

# Intelligence artificielle

resp.: Marie-Laure Mugnier

Algorithmes de recherche

Michel Leclère

[leclere@lirmm.fr](mailto:leclere@lirmm.fr)

# IA

- Résolution de problèmes (algorithmique)
  - Espaces de recherche
  - Technique de « Backtrack »
  - CSP
  - Algos de jeux
  - Planification
- Représentation de connaissances (logique)
  - Bases de connaissances
  - Systèmes à règles
  - Représentation de l'incertain
  - Aide à la décision

# Mais aussi...

- Agents intelligents
  - Modélisation/Programmation
  - Perception/Communication
    - » Vision
    - » TALN
  - Robotique
- Apprentissage

# Support de cours

- "*Artificial Intelligence, a modern approach*" (3<sup>ème</sup> édition), S. Russel & P. Norvig, Prentice Hall, 2010
  - Existe en français, disponible à la BU

# 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)

# 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 **fonction 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 **fonction de coût de chemin** permet de sélectionner une solution préférée parmi l'ensemble des solutions

# 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 !

# Formalisation d' un problème à état simple

## 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*

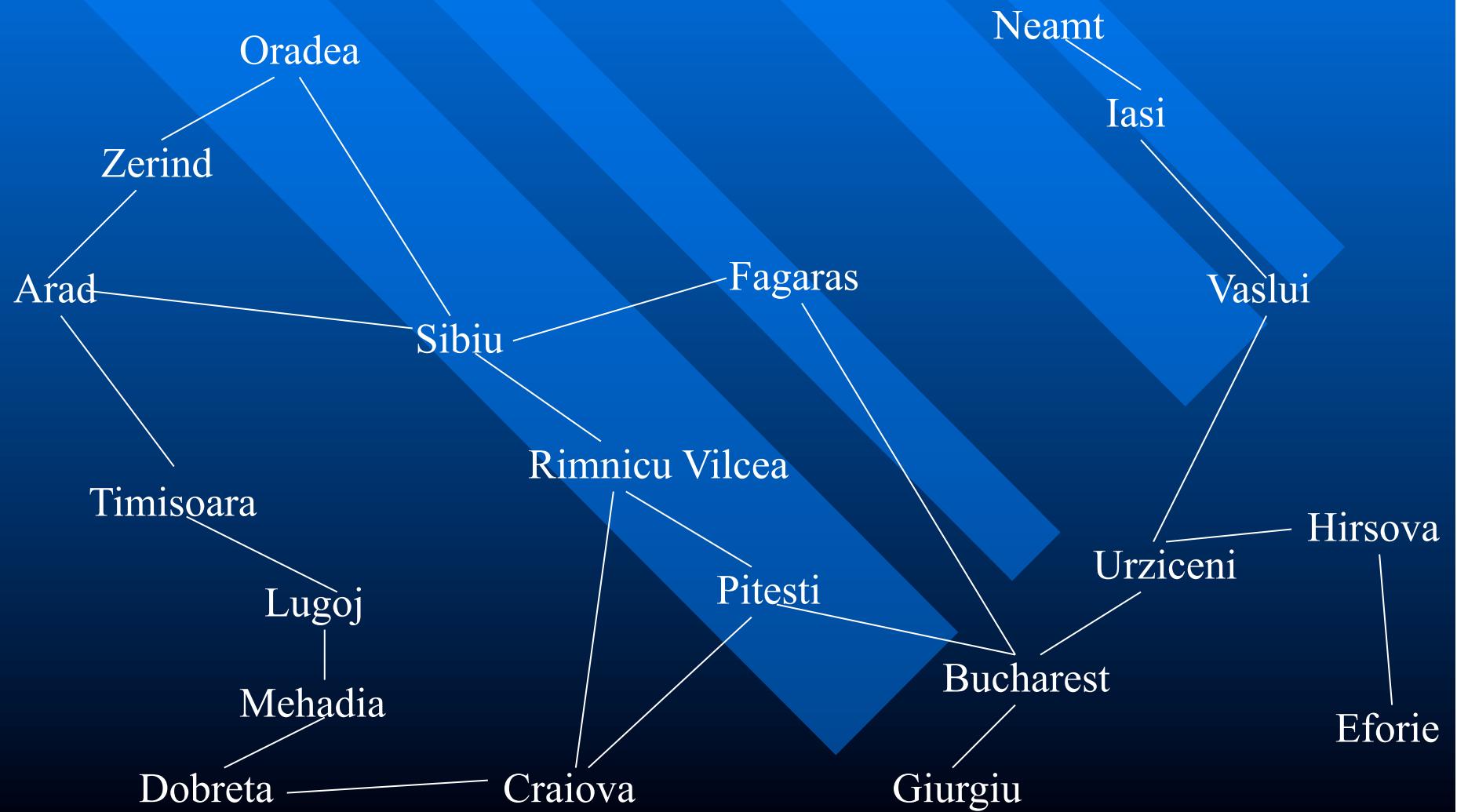
### *InitialState, Operators*

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

Actions possibles de l' agent

### *GoalTest, PathCost*

# Exemple 1 : Recherche d' une route de Arad à Bucharest



# Exemple 1 : 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(CostFunction(a)) pour tout  
a appartenant au chemin considéré

Pour tout a appartenant à A,

CostFunction(a) distance entre les deux villes liées par a

# 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 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 correspondent à des états sans successeur dans l'arbre ou des nœuds qui n'ont pas encore été développés.
- Un chemin est une séquence de sommets partant du sommet à une feuille.
  - Le coût d'un chemin est défini par la somme des coûts de chaque opérateur intervenant dans la construction du chemin.

# 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 nœud 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 de l' arbre de recherche
- On choisit de gérer la frontière comme une liste
  - On sélectionne toujours le nœud en tête de liste
  - La stratégie de 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(Set of Node)* Construit une liste de nœuds

*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  $\leftarrow$  GeneralSearch(Problem p, QueuingFn strategy)  
*// retourne une solution ou un échec*

Queue nodes  $\leftarrow$  MakeQueue(MakeNode(p.InitialState));

Loop do

    if Empty?(nodes) then return nil;

    Node n  $\leftarrow$  RemoveFront(nodes);

    if GoalTest(n.state) then return n;

    nodes  $\leftarrow$  strategy(Expand(n,p.operators),nodes)

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** ?

Attention à ne pas confondre la performance de la résolution et celle de l'exécution de la solution

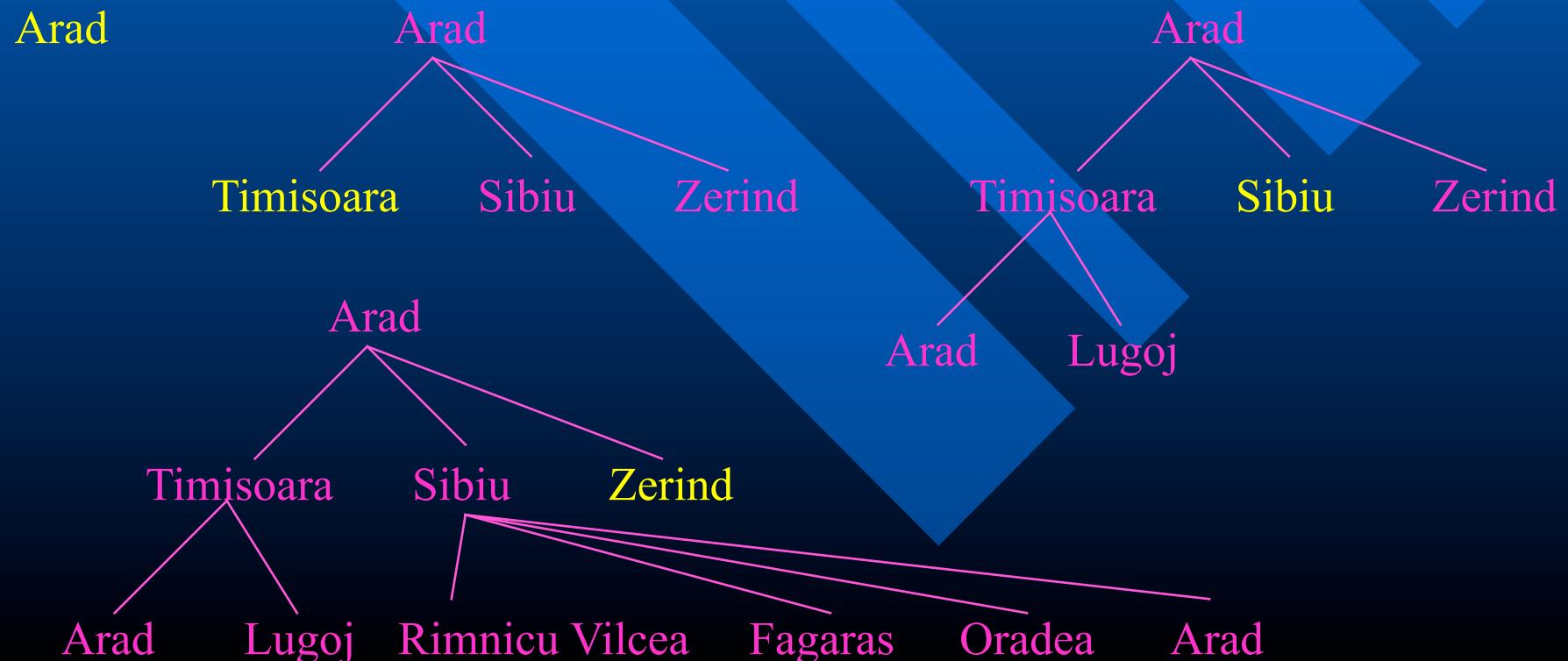
# Les différentes stratégies

- 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)

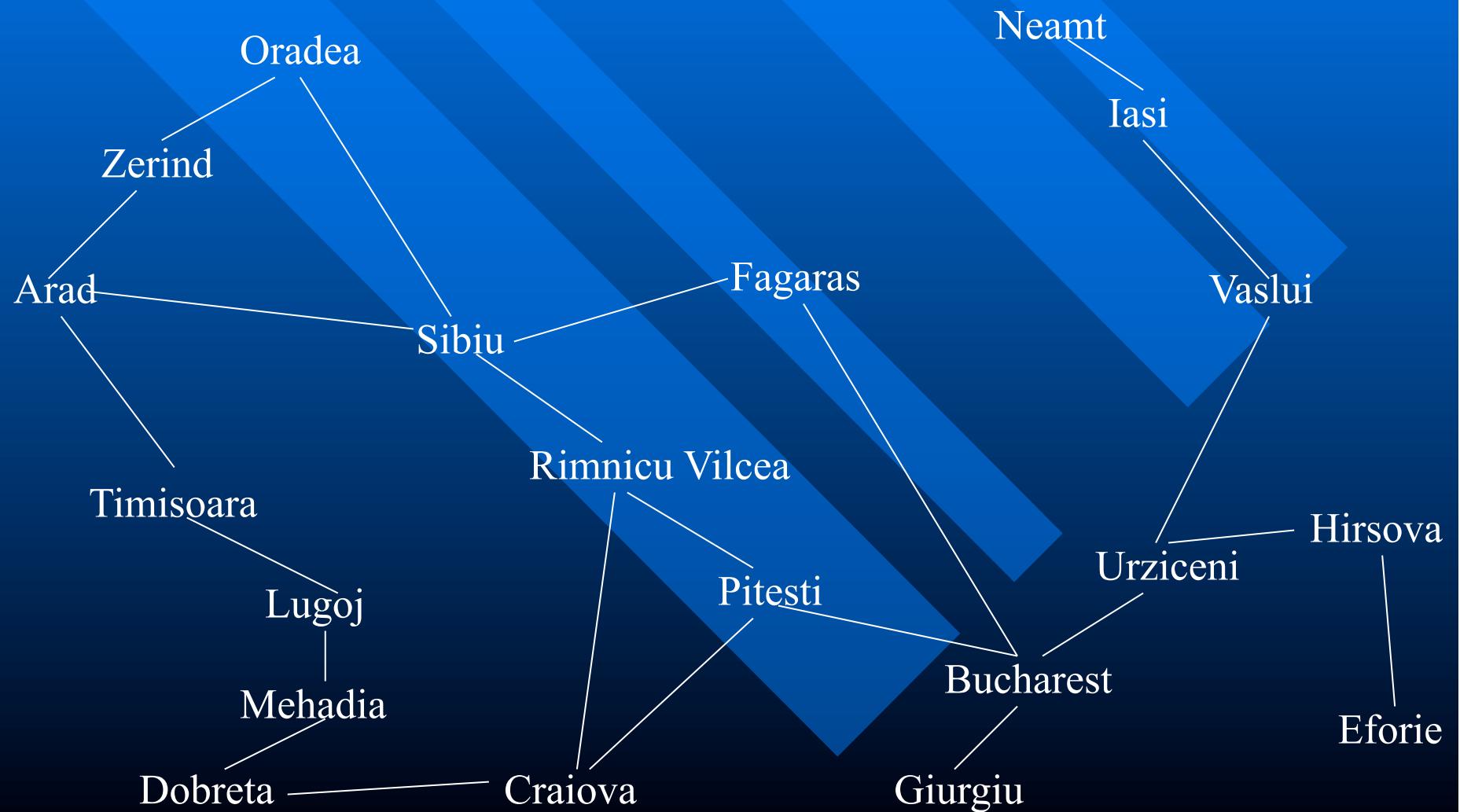
# La recherche en largeur

- 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

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



# Exemple 1 : 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 »

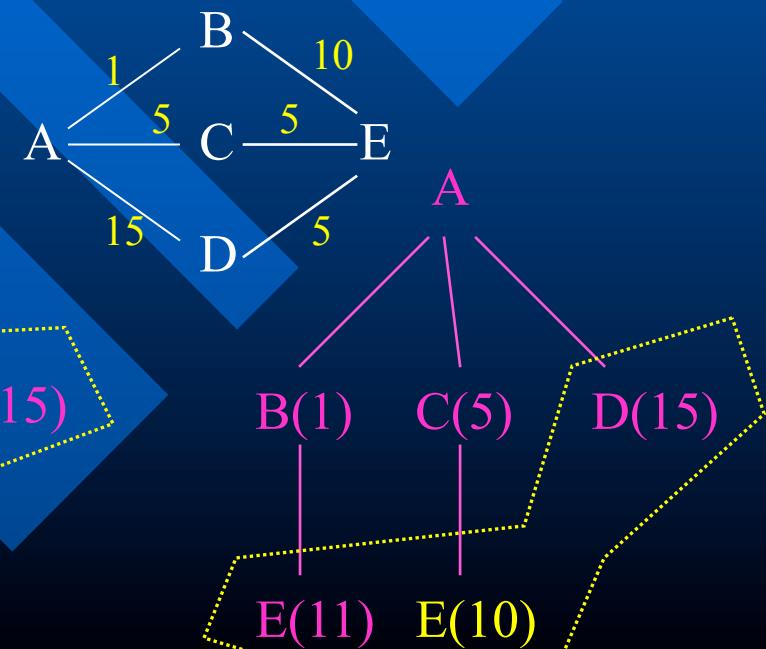
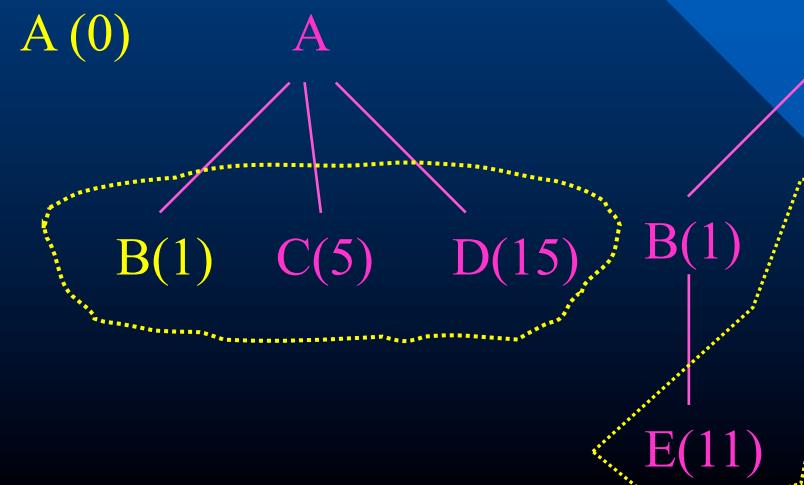
# La recherche par coût

## ■ 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  $O(b^{C/p})$

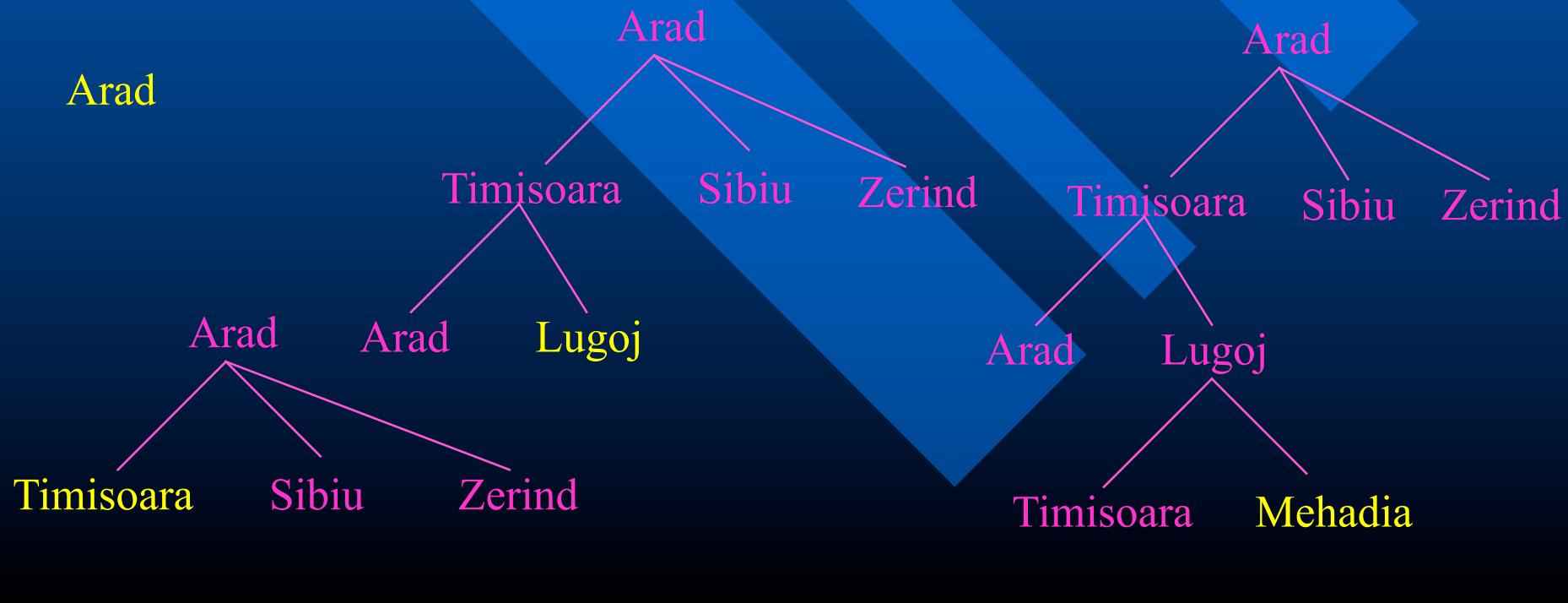
# La recherche en profondeur

## ■ Principe :

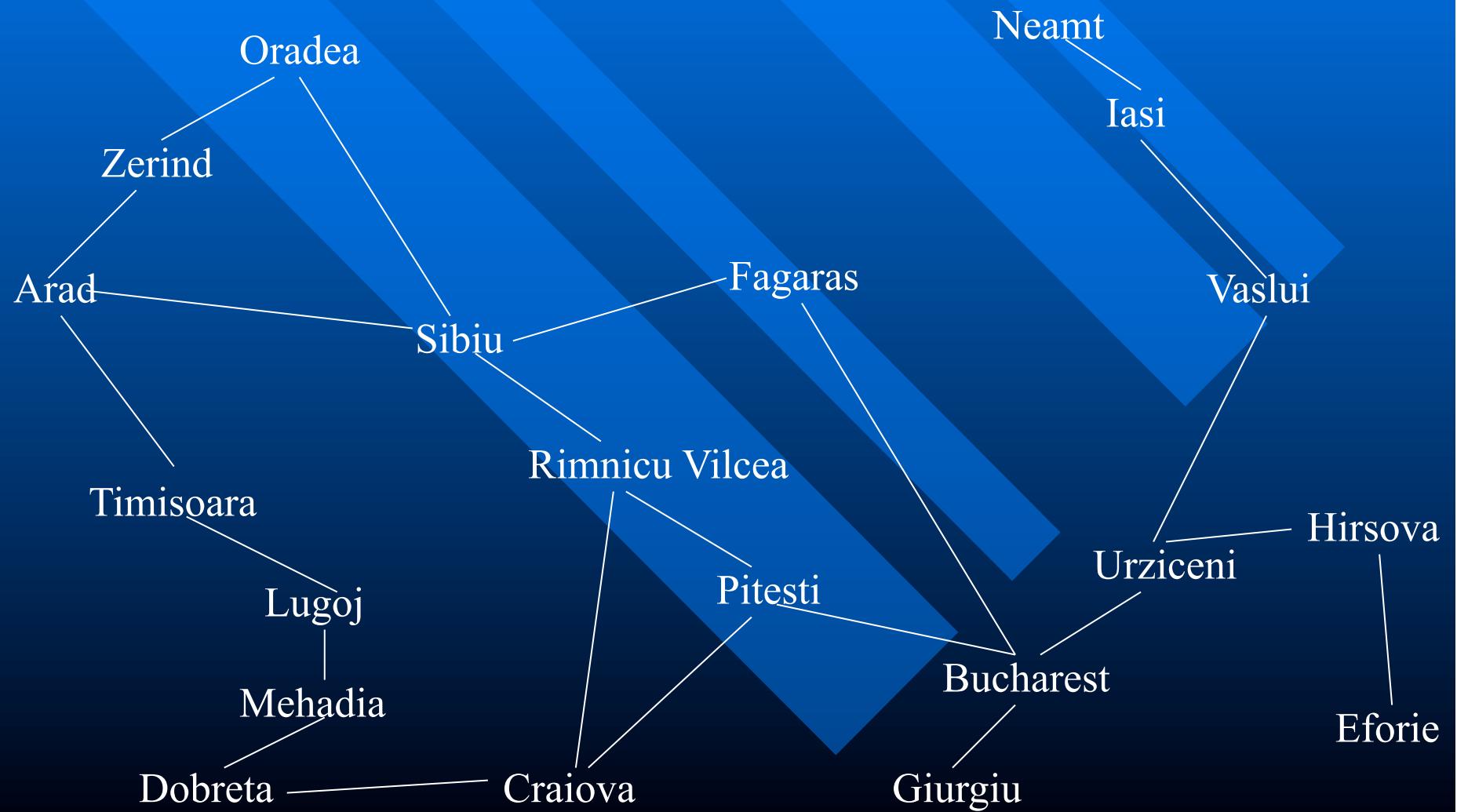
- On développe toujours un des nœuds le plus profond
- On ne remonte (on « backtrack ») sur un nœud plus haut que lorsqu'on tombe sur un nœud non but et non développable : appel à l'algo général avec insertion en tête

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

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



# Exemple 1 : Recherche d' une route de Arad à Bucharest



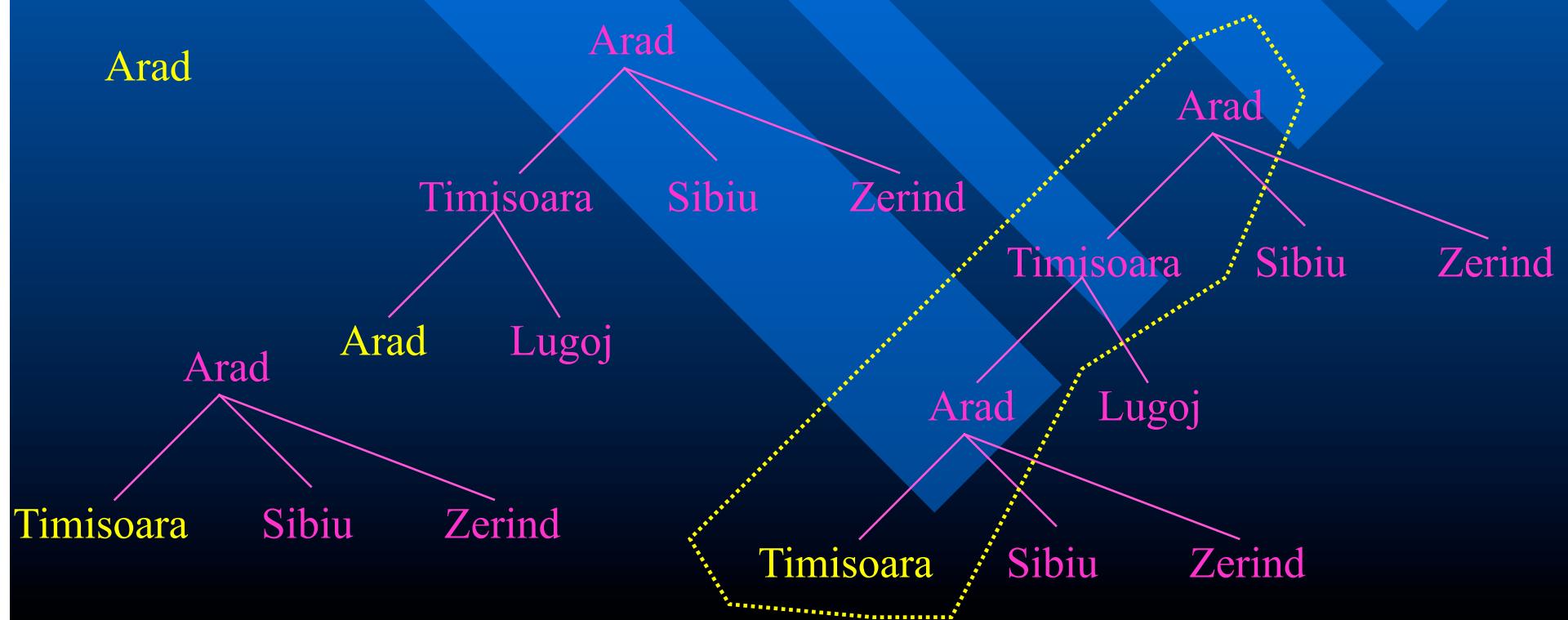
# Complexité de la profondeur

- 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 »*

# Solution 2 : Recherche en profondeur itérative

- On essaye de combiner les avantages de la largeur et de la profondeur : *on effectue une recherche en profondeur avec une profondeur maximale 0, puis 1, puis 2...*

Première itération :  $m=0$

Arad

prof. max. atteinte

# Solution 2 : Recherche en profondeur itérative

- On essaye de combiner les avantages de la largeur et de la profondeur : *on effectue une recherche en profondeur avec une profondeur maximale 0, puis 1, puis 2...*

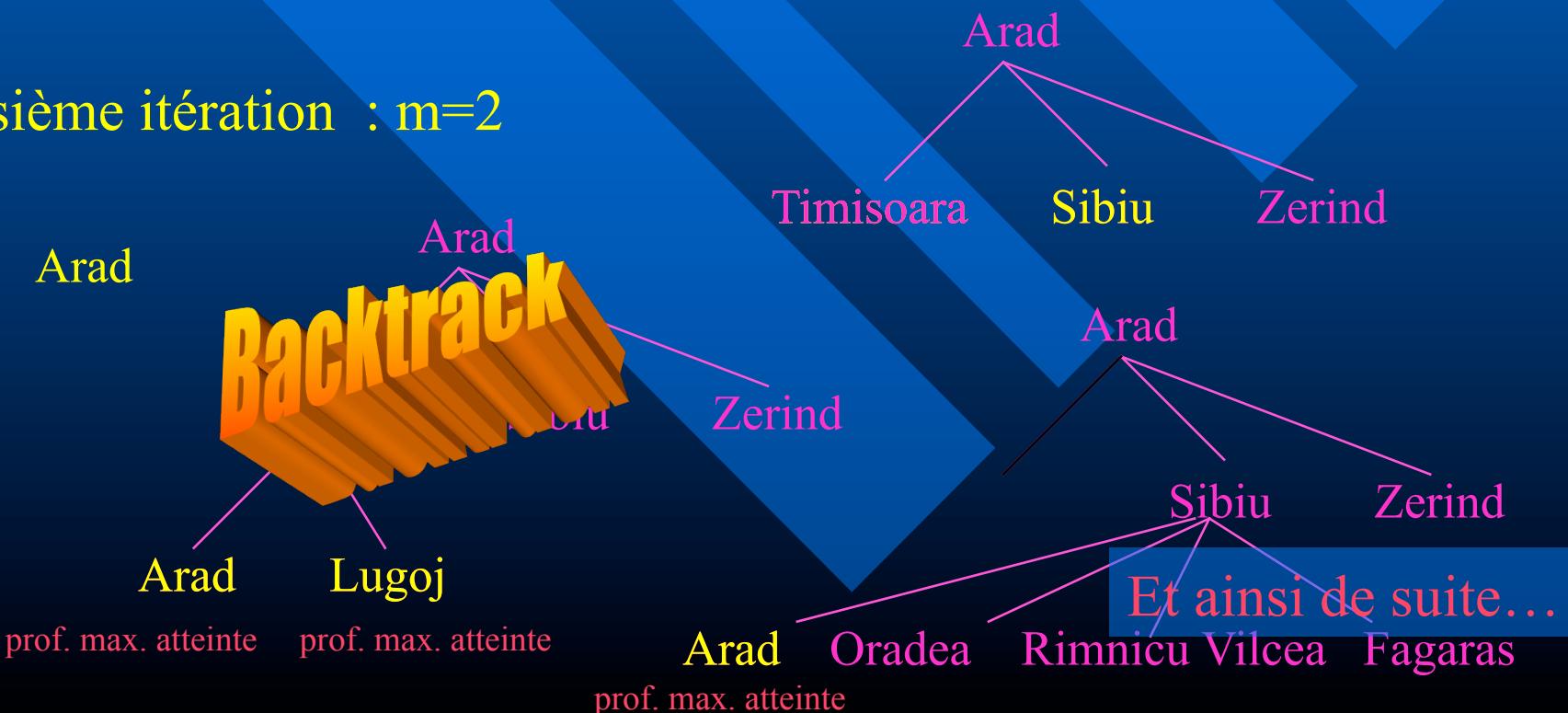
Deuxième itération :  $m=1$



# Solution 2 : Recherche en profondeur itérative

- On essaye de combiner les avantages de la largeur et de la profondeur : *on effectue une recherche en profondeur avec une profondeur maximale 0, puis 1, puis 2...*

Troisième itération :  $m=2$



# Recherche en profondeur itérative (1/2)

```
Node ← IterativeDeepeningSearch(Problem p)
// retourne une solution si elle existe !
for depth←0 to ∞ do
    Node sol←DepthLimitedSearch(p,depth);
    if sol≠nil then return sol;
end
```

```
Node or nil ← DepthLimitedSearch(Problem p, Int d)
// retourne une solution ou nil
Modification de DepthFirstSearch par prise en compte de la profondeur max d
```

# Recherche en profondeur itérative (2/2)

## ■ Performances

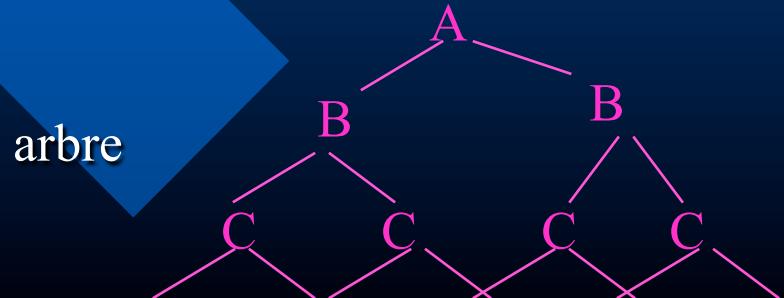
- Complétude : complète
- Optimalité : idem recherche en largeur
- Complexité spatiale: idem recherche en profondeur:  $bd$
- Complexité temporelle :
  - » idem recherche en largeur  $O(b^d)$  bien que les nœuds des niveaux moins profonds que la solution soient développés plusieurs fois :  $(d+1)1 + (d)b + (d-1)b^2 \dots + (1)b^d$

*Pour  $d=4$  et  $b=10$  on aura 12345 générations et tests au lieu de 11111*

**Idéale pour des problèmes dont l'espace de recherche est large et la profondeur inconnue**

# Suppression des états répétés (1/2)

- Dans de nombreux cas, les recherches perdent du temps à développer des états qui ont déjà été développés
  - 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

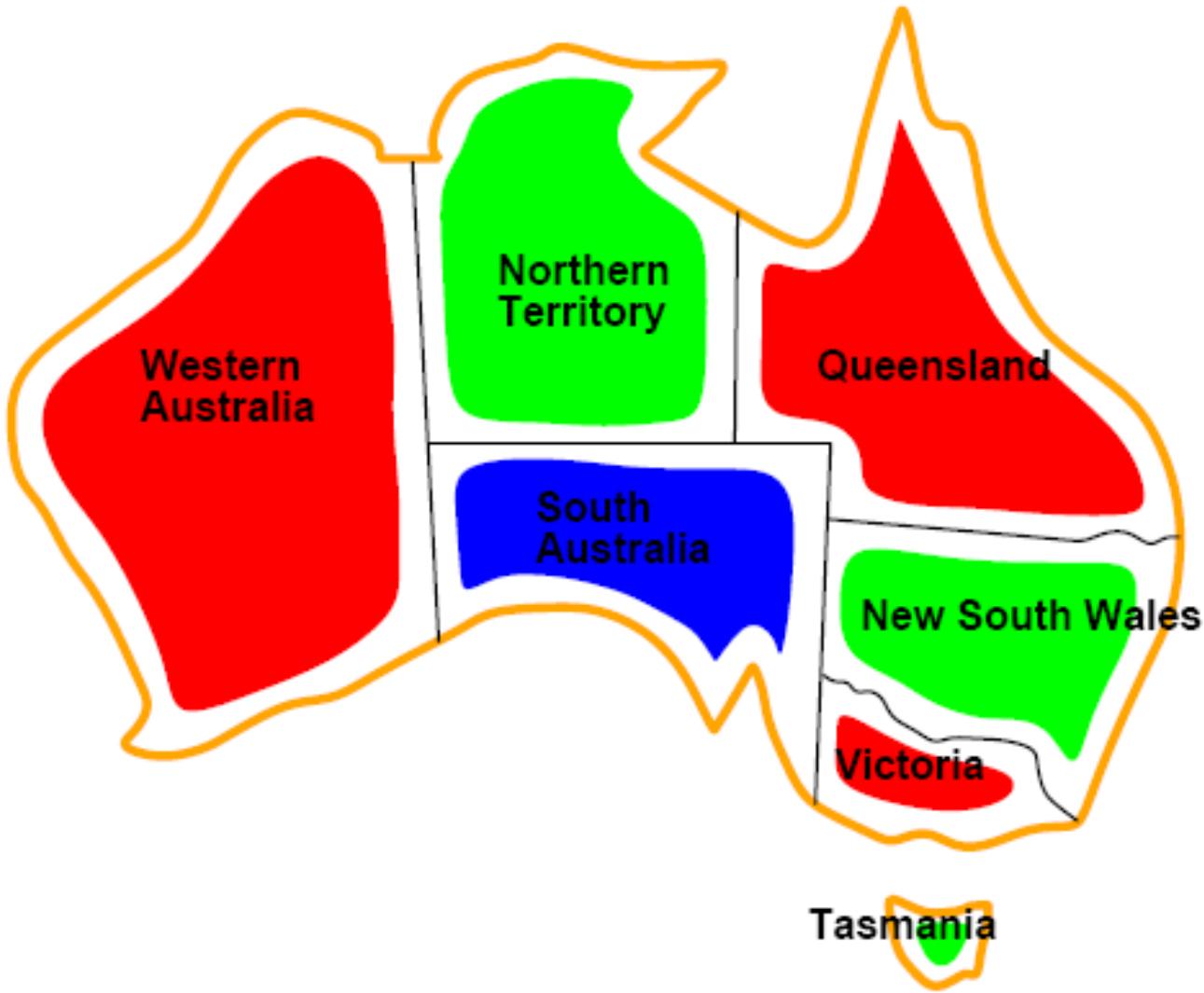
# Les CSP : Constraint Satisfaction Problem

- Un CSP est un type de problème particulier
  - Les états sont définis par les **valeurs** d'un ensemble de **variables**
    - » Les variables sont caractérisées par un **domaine** de valeurs possibles associé à chaque variable
  - Les actions sont des **assignations** de valeurs aux variables et/ou des **retraits** de valeurs possibles aux domaines
  - La fonction de test de but est un ensemble de **contraintes** auquel les valeurs des variables doivent satisfaire

# Exemple : problème de coloration de carte



# Une solution



# CSP vs. Problème Standard

- Les CSP définissent une structure particulière pour les états : **un ensemble de variables partiellement valuées**, et une structure particulière pour la fonction but : **un ensemble de contraintes**
- Les algorithmes de recherche d' une solution peuvent tirer parti de cette structuration
  - On passe d'une vision exploration d'un ensemble d'états : d'un état initial à un état but
  - À une vision **résolution pas à pas** d'un problème : de l'étape où aucune variable n'est assignée à l'étape où toutes les variables sont assignées
- Les techniques de résolution dépendent de la nature des domaines et contraintes

# Réseau de contraintes

- Un réseau de contraintes est un triplet  $P=(X,D,C)$ 
  - $X=\langle x_1, x_2, \dots, x_n \rangle$  est un tuple fini de variables
  - $D=D(x_1) \times \dots \times D(x_n)$  est le domaine de  $X$  où chaque  $D(x_i)$  (le domaine de la variable  $x_i$ ) est fini et donné en extension.
  - $C=\{c_1, \dots, c_m\}$  est un ensemble de contraintes
    - » Chaque contrainte  $c_i$  est une relation (un ensemble de tuples) définie sur un tuple de variables  $X(c_i)=\langle x_{i1}, \dots, x_{ik} \rangle$  où  $k$  est l'arité de la contrainte.
    - »  $c_i$  est donc un sous-ensemble de  $D(x_{i1}) \times \dots \times D(x_{ik})$

# Assignation et Solution

- Soit  $P=(X,D,C)$ 
  - Une **assignation**  $A$  sur un sous-ensemble  $Y$  des variables de  $X$  est une application qui associe à chaque variable  $x$  de  $Y$  une valeur de son domaine  $D(x)$
  - Une assignation  $A$  **viole une contrainte**  $c$  ssi
    1.  $X(c) \subseteq Y$
    2.  $A[X(c)] \notin c$
  - $A$  est **localement consistante** si  $A$  ne viole aucune contrainte de  $C$
  - Une **solution** est une assignation sur  $X$  qui ne viole aucune contrainte de  $C$ 
    - » L'ensemble des solutions de  $P$  est notée  $\text{Sol}(P)$

# Problème de satisfaction de contraintes

- Un CSP est le problème défini par :
  - Instance : un réseau de contraintes  $P=(X,D,C)$
  - Question :  $\text{Sol}(P) \neq \emptyset$
- CSP est NP-complet

# Variables, Domaines, Contraintes

- Propriété : Tout CSP n-aire peut être ramené à un CSP binaire
  - Unaire : ?
    - $1 < a < 5 \rightarrow ?$

# Variables, Domaines, Contraintes

- Propriété : tout CSP n-aire peut être ramené à un CSP binaire
  - Unaire : restriction de domaine
    - $1 < a < 5 \rightarrow D(a) = \{2,3,4\}$

# Variables, Domaines, Contraintes

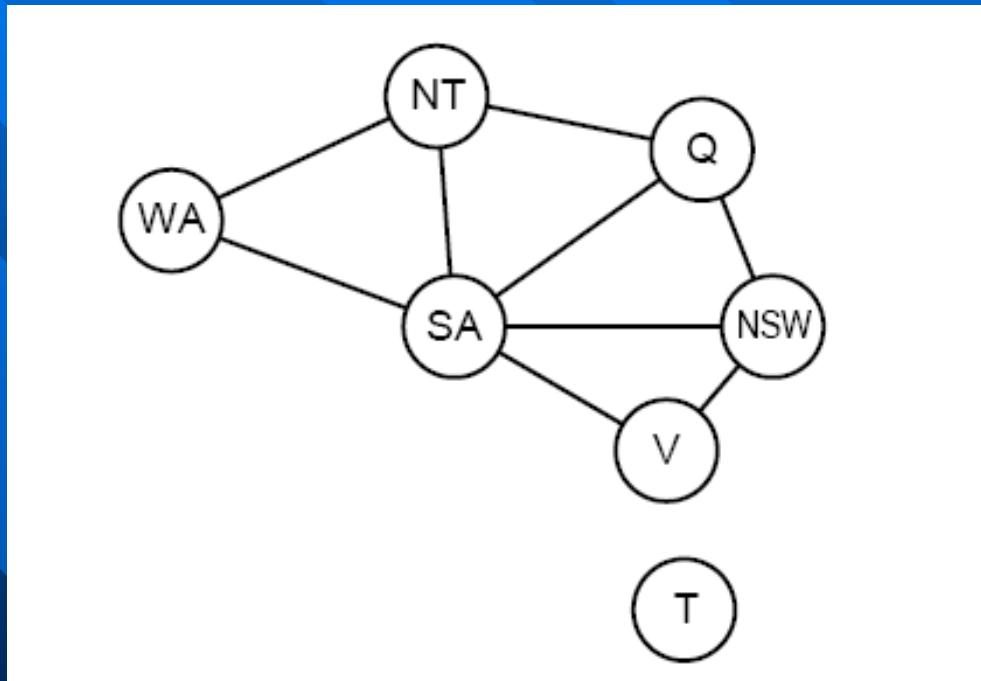
- Propriété : tout CSP n-aire peut être ramené à un CSP binaire
  - Unaire : restriction de domaine
    - $1 < a < 5 \rightarrow D(a) = \{2,3,4\}$
  - N-aire avec  $n > 2$  : ?
    - $a+b=c \rightarrow ?$

# Variables, Domaines, Contraintes

- Propriété : tout CSP n-aire peut être ramené à un CSP binaire
  - Unaire : restriction de domaine
    - $1 < a < 5 \rightarrow D(a) = \{2,3,4\}$
  - N-aire avec  $n > 2$  : introduction de nouvelles variables dont les domaines sont des tuples
    - $a+b=c \rightarrow$ 
      - $X = \langle a, b, c, ab \rangle$
      - $D(ab) = D(a) \times D(b)$
      - $C = \{a=\text{Proj1}(ab), b=\text{Proj2}(ab), \text{Proj1}(ab)+\text{Proj2}(ab)=c\}$

# Représentation graphique d'un CSP binaire

- Représentation
  - Les variables X → des sommets
  - Les contraintes C → des arêtes



- Les algorithmes de recherche peuvent alors tirer parti de la structure du graphe
  - Ex. : T est un pb indépendant, SA est très constraint

# Représentation d'un CSP n-aire

- Un multi-hypergraphe dont les sommets sont les variables et les arêtes sont les contraintes que l'on représente par son biparti d'incidence :
  - des sommets variables
  - des sommets contraintes
  - des arêtes indiquant la participation d'une variable à une contrainte
- On peut étiqueter
  - les sommets variables par leur domaine
  - les sommets contraintes par leurs tuples
  - les arêtes adjacentes à une contrainte par la position de la variable dans la contrainte

# Exemple :

- Les puzzles arithmétiques codés

$$\begin{array}{r} \text{T W O} \\ + \text{T W O} \\ \hline = \text{F O U R} \end{array}$$

# Résolution d' un CSP : algo naïf

- L' état initial est l' état dans lequel aucune variable n' est assignée :  $A = \{\}$
- Une action consiste à choisir une variable non assignée et à lui associer une des valeurs de son domaine :
  - $A \leftarrow A \cup \{(x_i, v)\}$  où  $v \in D(x_i)$
  - Le facteur de branchement est donc borné par la somme des tailles des domaines
  - La profondeur de l'arbre de recherche est naturellement bornée par le nombre de variables
- Lorsque toutes les variables sont assignées, la fonction de test de but vérifie si les contraintes sont satisfaites ou pas
  - A est il une solution de P ?

On peut donc utiliser un algorithme de recherche en profondeur  $\rightarrow O((\sum_i |D_i|)^{|X|})$

# Propriétés de l' espace de recherche

- L'ordre d'assignation des variables n'affecte pas la solution
  - Quelque soit le chemin emprunté, une même assignation des variables définit un seul état
- Lors du déroulement de la recherche en profondeur on choisit un ordre d' assignation des variables

$$O(\Pi_i |D_i|)$$

# Propriétés de l'espace de recherche

- Certaines branches de l'arbre de recherche peuvent être coupées en tirant parti du fait que le test de but est un ensemble de contraintes
  - Si une assignation d'un sous-ensemble des variables ne viole un sous-ensemble des contraintes, il est inutile d'étendre cette assignation car elle ne conduira qu'à des « échec ».
  - Exemple :
    - »  $A=\{(WA, Red), (NT, Red)\}$  ne pourra jamais être prolongé en un  $A'$  satisfaisant car il viole la contrainte  $c=(WA \neq NT)$

# Résolution d'un CSP : Backtracking algorithm

- L'algorithme prend en entrée
  - Un réseau de contraintes
  - Une assignation partielle localement consistante
- On teste à chaque extension de l'assignation partielle courante si elle ne viole pas de contraintes
  - Si ok, on continue à développer la branche
  - Sinon, on « backtrack » immédiatement

# Backtracking algorithm

BacktrackingSearch(Problem p) = BT({},p);

Fonction BT(Assignation a, Problem p) : Booléen

Début

si |a| = |X| alors  
        afficher a;  
        retourner true;  
    finsi;

    x  $\leftarrow$  ChoixVariableNonAssignée(X, a);  
    pour tout v  $\in$  Tri(D(x)) faire

si Consistant(a  $\cup$  {(x,v)}, C) alors  
            |     si BT(a  $\cup$  {(x,v)}, p) alors retourner true; finsi;  
            finsi;  
        finpour;  
    retourner false;

Fin

# Exemple



Variables  $WA, NT, Q, NSW, V, SA, T$

Domains  $D_i = \{red, green, blue\}$

Constraints: adjacent regions must have different colors

e.g.,  $WA \neq NT$  (if the language allows this), or

$(WA, NT) \in \{(red, green), (red, blue), (green, red), (green, blue), \dots\}$

# Améliorations du Backtrack

## ■ Les pistes générales

- Comment choisir l'ordre des variables
  - » Fonction **ChoixVariableNonAssignée**
- Comment choisit l'ordre des valeurs
  - » Fonction **Tri**
- Comment détecter les incohérences au plus tôt
  - » Fonction **Consistant**

## ■ Etude des propriétés structurelles des problèmes

- Composantes connexes, structure arborescente...

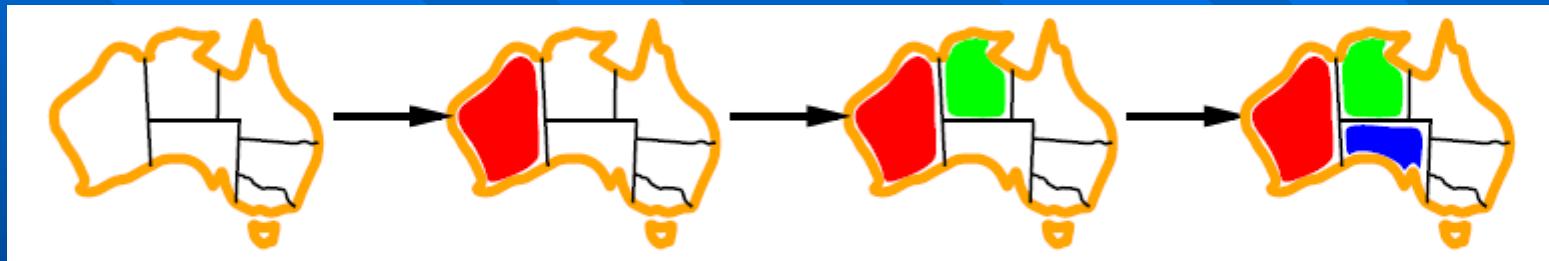
# Les idées d'heuristiques

- Choix de la variable la plus contrainte
  - Celle qui a le plus petit nombre de valeurs restantes (si 0 alors échec immédiat)

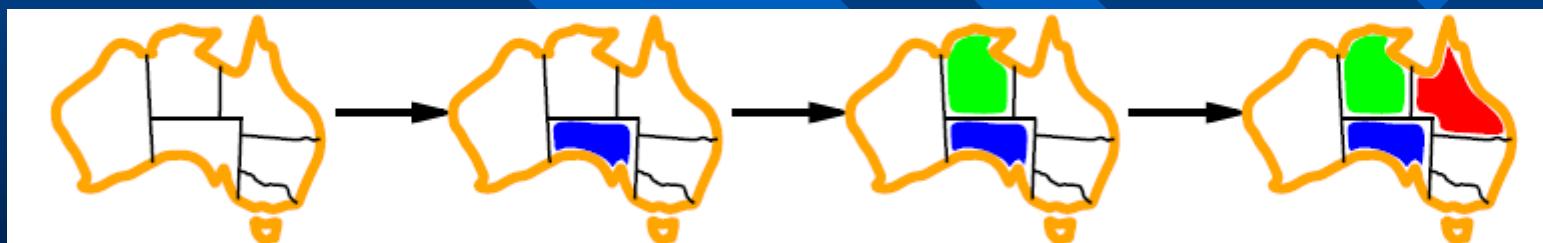


# Les idées d'heuristiques

- Choix de la variable la plus contrainte
  - Celle qui a le plus petit nombre de valeurs restantes (si 0 alors échec immédiat)

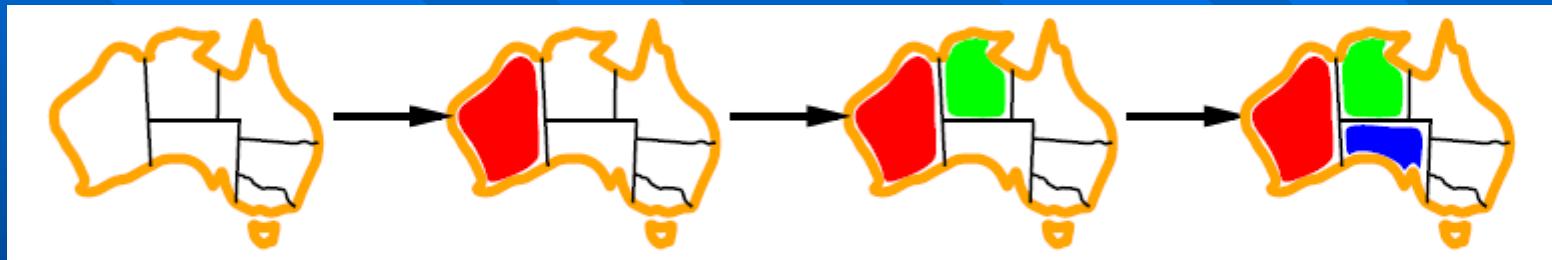


- Choix de la variable la plus contraignante (pour départager les égaux)
  - Plus grand nombre de contraintes avec les variables restantes

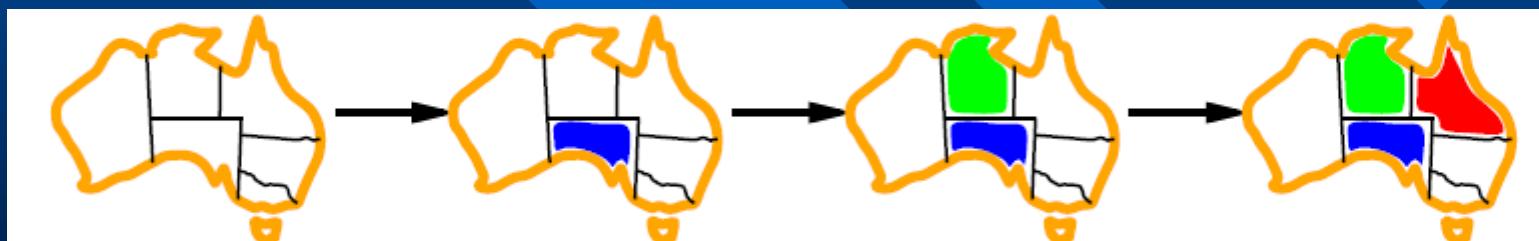


# Les idées d'heuristiques

- Choix de la variable la plus contrainte
  - Celle qui a le plus petit nombre de valeurs restantes (si 0 alors échec immédiat)



- Choix de la variable la plus contraignante (pour départager les égaux)
  - Plus grand nombre de contraintes avec les variables restantes



- Choix de la valeur la moins contraignante
  - Plus grand nombre de valeurs possibles dans les domaines des variables restantes



# Propriétés de l'espace de recherche

- Les choix faits dans les assignations précédentes peuvent avoir produit une assignation partielle
  - localement consistante
  - mais ne permettant pas d'obtenir une solution
- Dans ce cas l'algorithme de backtracking testera tous les choix de complétions avant de s'en rendre compte et de revenir à une assignation précédente
  - On peut en travaillant sur les domaines améliorer le comportement de la recherche
  - Les « look-ahead » algorithmes

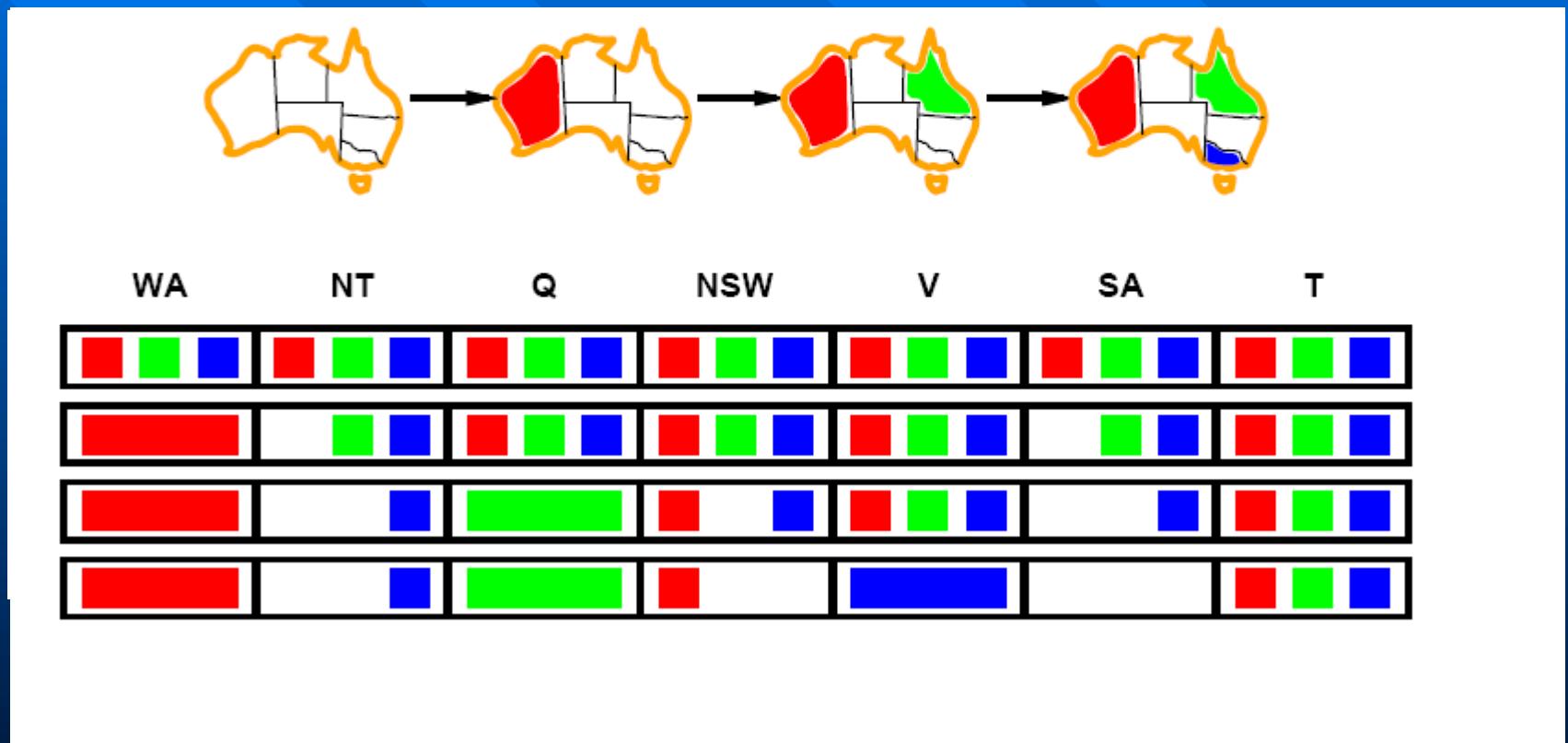


# Le forward checking

- L'idée consiste à chaque assignation de variable  $x$  à restreindre les domaines des variables non encore assignées
  - On supprime dans les domaines des variables  $y$  non assignées et reliées à  $x$  par une contrainte  $c$  les valeurs qui violent  $c$
  - Si un domaine devient vide alors « backtracking » immédiat et restauration des domaines
  - Les domaines des variables assignées sont réduits à la valeur de leur assignation

*Remarque : il n'est plus utile de vérifier la consistance d'une assignation dans le backtrack*

# Le forward checking sur l'exemple



# Forward Checking

Fonction FC(Assignation a, Problem p) : Booléen

Début

si |a| = |X| alors  
        afficher a;  
        retourner true;  
    finsi;

    x  $\leftarrow$  ChoixVariableNonAssignée(X, a);

    D<sub>old</sub>  $\leftarrow$  D ;

pour tout v  $\in$  Tri(D(x)) faire

si Propage({(x,v)}, p) alors  
            si FC(a  $\cup$  {(x,v)}, p) alors retourner true; finsi;

finsi;

        D  $\leftarrow$  D<sub>old</sub> ;

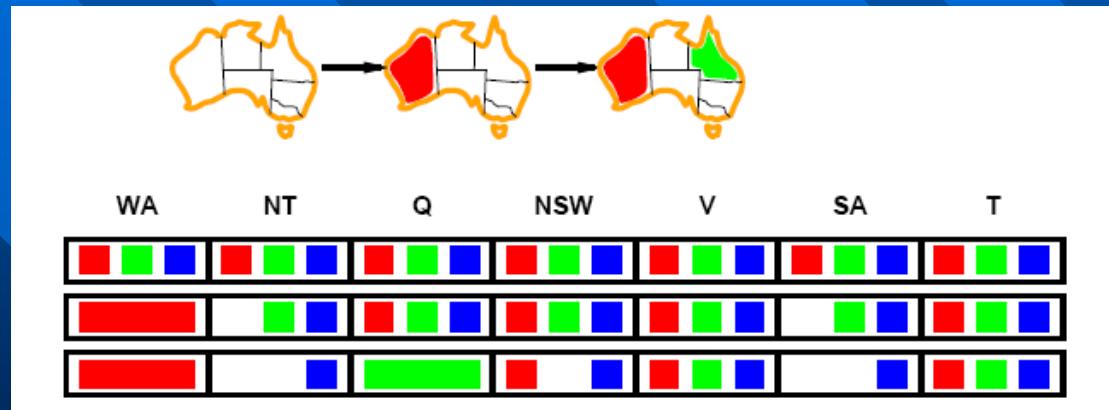
finpour;

retourner false;

Fin

# La propagation de contraintes

- Le forward checking propage l'information des variables assignées aux non-assignées
  - Mais il ne détecte pas tous les cas où la solution partielle ne pourra pas être étendue en une solution complète
  - Sur l' exemple :
    - » NT et SA ne pourront pas être bleu en même temps



- La propagation de contraintes consiste à propager les informations de suppression de valeurs possibles de manière itérative jusqu'à ce que toutes les variables soient localement consistantes

# La résolution de problème c'est :

- Modéliser :
  - Etat initial
  - Etats buts
  - Actions de changement d'état
  - Coûts
    - » Choix d'un type et d'un langage de représentation
- Analyser les données
  - les états, les actions, l'espace des états (taille, structure, heuristique...)
    - » Proposer un algorithme d'exploration des états ou réutiliser un algorithme connu (Dijkstra, BT, FC, MAC, A\*, αβ-pruning...)
- Exhiber les caractéristiques de l' algorithme
  - Complétude, optimalité, complexités
    - » Rédaction de preuves basées sur l'identification d'invariants