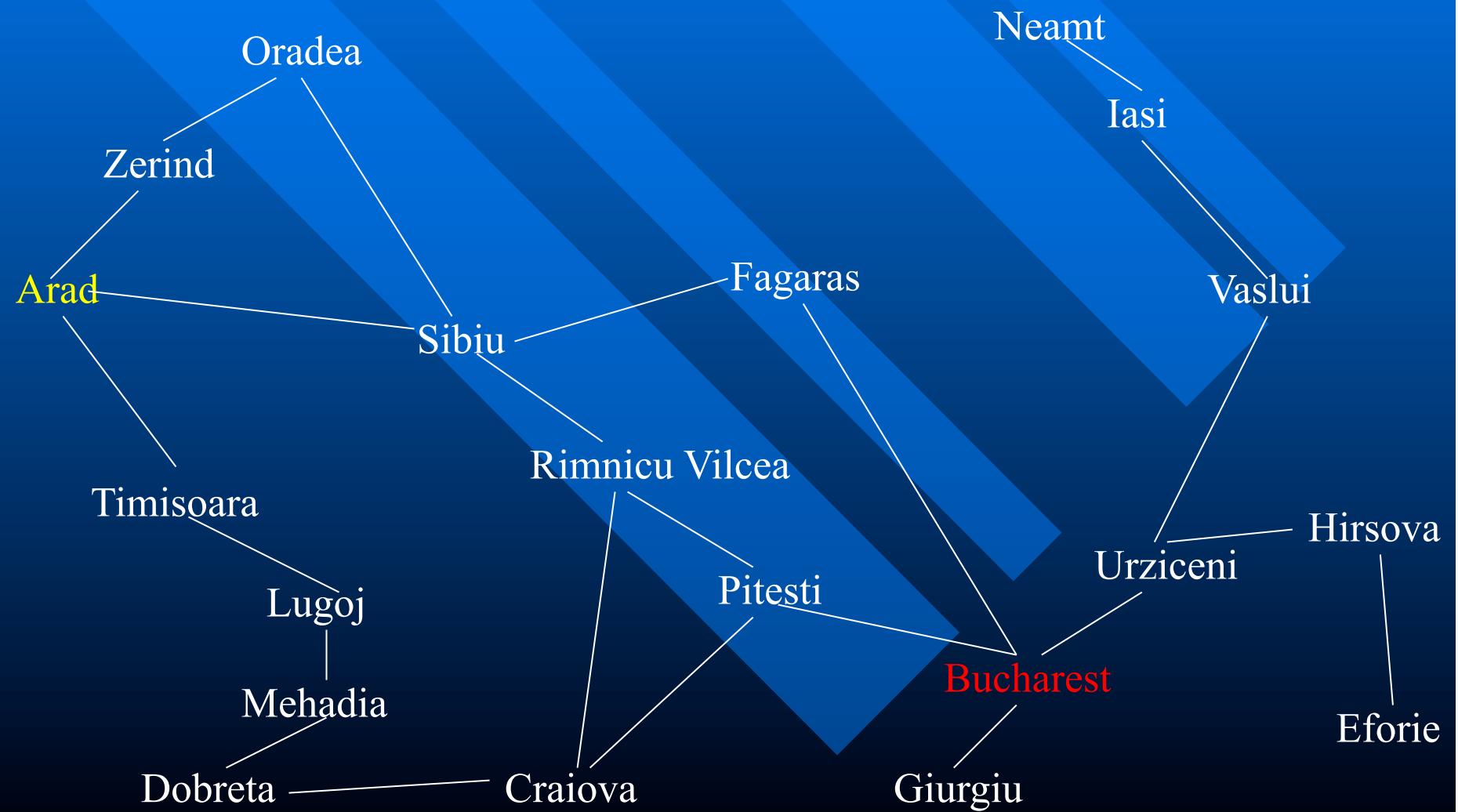
- La performance d'une stratégie de résolution se mesure selon quatre points de vue :
 - **Complétude** : la technique de résolution marche t'elle dans tous les cas ?
 - **Optimalité** : la technique de résolution trouve t'elle une solution de coût minimal ?
 - **Complexité** : la technique est-elle coûteuse
 - » en **temps** ?
 - » en **mémoire** ?

Ne pas confondre performance de la résolution et celle de l'exécution de la solution

Les différentes stratégies

- On distingue les stratégies de recherche simples
 - Sans information autre que la liste des opérateurs et la fonction de test de but atteint
- Des stratégies de recherche heuristique
 - Disposant d' information supplémentaire sur le problème permettant de privilégier certaines branches dans l' arbre de recherche (ou d' en éviter d' autres)

Autre exemple : Recherche d'une route de Arad à Bucharest



Autre exemple : Modèle graphique des routes roumaines

- Les états et les opérateurs de l'agent sont respectivement représentés par les sommets S et les arêtes A du graphe $G=(S,A)$ précédent.
- L'état initial est le sommet *Arad* et l'état but le sommet *Bucharest*.

Initial-State *{Arad}*

Operators *A*

GoalTest *{Bucharest}*

PathCost *Somme(Coût(a)) pour toute arête
a appartenant au chemin considéré*

avec *Coût(a)* donnant la distance entre les deux villes liées par l'arête a

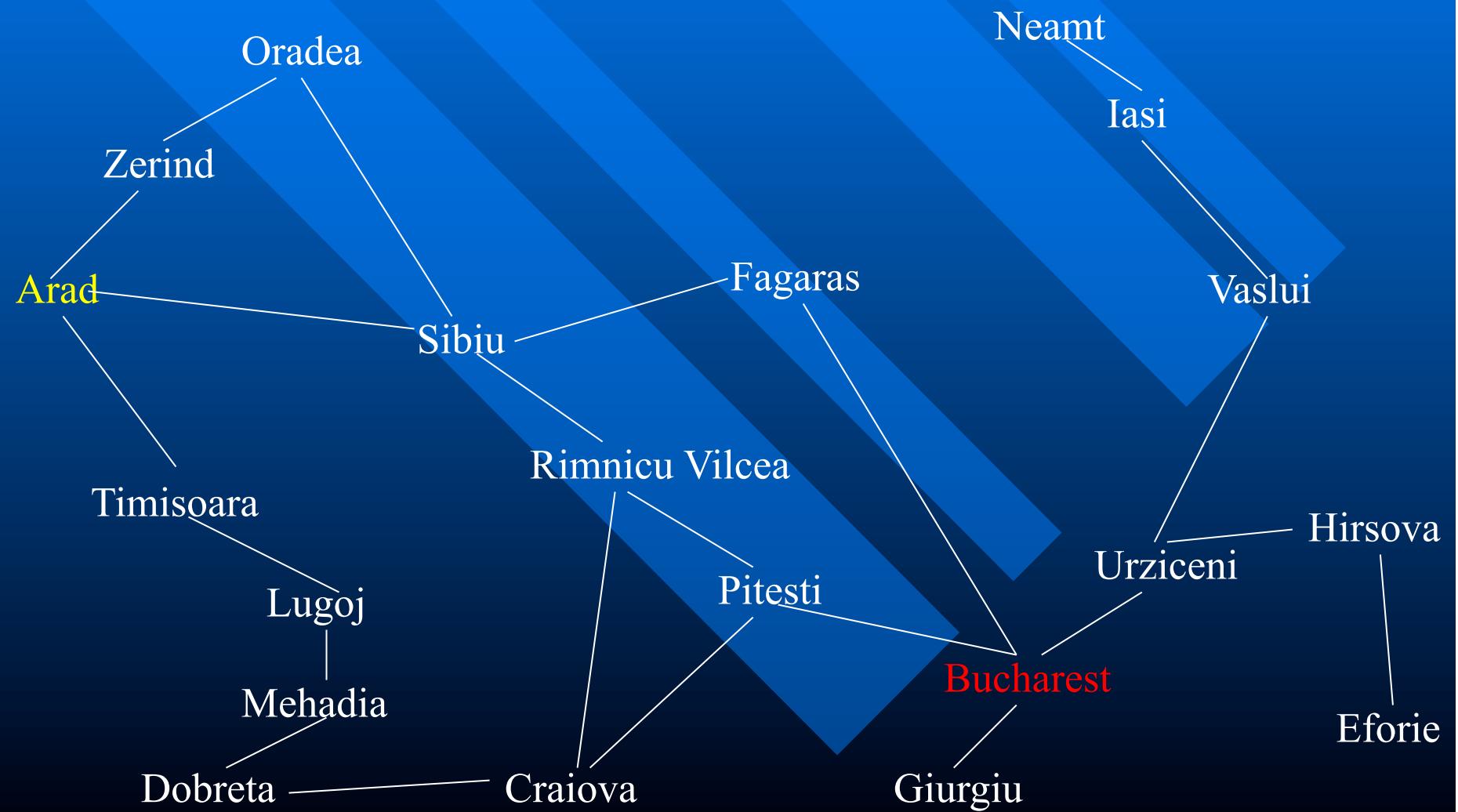
La recherche en largeur

- On développe l'arbre de recherche en largeur
 - Les nœuds de profondeur d sont développés avant ceux de profondeur $d+1$
 - Soit p un problème la recherche en largeur est effectué par un appel à l'algo général avec insertion en fin → la frontière est gérée en file

$BreadthFirstSearch(p) = GeneralSearch(p, \text{EnqueuedAtEnd})$



Autre exemple : Recherche d'une route de Arad à Bucharest



Complétude de la largeur

- Si une solution existe elle sera trouvée
 - Tous les chemins sont étudiés de manière systématique
- Donc stratégie **complète**
 - Même si l'arbre de recherche est infini et/ou si on ne teste pas que l'on est revenu à un état déjà exploré

Optimalité de la largeur

- Trouve une solution la plus proche de la racine
- Donc optimale seulement si
 - le critère d'optimalité diminue avec le nombre d'opérations effectuées
 - et que 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 à chaque expansion)
- On considère $1+b+b^2+b^3\dots+b^d+(b^d-1)b$ 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 !

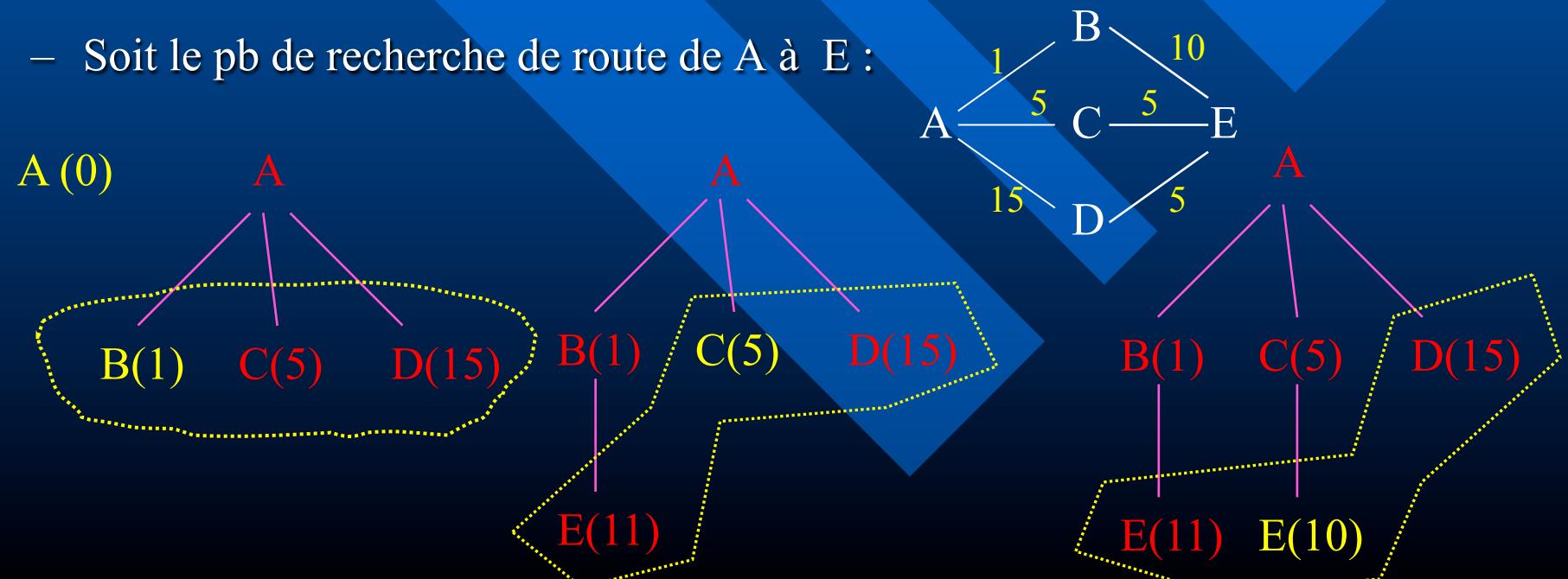
- Supposons qu'une machine soit capable de tester et de développer 1000 nœuds par seconde et qu'un nœud nécessite 100 octets de stockage alors pour un facteur de branchement $b=10$

Profondeur	Nb noeuds	Temps	Espace
3	11101	11 s	1 Mo

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

La recherche par coût

- Algorithme de Dijkstra du plus court chemin
 - On sélectionne parmi les nœuds frontière le nœud dont le coût associé à son chemin depuis la racine est le moins élevé
- UniformCostSearch(p) = GeneralSearch(p, EnQueuedByPathCost)*
- Soit le pb de recherche de route de A à E :



Performances de la recherche par coût

- Complétude :
 - Complète si les coûts sont positifs
- Optimalité :
 - Optimale si le PathCost augmente avec le nombre d'opérateurs:
 $\forall n \text{ } PathCost(\text{successeur}(n)) \geq PathCost(n)$
 - Quand le PathCost est la somme du coût des opérateurs, elle est optimale si les opérateurs n'ont pas de coût négatif
- Complexité spatiale et temporelle :
 - Soit C le coût de la solution, p le coût de l'action min, la profondeur maximum de l'arbre de recherche sera C/p , soit une complexité de $O(b^{C/p})$

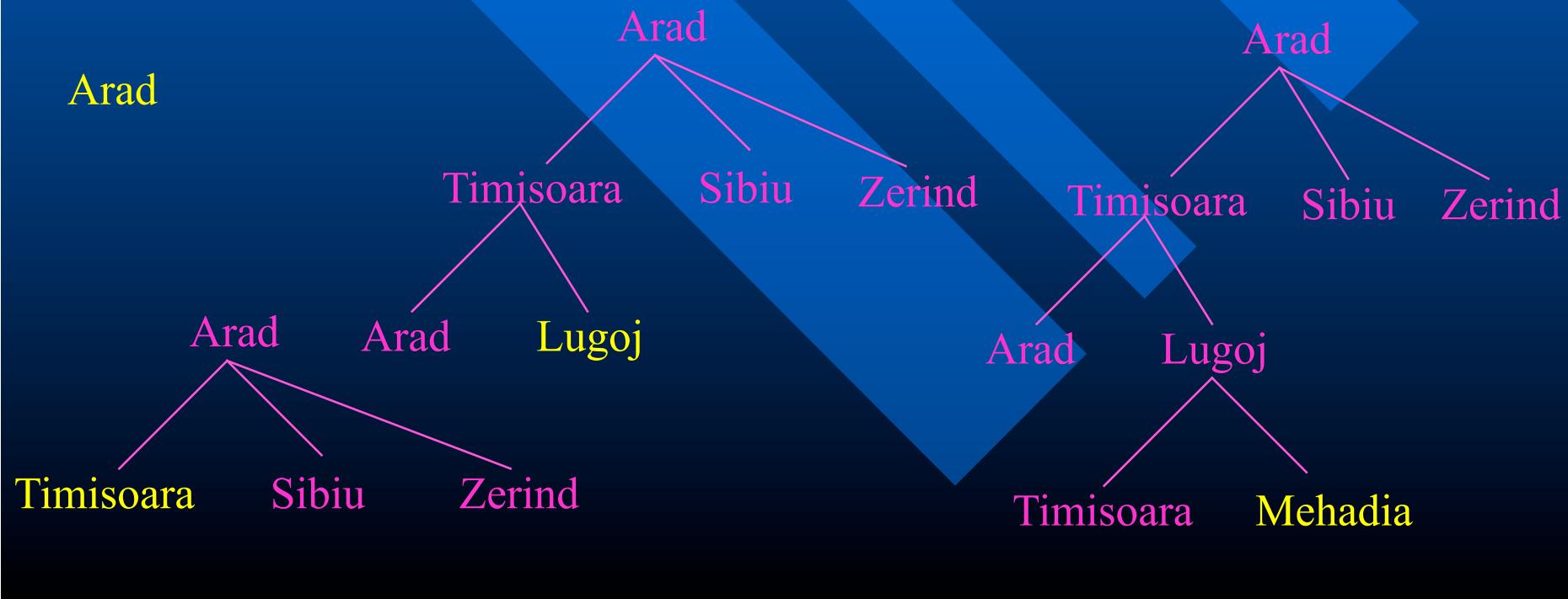
La recherche en profondeur

■ Principe :

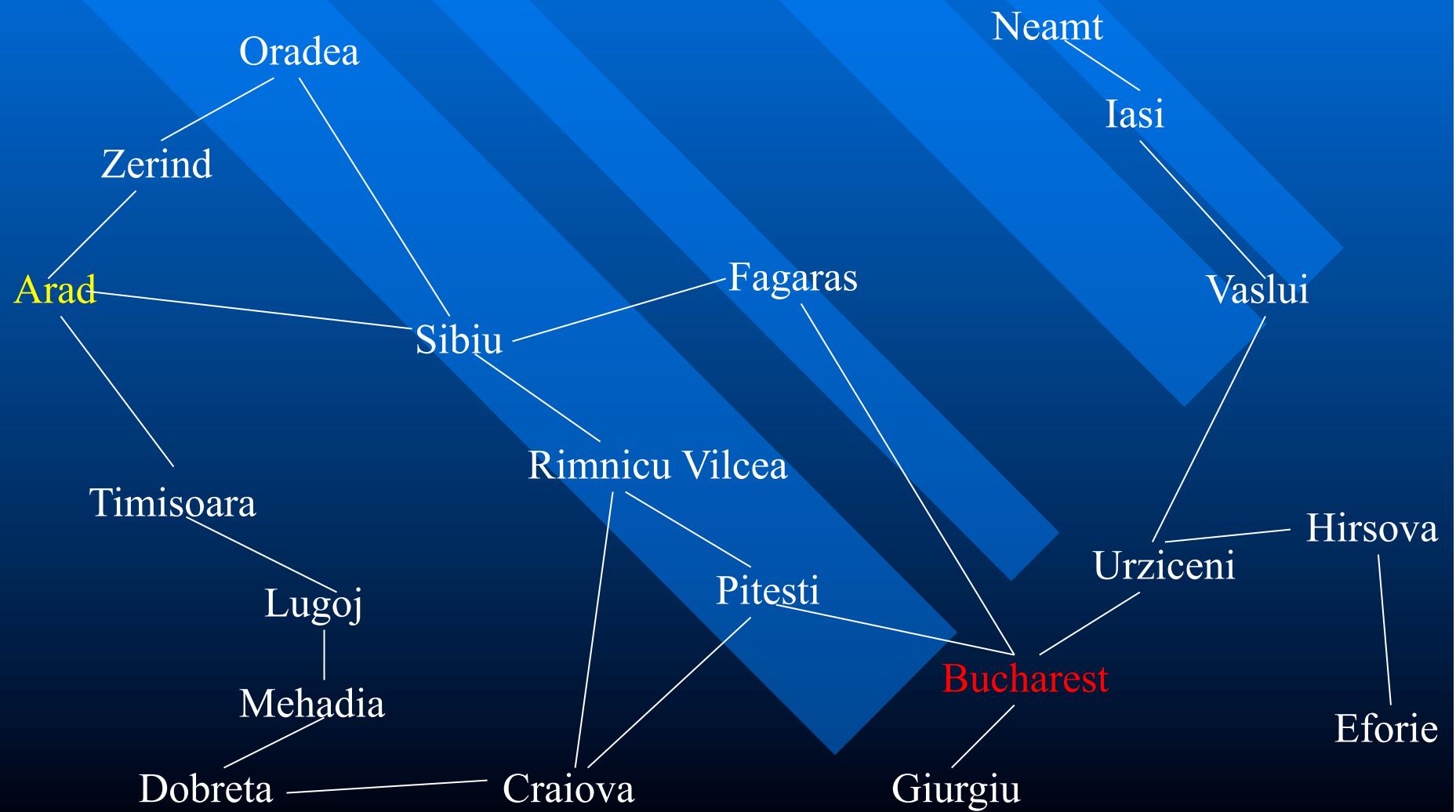
- On développe toujours un des nœuds le plus profond
- On ne remonte que lorsqu'on tombe sur un nœud non but et non développable :
appel à l'algo général avec **insertion en tête** → la frontière est gérée en pile

$$\text{DepthFirstSearch}(p) = \text{GeneralSearch}(p, \text{EnQueuedAtFront})$$

- Avantage : facilement implantable de manière récursive (évite de gérer la pile)



Autre exemple : Recherche d'une route de Arad à Bucharest



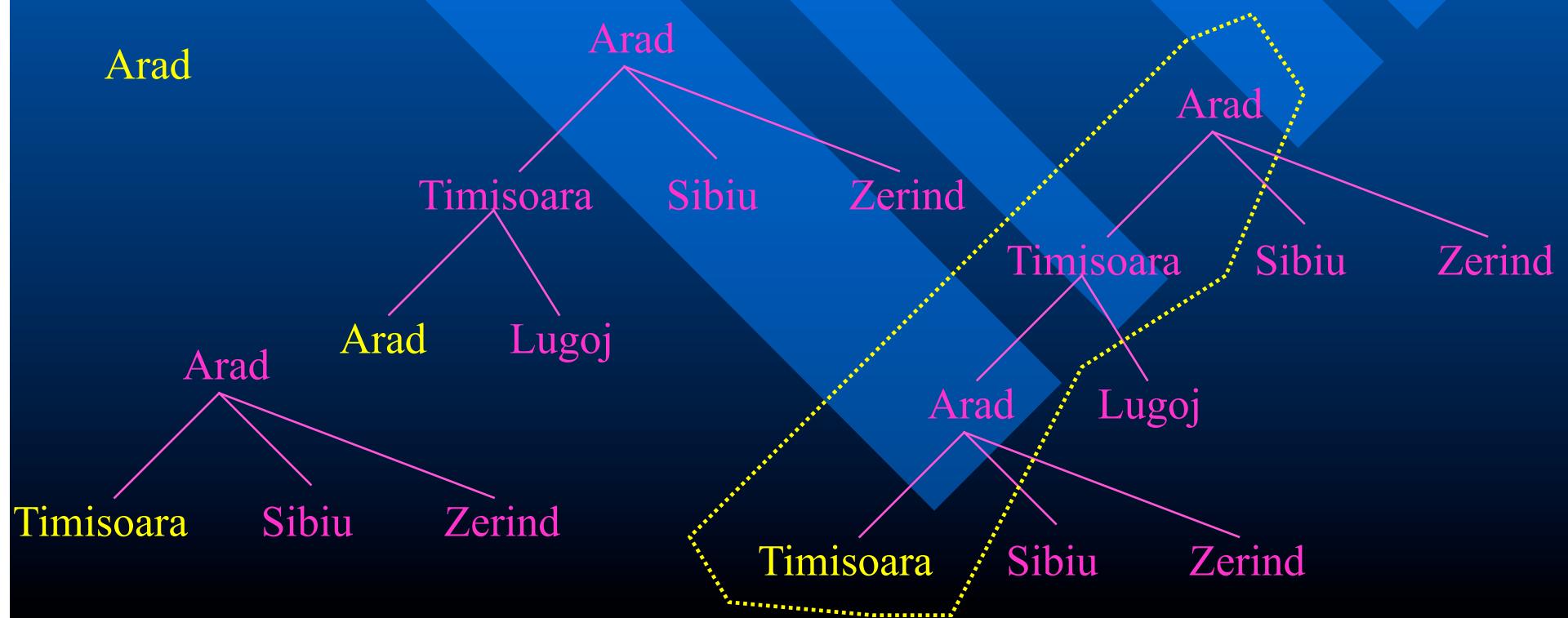
Complexité de la profondeur

- Peu coûteuse en mémoire car on ne mémorise qu'un chemin (plus les nœuds frontières) et non l'arbre entier
- 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*

Problème de la profondeur

- On insère les nœuds développés en tête de liste
 - Pb. : cela peut conduire à développer des branches infinies ou créer des cycles



Performances de la profondeur

- Non complète :
 - à cause des branches infinies potentielles et des cycles
- Non optimale :
 - car elle retourne la première solution rencontrée sans aucune corrélation avec un critère de coût

Donc à éviter pour des problèmes dont les arbres de recherche ont une profondeur infini ou très grande

Rétablir la complétude

- Idée : une **borne** pour ne pas explorer de branches infinies

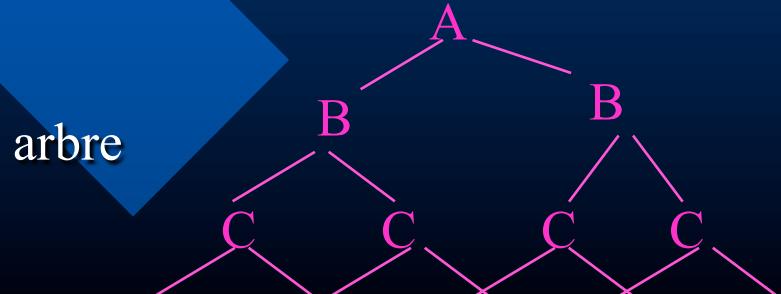


Solution 1: profondeur maximale

- On dispose d'une connaissance de la **profondeur maximale** d'une solution
 - Graphe des états fini :
 - » Ici 20 états dans le graphe donc 19 opérations maximum
 - Diamètre du graphe (max. des longueurs des plus courts chemins entre deux sommets quelconques) : *9 entre Lugoj et Neamt*
- On l'implante en modifiant la définition des opérateurs
si l'état actuel ne dépasse pas la profondeur max. alors générer les états suivants sinon « backtrack »

Suppression des états répétés (1/2)

- Dans de nombreux cas, les recherches perdent du temps à explorer des états qui ont déjà été explorés
 - Dans ce cas les arbres de recherche peuvent être infinis
- Pour se ramener à des arbres finis, on peut couper (« prune ») les branches de l'arbre contenant des états répétés afin de limiter l'arbre à l'espace des états
 - Même si l'arbre n'est pas infini, cela réduit la complexité des algorithmes
 - Exemple :



Suppression des états répétés (2/2)

- 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 cycles : nécessite une comparaison avec tous les nœuds prédecesseurs jusqu'à la racine : $O(d)$
 - Ne pas générer d'état déjà générés : nécessite une comparaison avec tous les nœuds déjà générés et donc un stockage de tous les états du graphe : $O(b^d)$ ou plutôt $O(s)$ où s est la taille de l'espace des états

La recherche heuristique

- Jusqu'à présent nous considérons que nous n'avions aucune autre information sur le problème que les opérateurs et la fonction de test de but.
- Dans certains cas, on possède des informations en plus qui peuvent aider à choisir le nœud successeur et donc permettre d'obtenir une solution plus rapidement presque toujours

Heuristique

- Cela revient à considérer que l'on dispose d'une **fonction d'évaluation d'état** qui retourne le coût d'un chemin d'un état à l'état but le plus proche
 - Une telle fonction ne peut être qu'estimée sinon on peut directement aller à la solution
- Une **heuristique est une fonction qui estime** le coût d'un chemin d'un état à l'état but le plus proche
 - L'heuristique d'un état but doit être 0