Recherche dans les graphes d'états

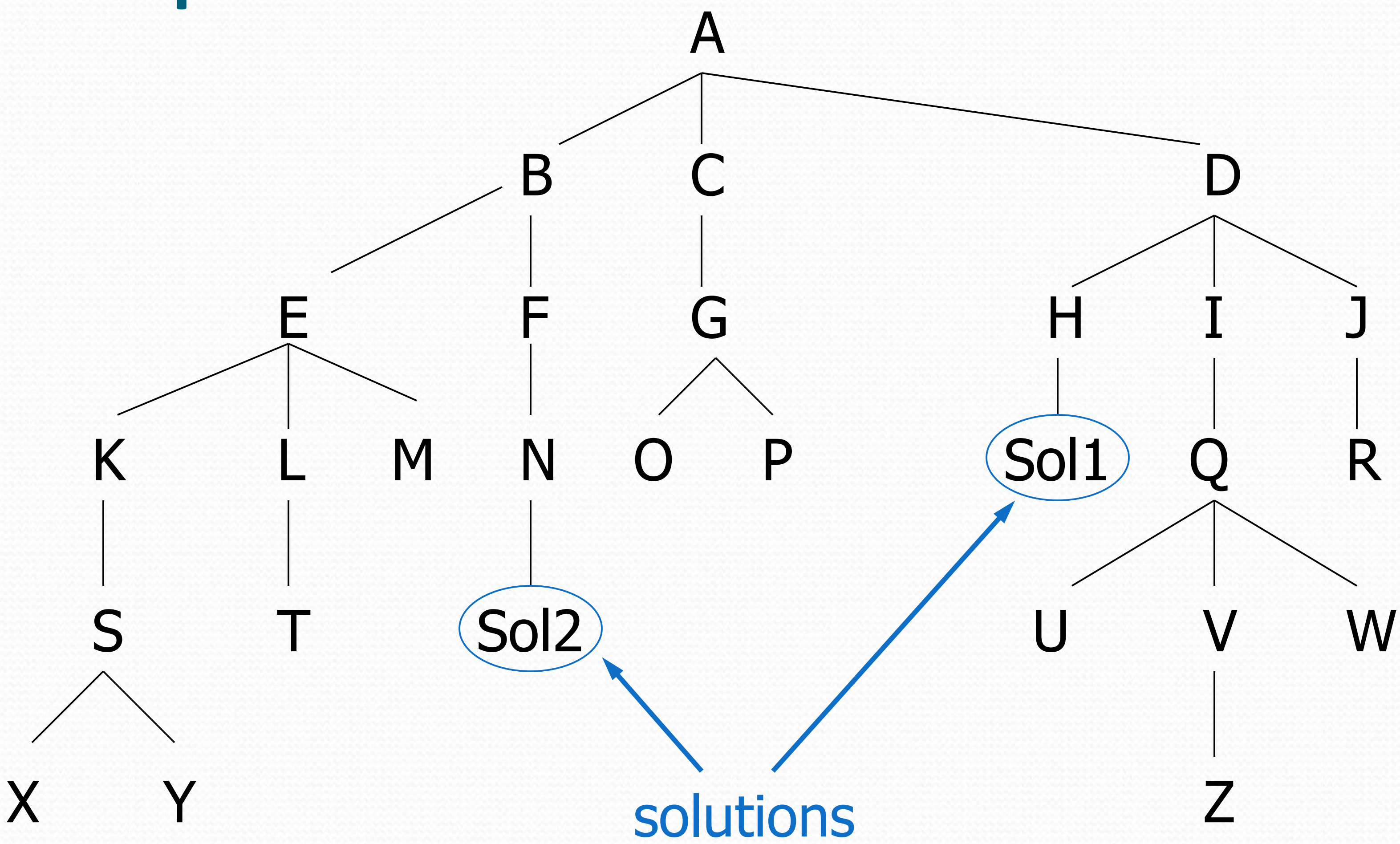
Plan

- Algorithmes de recherche dans un graphe
  - non-informés : profondeur, largeur, coût uniforme
  - informés : A*

Évaluation des algos de recherche dans les graphes d'états

- Complétude : trouver une solution si elle existe
- Admissibilité : trouver la meilleure solution
- Complexité en temps : temps pour trouver la solution
- Complexité en espace : espace mémoire pour faire la recherche
  - b : facteur de branchement
  - p : profondeur du (meilleur) nœud solution
  - p_{max} : profondeur maximum de l'espace de recherche

Le problème illustratif



Algorithmes général de recherche avec graphe

- Construit un sous-graphe du graphe d'état implicite qui représente les tentatives de résolution
- Exhibe un chemin solution dans ce sous-graphe
- Utilise différentes stratégies d'exploration reposant sur l'ordre des nœuds explorés

Notations

- OUVERT : ens. des nœuds en attente de développement
- FERMÉ : ens. des nœuds déjà développés
- b : facteur de branchement
- p : longueur du chemin solution le plus court
- p_{max} : profondeur maximum de l'espace de recherche
- e_o : état initial ; $et_1..et_j$: les états terminaux
- k(n,s) : coût pour passer de l'état n à son successeur s
- père(n) : pointeur vers le prédécesseur de n appartenant au chemin solution

Trame d'un algorithme avec graphe

OUVERT ← {e_o}; FERMÉ ← {}; succès ← faux;

<u>tant que</u> OUVERT ≠ {} et succès = faux

Étape 1 : **choisir** n dans OUVERT

<u>si</u> n est terminal <u>alors</u> succès ← vrai <u>sinon</u>

Étape 2 : développer n

Supprimer n de OUVERT et l'ajouter à FERMÉ

<u>pour chaque</u> s successeur <u>de</u> n

<u>si</u> s ni dans OUVERT ni dans FERMÉ (on ne visite pas deux fois le même nœud)

<u>alors</u> ajouter s à OUVERT ; père(s) ← n <u>fsi</u>

<u>Échec</u> si OUVERT = {}

<u>fsi</u>

<u>ftantque</u>

Recherche dans graphe d'états

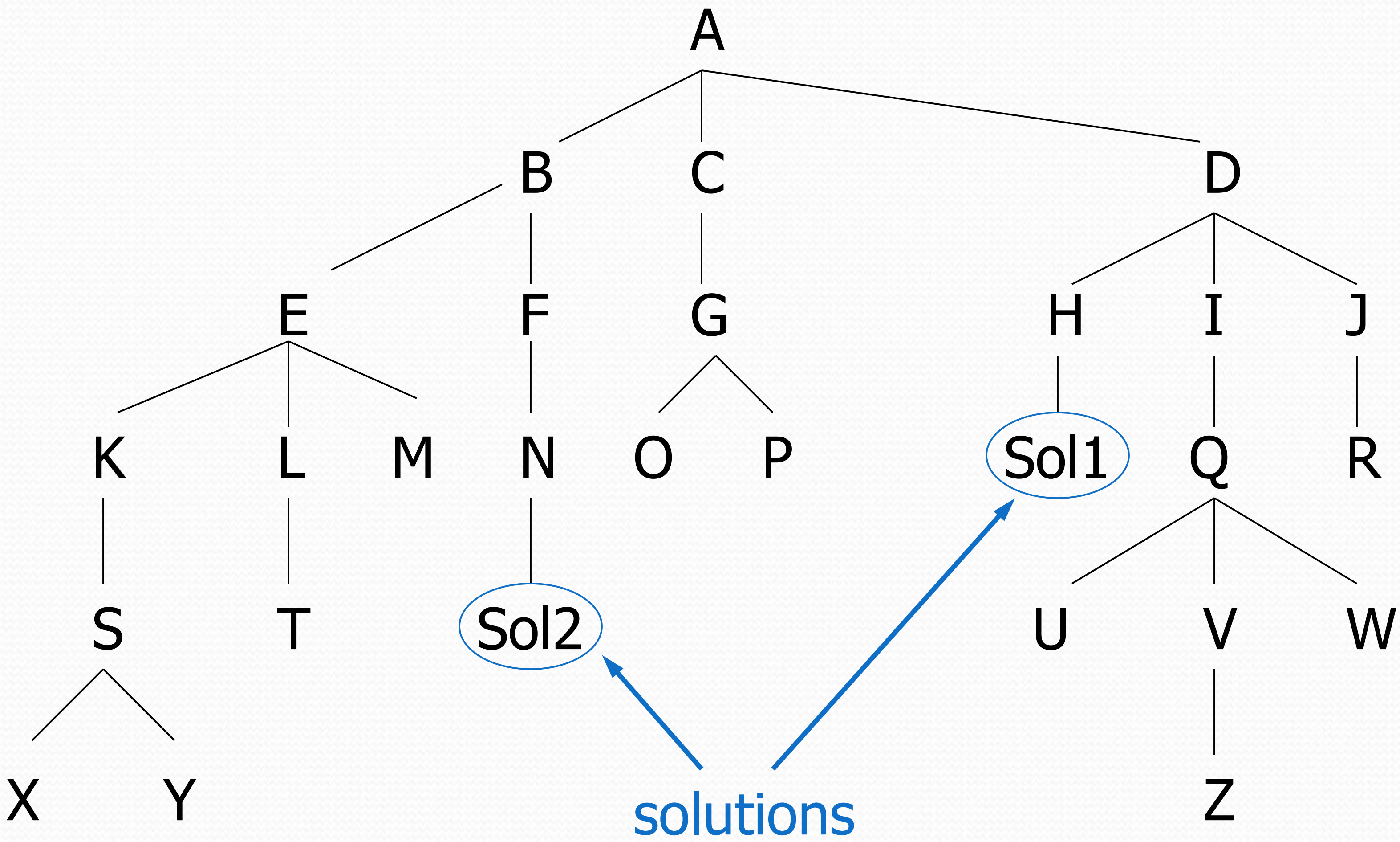
Plan

- Algorithmes avec graphe non-informés
  - Largeur d'abord
  - Profondeur d'abord
  - **Profondeur itérative**
  - Coût uniforme
- Algorithmes avec graphe informés

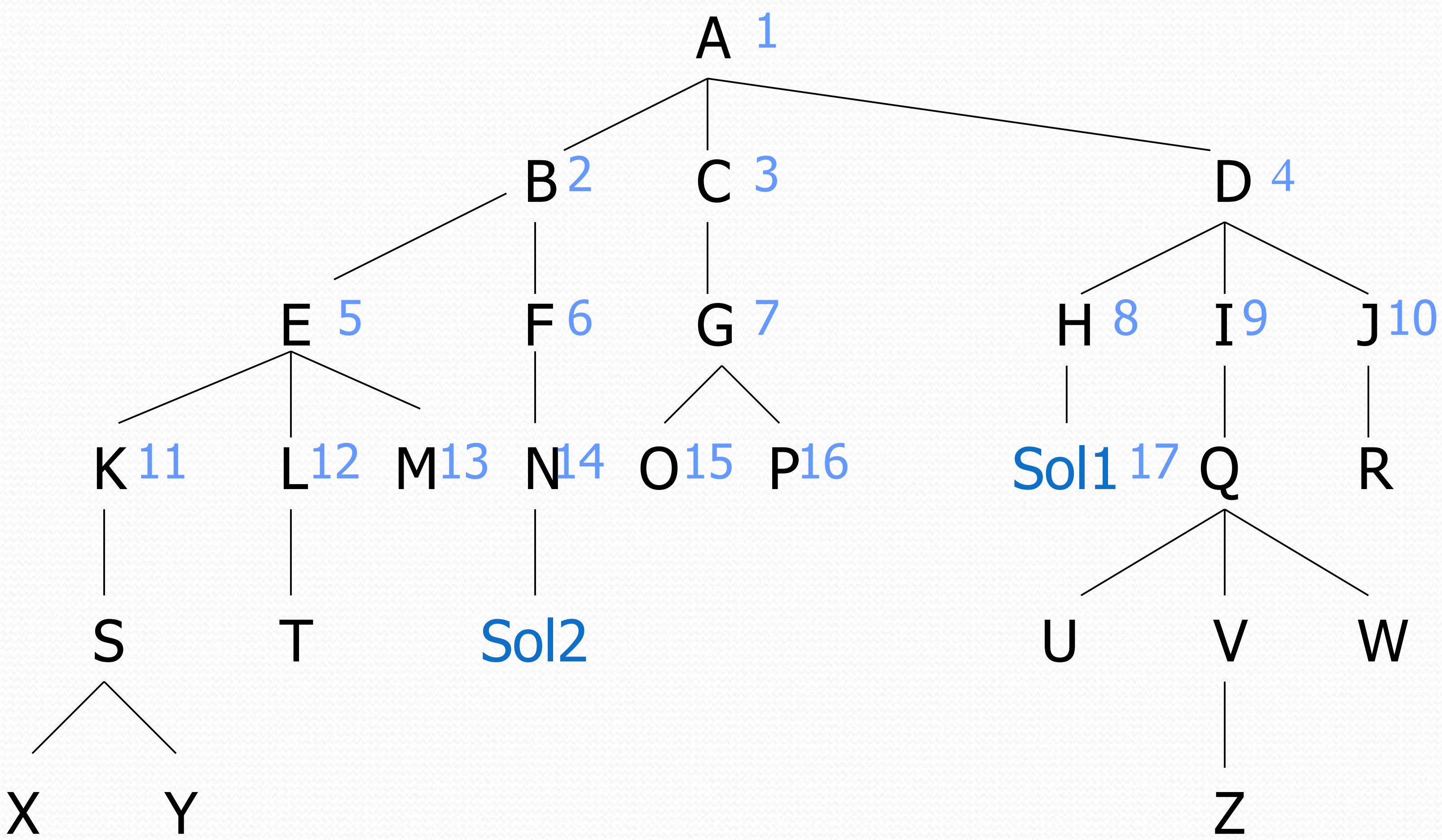
Largeur d'abord (*breadth-first*)

- Principe : développer d'abord le nœud le moins profond dans l'étape 1
  - OUVERT est une file (FIFO) (insertion des successeurs à la fin de la liste d'attente)
- Complet (si b est fini) et optimal (si coût constant, non sinon)
- Complexité temps (au pire) : $(b^{pmax+1}-1)/(b-1) \sim O(b^{pmax})$
- Complexité espace $\sim O(b^{pmax})$ (garder chaque nœud)

Exo



Largeur d'abord (*breadth-first*)

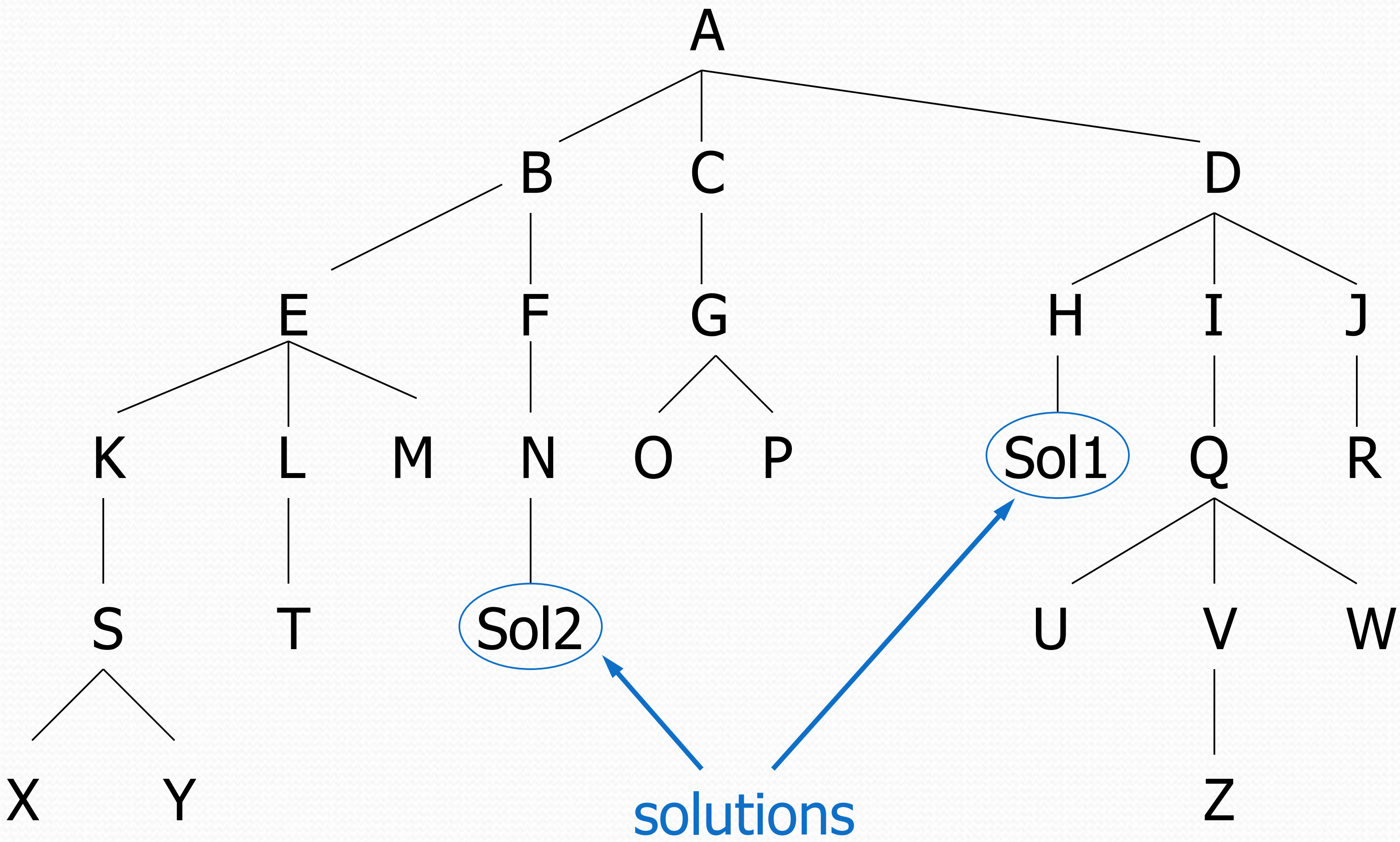


Profondeur d’abord (*depth-first*)

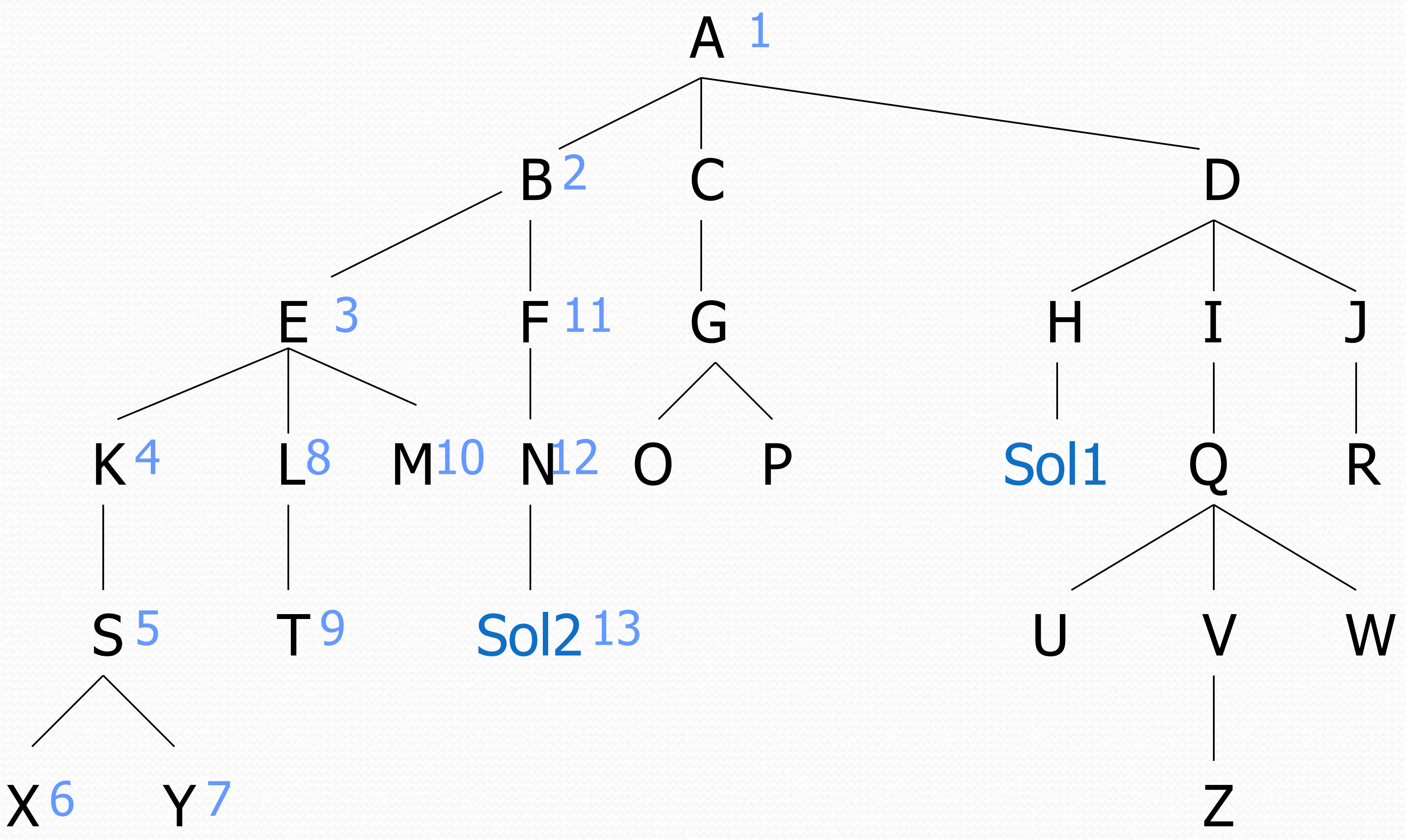
- Principe : développer d’abord le nœud le plus profond (étape 1)
  - OUVERT est une pile (LIFO) (insertion des successeurs en tête de la liste d’attente ; production dans le même ordre que *backtrack*)
- Complet (si graphe acyclique) mais pas optimal (on trouve un chemin)
  - Risque de s’égarer dans un chemin infini

    $\Rightarrow$ on peut borner la profondeur à d
- Complexité temps (pire) ~ $O(b^{pmax})$
- Complexité espace ~ O(b*pmax)

Exo



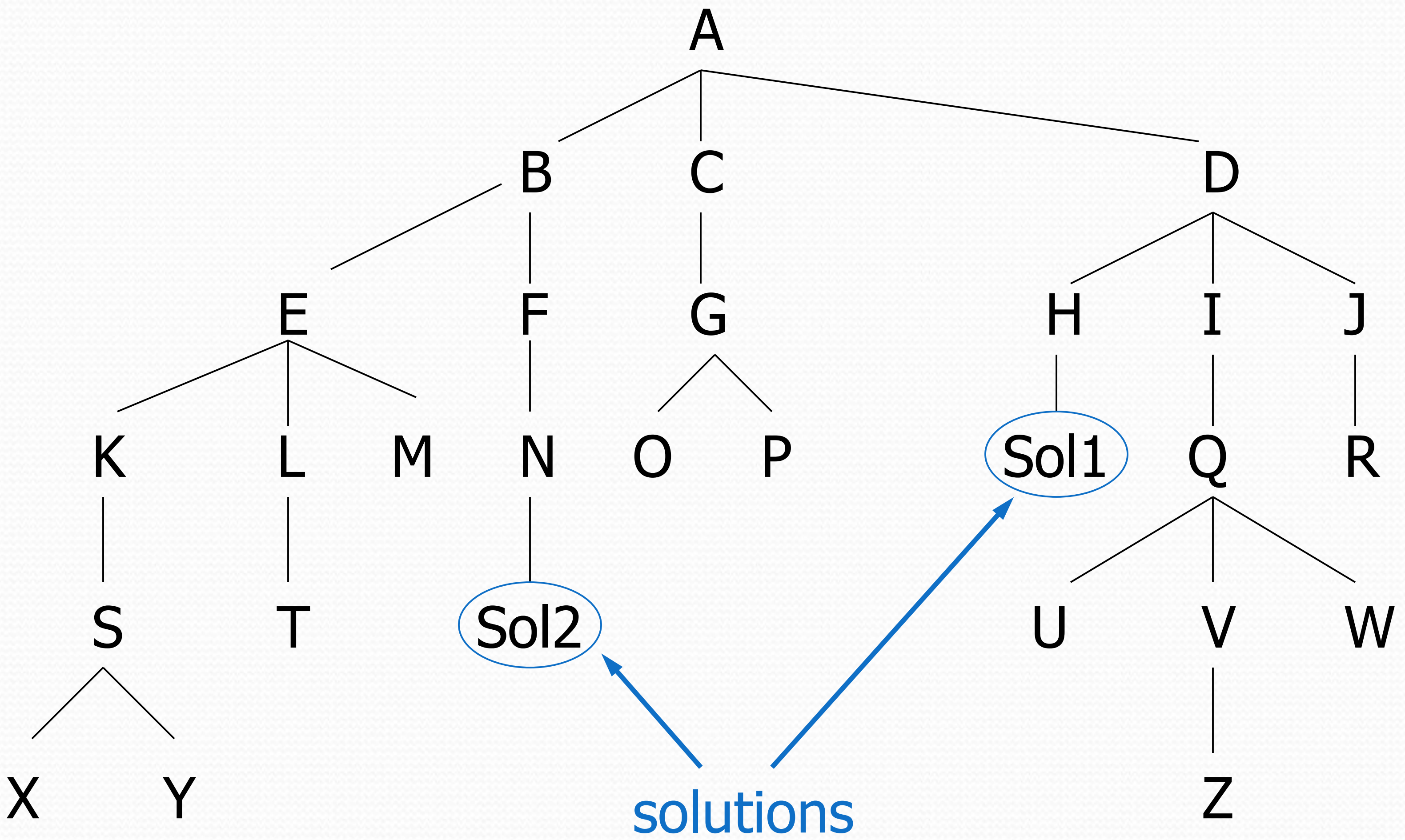
Profondeur d'abord (*depth-first*)



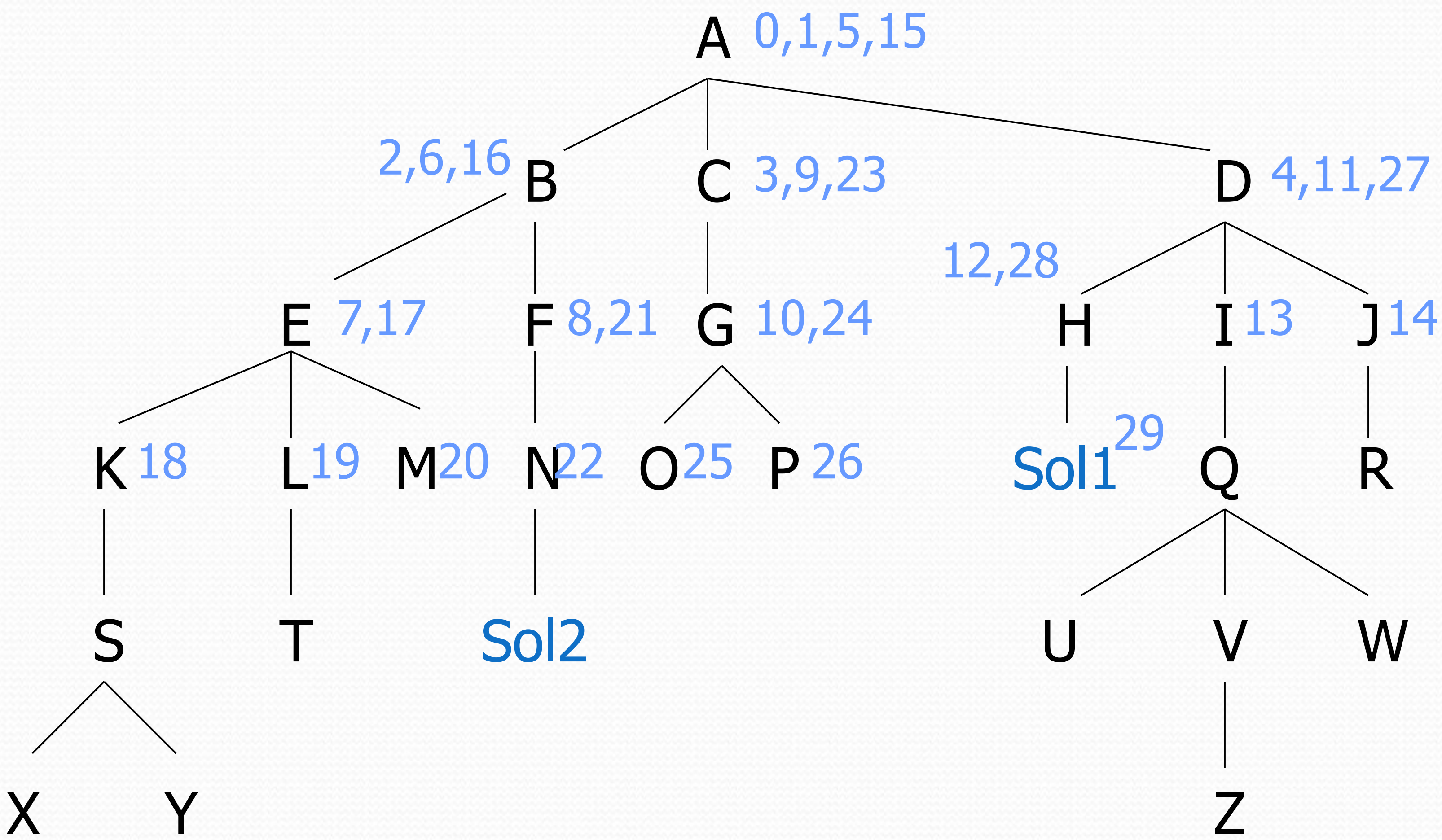
Profondeur itérative (*iterative deepening* IDA)

- Combine largeur et profondeur d'abord
- On applique profondeur d'abord pour d allant de 0 à p_{max} (p_{max} change à chaque tour d'itération)
- Complet et optimal si coût=1 par étape (comme largeur d'abord)
- Complexité temps : $\sum$ (pmax-i+1)b^i ~ O(b^{pmax}) (profondeur d'abord)
- Complexité espace ~ O(b*pmax) (profondeur d'abord)
- Bcp de redondance mais c'est moins cher que le développement du dernier niveau

Exo



Profondeur itérative (*iterative deepening* IDA)



Recherche dans graphe d'états

<u>Plan</u>

- Algorithmes avec retour arrière
- Algorithmes avec graphe non-informés
  - Largeur d'abord
  - Profondeur d'abord
  - Profondeur itérative
  - Coût uniforme
- Algorithmes avec graphe informés

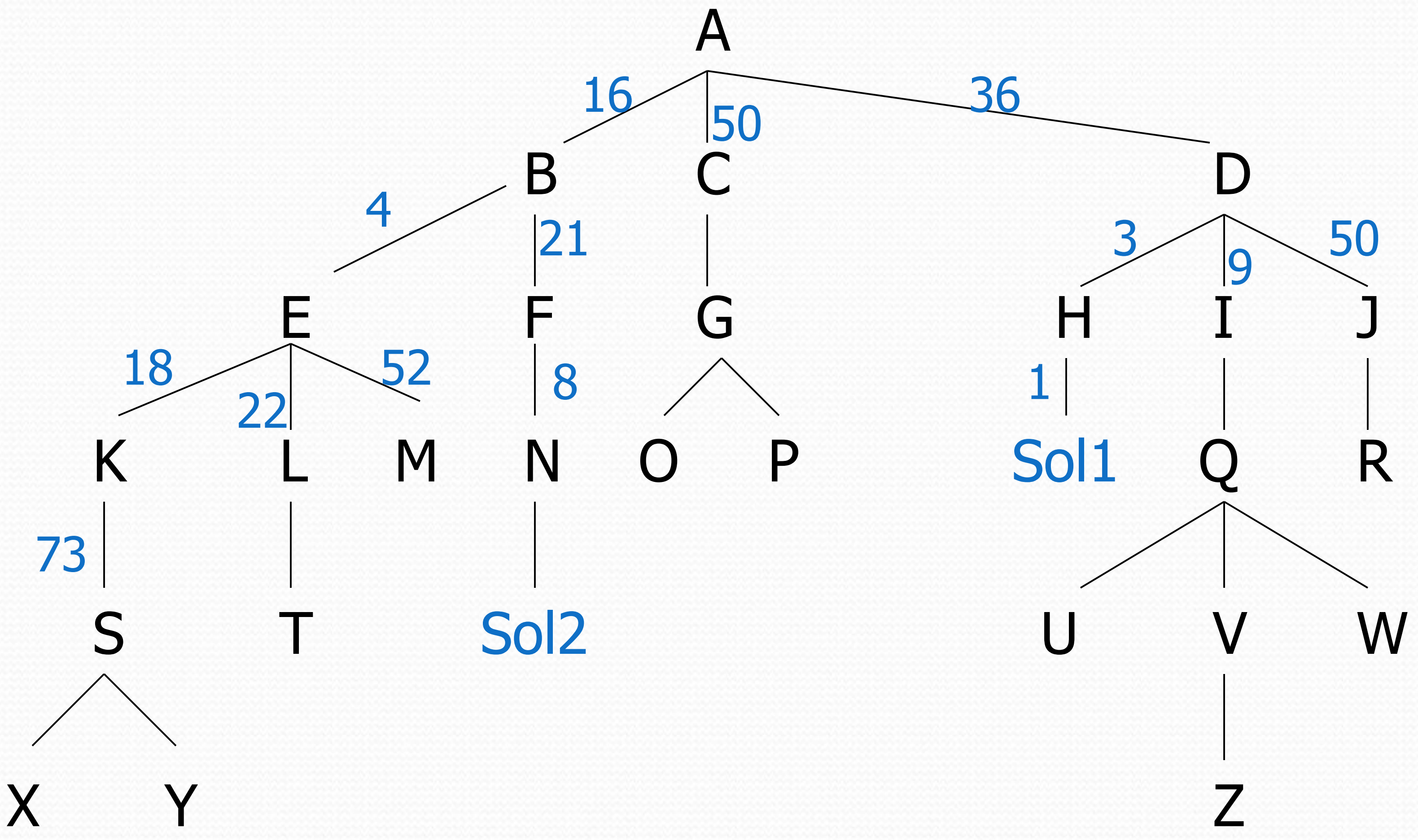
Coût uniforme *(uniform cost) ou coût minimal*

- Principe : développe le nœud de coût le plus faible (donné par une fonction g)

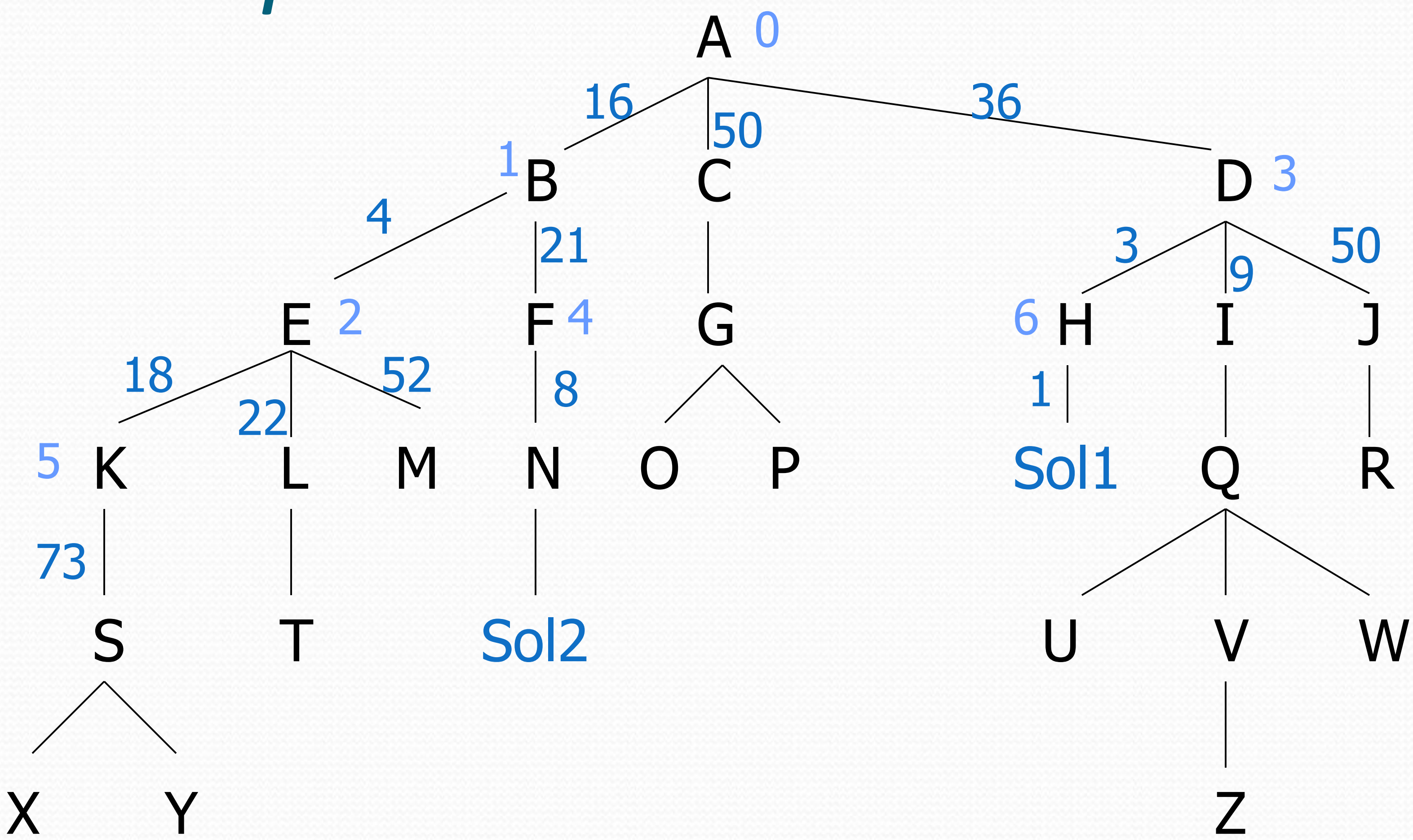
  => variante de largeur d'abord quand les arcs sont valués (insertion des successeurs dans l'ordre de coût croissant)
- Il ne faut pas de coûts négatifs

  i.e. si s est un successeur de n, $g(s) \geq g(n)$
- Complet et optimal
- Complexité temps ~ $O(b^{pmax})$
- Complexité espace ~ $O(b^{pmax})$

Coût uniforme : *exo*



Coût uniforme *(uniform cost)* *exemple*



Récapitulatif (graphes non informés)

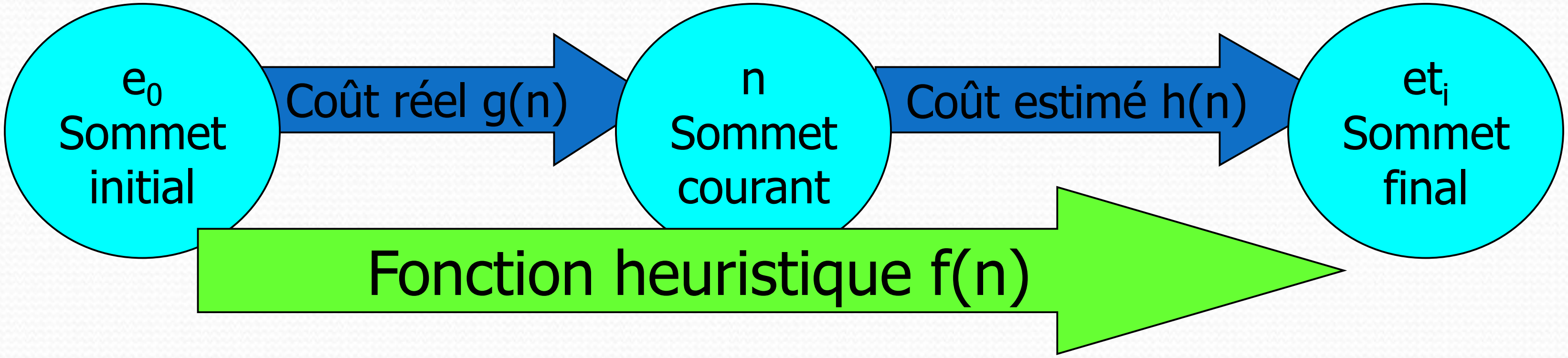
	Largeur d'abord	Profondeur d'abord	Profondeur itérative	Coût uniforme
Temps	$O(b^p)$	$O(b^{p_{max}})$	$O(b^p)$	$O(b^p)$
Espace	$O(b^p)$	$O(b*p_{max})$	$O(b*p)$	$O(b^p)$
Optimal ?	oui(cond)	non	oui(cond)	oui
Complet ?	oui	oui(cond)	oui	oui

Information heuristique

- Le but est d'utiliser de l'information liée au pb pour restreindre le nombre de nœuds à développer avant d'atteindre la solution
- Le choix de l'étape 1 se fait grâce à une fonction d'évaluation f ; on parle alors de meilleur d'abord *(best-first)*
- f combine souvent deux fonctions *g* et *h*
  - g(n) = coût nécessaire de e_o à n (généralement celui du meilleur chemin de e_o à n jusqu'à présent)
  - h(n) = estimation du coût de n à un et_i (plus difficile à choisir)

Information heuristique

- On utilise en même temps
  - le coût g pour arriver au sommet traité
  - la distance h estimée entre le sommet et l'objectif
- Favorise les sommets qui sont sur le plus court chemin allant de e_0 à un et_i

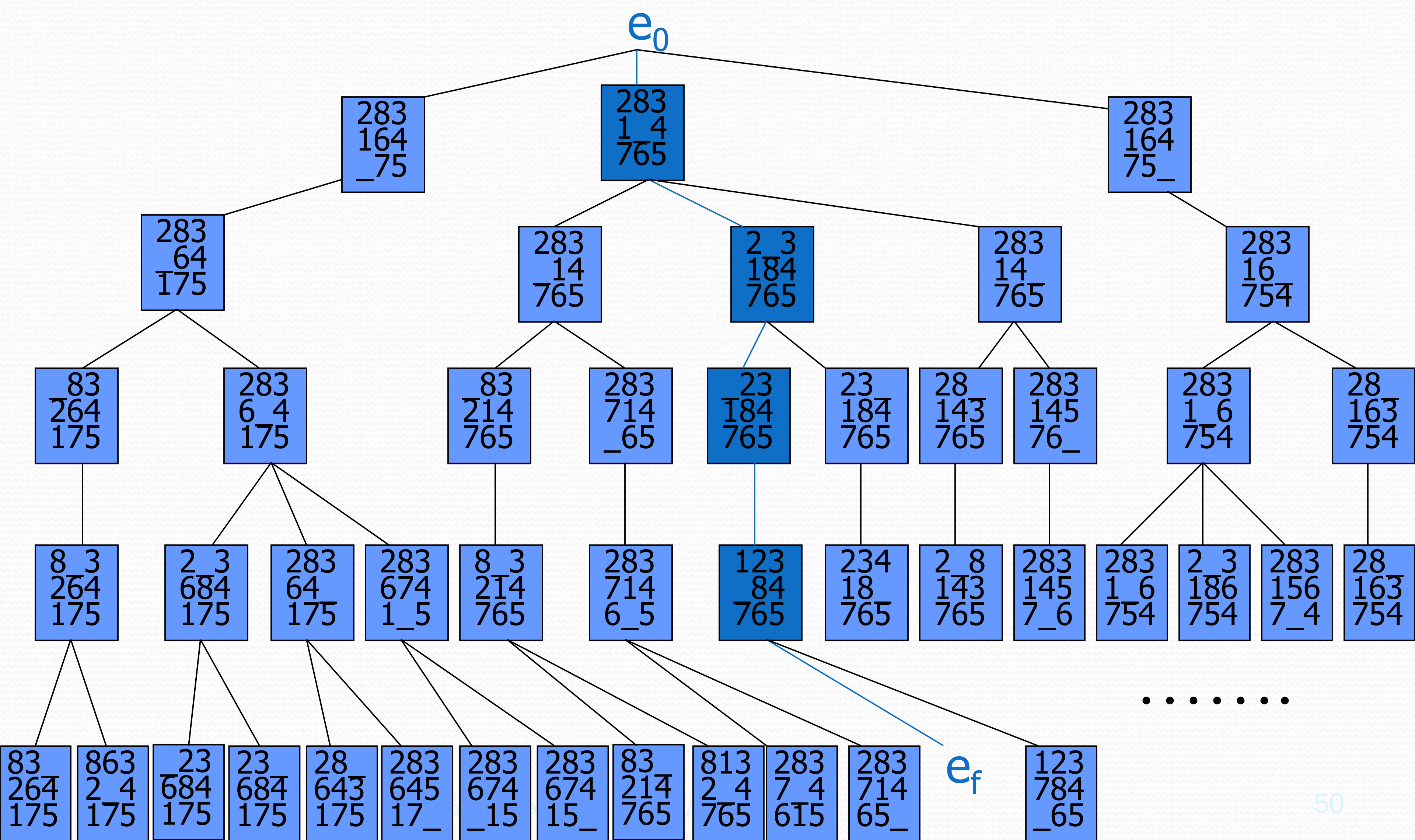


Information heuristique - Exercice du taquin

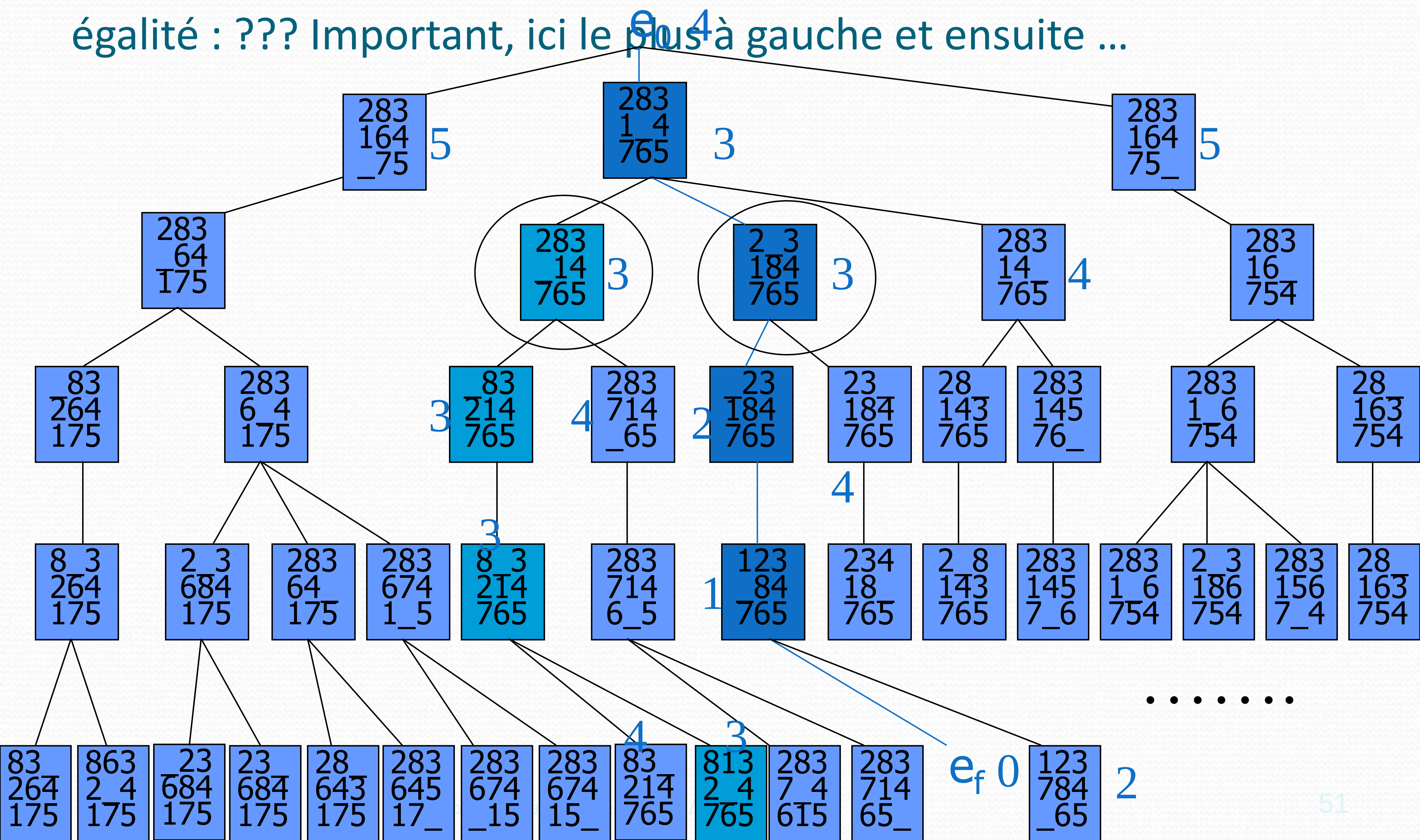
- $f_1 = 0 + h_1$ = nombre de pièces mal placées (sauf le blanc)
- f_2 = profondeur du nœud courant + h1
- Résoudre le problème (développer le graphe jusqu'à trouver une solution) avec f_1 puis f_2 et e_0 =
- Quelle est l'heuristique la plus efficace ?

2	8	3
1	6	4
7	_	5

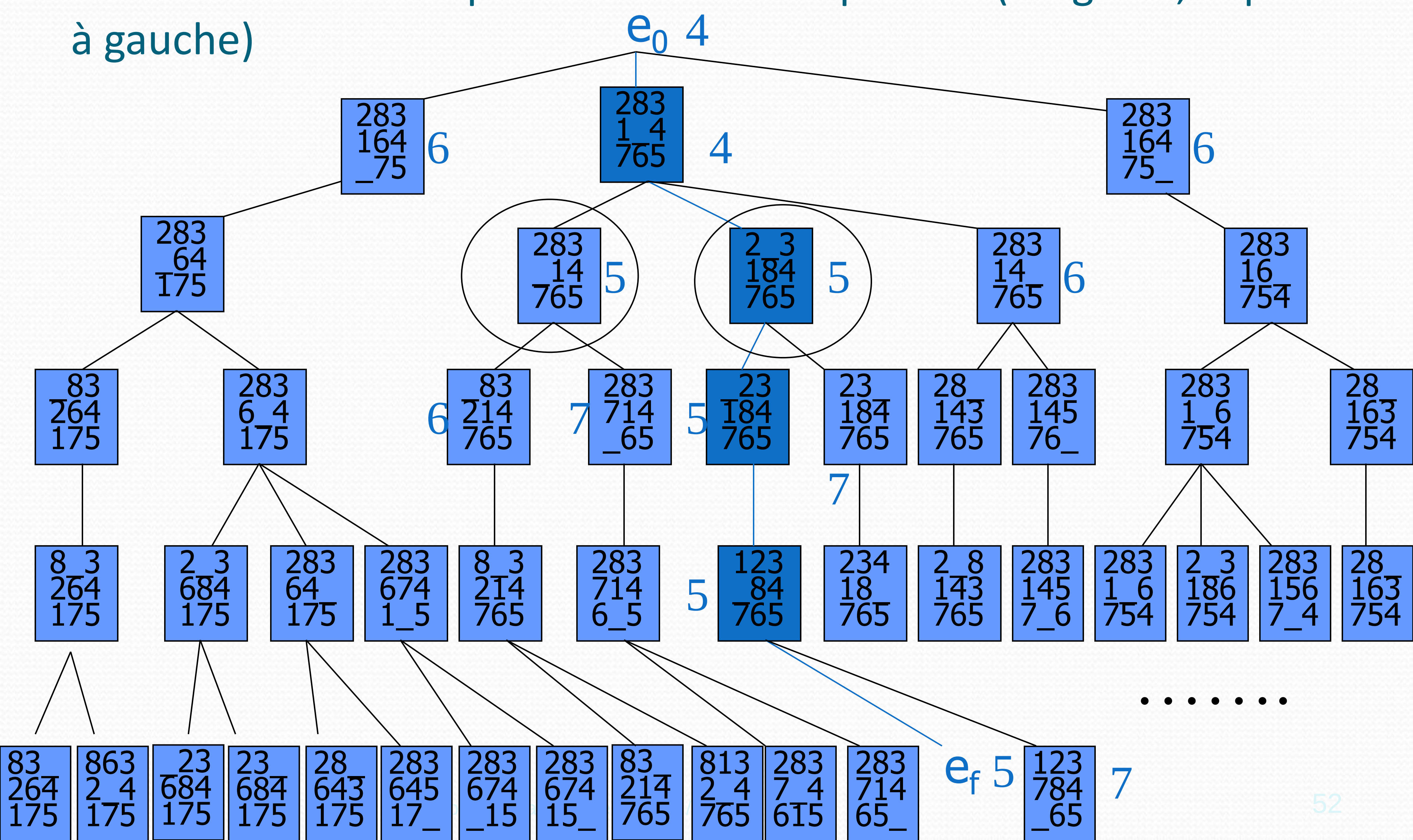
Information heuristique - exercice du taquin - graphe



Information heuristique - exercice du taquin – f1 (si égalité : ??? Important, ici le plus à gauche et ensuite …



Information heuristique - exercice du taquin - f2 (si égalité, le plus à gauche)



Coût uniforme (*uniform-cost*) et gradient

2 cas particuliers de « meilleur d'abord », parmi une famille

Uniform-cost (coût minimal)

- Choisir le nœud n de OUVERT de coût minimum trouvé jusqu'à présent entre e_0 et n *i.e.* h=0
- Largeur d'abord = *uniform-cost* avec des opérateurs ayant des coût égaux (ou nuls)

Gradient

- Choisir le nœud n de OUVERT de coût estimé minimum entre n et un et_i *i.e.* g=0

Recherche dans graphe d'états

Plan

- Algorithmes avec retour arrière
- Algorithmes avec graphe non-informés
- Algorithmes avec graphe informés
  - Information heuristique
  - Algorithme A*

Algorithme A* : meilleur d'abord avec conditions sur la fonction d'évaluation

- On part d'un état initial e_o
- On souhaite atteindre un des états $et_1, \ldots, et_n$
- $g^*(X)$: longueur du chemin le plus court entre e_o et X
  - **estimation $g(X)$: meilleure solution trouvée pour l'instant $g(X) \geq g^*(X)$**
- $h^*(X)$: longueur du chemin le plus court entre X et un état final et_i
  - **estimation $h(X)$ (donnée) avec $h(X) \leq h^*(X)$ (h minorante)**
- $f^*(X)$: longueur du chemin (le plus court, passant par X) entre e_o et un état final et_i : $f^*(X) = g^*(X) + h^*(X)$
  - **estimation $f(X) = g(X) + h(X)$**

Algorithme A* : définitions (2)

- h est parfaite ssi pour tout n : $h(n) = h^*(n)$
- h est minorante (ne surestime pas le coût de n) ssi pour tout n : $0 \leq h(n) \leq h^*(n)$
- h est monotone ssi pour tout n et tout s successeur de n : $h(n) - h(s) \leq k(n,s)$

Algorithme A*

OUVERT ← $\{e_o\}$, FERMÉ ← {}, succès ← faux

<u>tant que</u> OUVERT ≠ ∅ <u>et</u> ¬succès <u>faire</u>

X = $\text{argmin}_{X \in OUVERT}(f(X))$ (X tel que f(X) est minimum)

<u>si</u> X est solution <u>alors</u> succès ← vrai

<u>sinon</u>

OUVERT = OUVERT – {X} ; FERMÉ = FERMÉ U {X}

<u>pour tout</u> op ∈ liste_op(X) <u>faire</u>

X' = op(X)

<u>si</u> (X' ∉ OUVERT U FERMÉ) <u>ou</u> ((X' ∈ OUVERT et g(X') > g(X) + k(op)) <u>alors</u> (si X' est déjà dans ouvert, il faut vérifier si le nouveau chemin n'est pas plus court, et si c'est le cas, mettre à jour g(X') et père)

père(X') = X; g(X') = g(X)+ k(op); OUVERT = OUVERT U {X'}

sinsi X' ∈ FERMÉ et g(X') > g(X) + k(op) : g(X') = g(X)+ k(op); OUVERT = OUVERT U {X'} ; FERMÉ = FERMÉ– {X'} (pour éviter les mises à jour de tous les successeurs de X', on sort X' de FERME et on le met dans OUVERT)

fsi

<u>fsi</u>

<u>fpour</u>

<u>Fsi</u>

<u>ftantque</u>

Attention …

Algorithme A*(remarques)

A* : Recherche la solution optimale à l'aide de la fonction f ; algorithme de type meilleur d'abord

- g(X) (et donc f(X)) peut varier pendant la recherche => mettre à jour les valeurs des nœuds concernés
- h est en revanche statique et dépend du pb (*e.g.* h = nb de cases mal placées au taquin)
- Complexité de A* : variable, dépend de h (h permet d'élaguer l'arbre) et du calcul de h

A* utilisant Possibles

CH = <{no, n1, … nk}, g(CH), h(nk)>

CH_o = <{no, o, h(no)}

POSSIBLES ← {CH_o}, succès ← faux

<u>tant que</u> POSSIBLES ≠ ∅ <u>et</u> ¬succès <u>faire</u>

  CH = $\mathrm{argmin}_{CH \in POSSIBLES}$ (f(CH)) (CH tel que f(CH) est minimum)

  <u>si</u> CH est solution (nk état terminal) <u>alors</u> succès ← vrai

  <u>sinon</u>

    POSSIBLES = POSSIBLES – {CH} ;

    <u>pour tout</u> successeur $succ_i$(nk) de nk <u>faire</u>

      ajouter dans POSSIBLES le chemin <{no, … nk, $succ_i$(nk)}, g(CH) + c(nk, $succ_i$(nk), h($succ_i$(nk))>

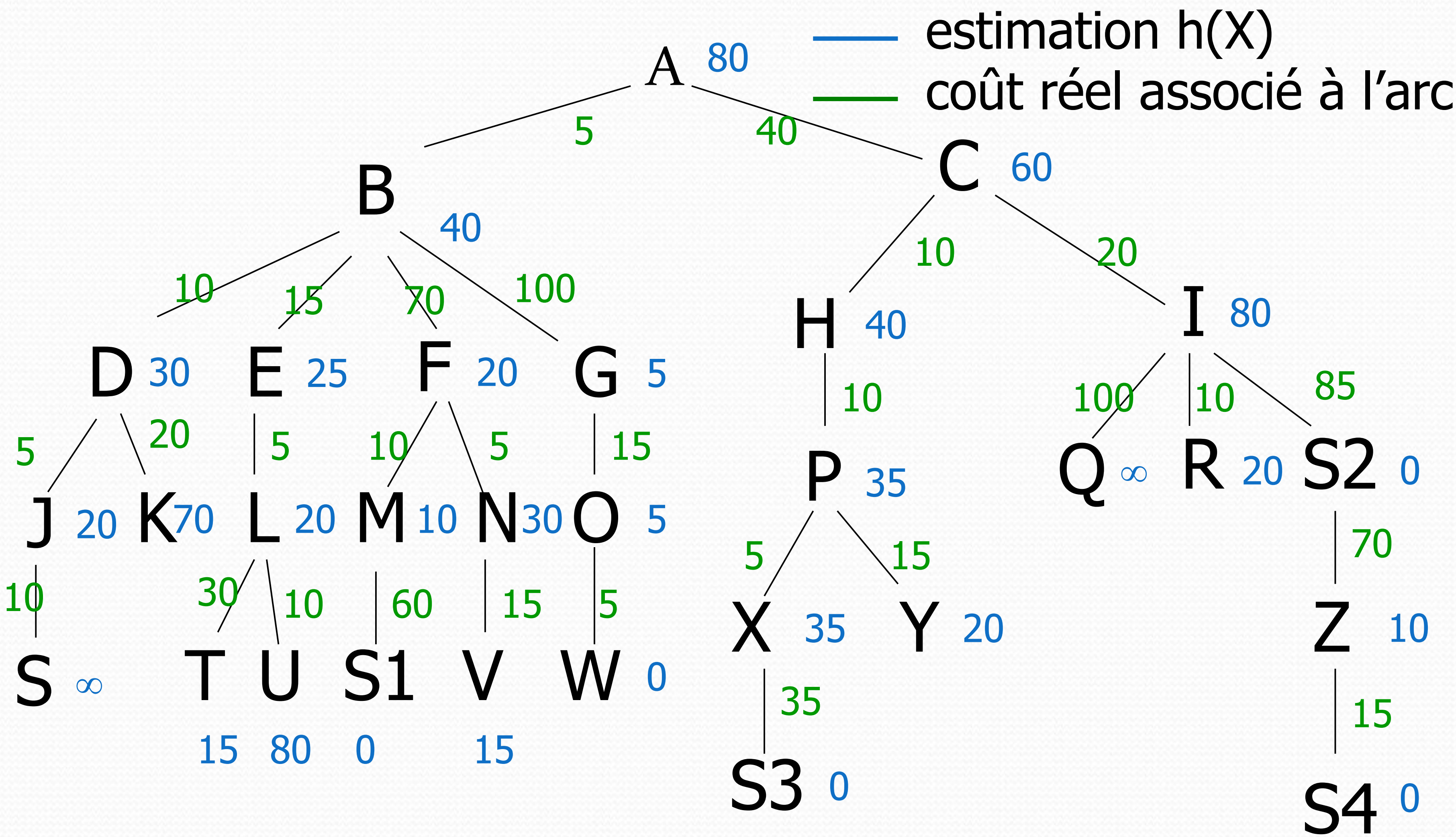
    <u>fpour</u>

  <u>Fsi</u>

<u>ftantque</u>

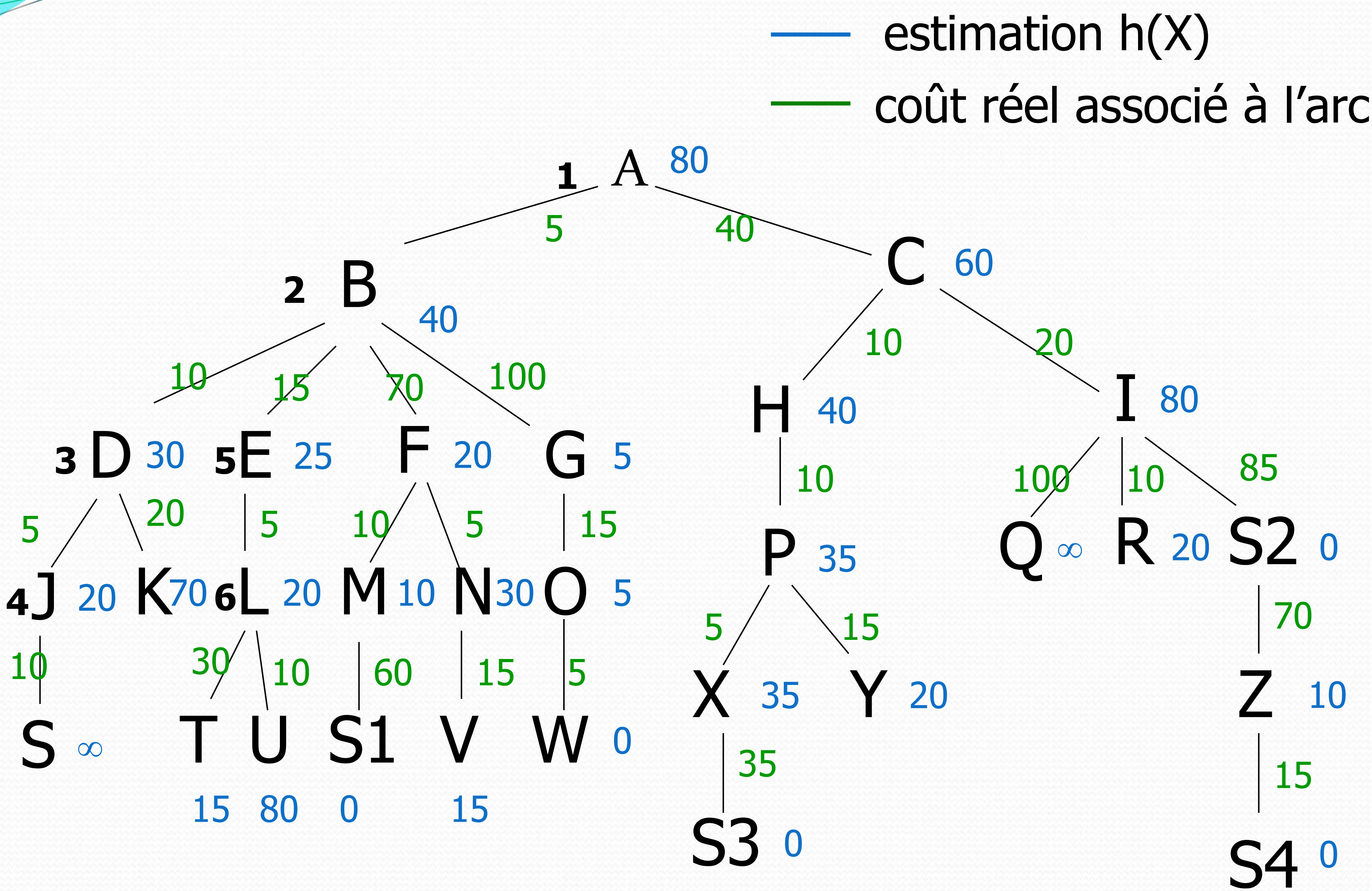
Redondances car on garde tous les chemins mais pas de mises à jour …
Ajouter un test de circuit
Ne garder que le meilleur chemin entre la racine et un noeud nk

Exemple

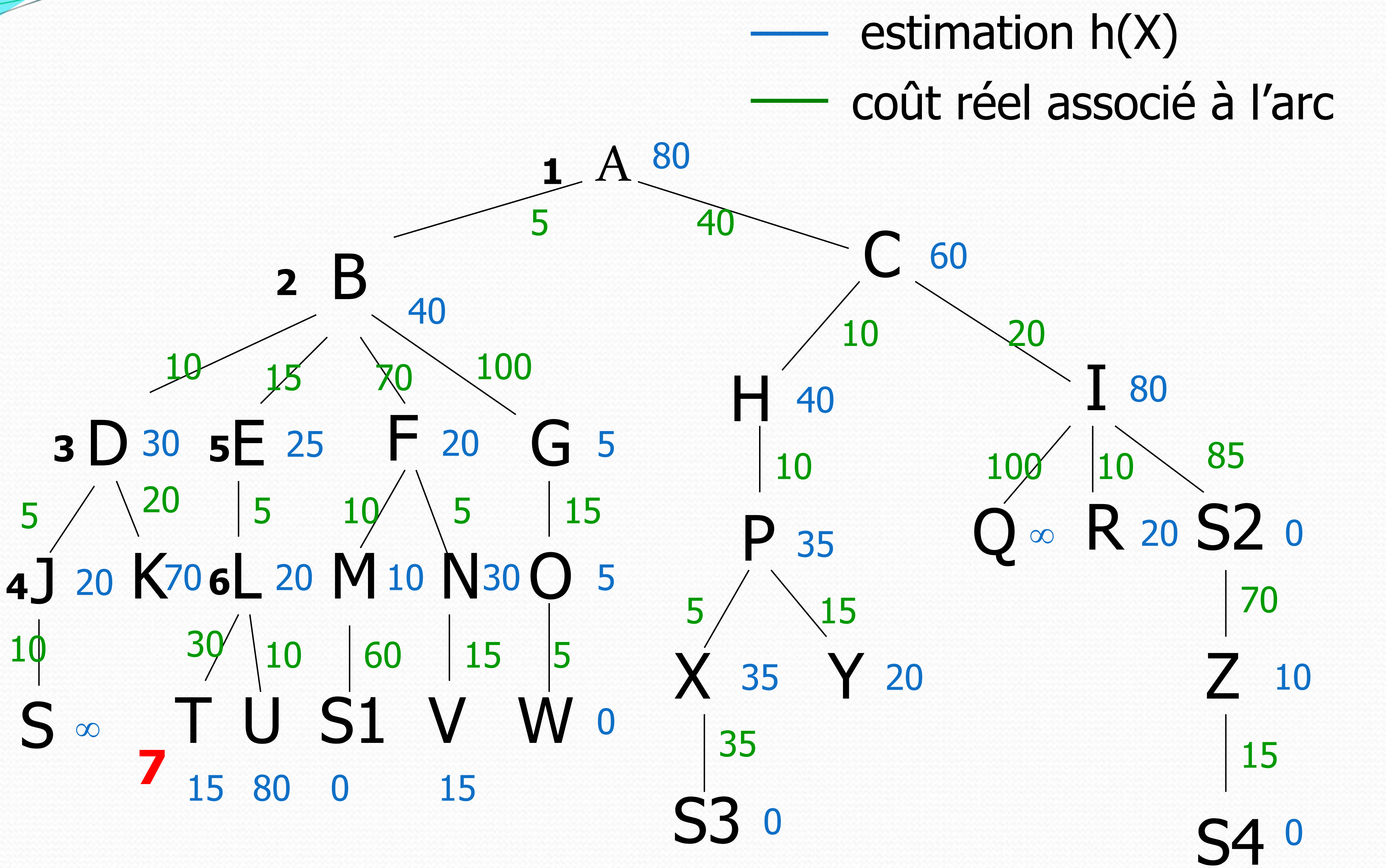


estimation h(X)
coût réel associé à l'arc
A 80
5
40
B 40
C 60
10
15
70
100
10
20
D 30
E 25
F 20
G 5
H 40
I 80
5
20
5
10
5
15
10
100
10
85
J 20
K 70
L 20
M 10
N 30
O 5
P 35
Q ∞
R 20
S2 0
10
30
10
60
15
5
5
15
70
S ∞
T
U
S1
V
W 0
X 35
Y 20
Z 10
15
80
0
15
35
15
S3 0
S4 0

Exemple … Quel est le prochain nœud à développer?



Exemple à finir…



Propriétés de A*

- Si :
  - h est minorante et coïncidante (h-but) = 0)
  - tous les arcs ont un coût supérieur à un $\varepsilon > 0$
  - chaque sommet a un nombre fini de voisins
- alors :
  - A* termine s'il y a un but accessible à une profondeur finie
  - A* est admissible (trouve un chemin optimal)

Autre propriété

- Si :
  - h est monotone ($h(n) \leq h(s) + k(n,s)$)
- alors :
  - aucun nœud n'est développé plusieurs fois
  - (pas de mise à jour nécessaire)

Intuitivement, h monotone requiert que la diminution de la distance estimée vis-à-vis du but ne diminue pas plus que le coût entre les deux nœuds (inégalité triangulaire).

Souvent assez difficile à prouver pour une fonction h donnée. Distance à vol d'oiseau : vérifiée.

On a : toute heuristique monotone et coïncidante est minorante

Algorithme A* - démonstration

Lemme 1

- À chaque étape précédant la terminaison de A*, il existe un sommet n non exploré (dans OUVERT) tel que
  - n est sur un chemin optimal vers un but
  - A* a trouvé un chemin optimal de e_o à n
  - $f(n) \leq f^*(e_o)$

On a $f(n)= g^*(n) + h(n)$ et $h(n) <= h^*(n)$ donc $f(n) <= f^*(n)$

Algorithme A* - démonstration

Quelques éléments à retenir

- Pour tout n
  - $g(n) \geq g^*(n)$ (= sur chemin optimal)
  - $(0 \leq)$ $h(n) \leq h^*(n)$ (h minorante)
- $f^*(e_o) = h^*(e_o) = f^*(et_i) = g^*(et_i) = f^*(n)$ (pour n sur chemin optimal)

Algorithme A* - démonstration

<u>Preuve du lemme 1 par récursivité</u>

- À la 1e étape, le sommet e_o est choisi (il appartient à OUVERT); il est sur le chemin optimal et $f(e_o) = h(e_o) \leq h^*(e_o) = f^*(e_o)$
- Supposons la propriété vraie à l'itération m et soit n le sommet non exploré trouvé à cette étape
- À l'étape m+1, si n n'est pas sélectionné, il possède toujours la même propriété ($f(n) \leq f^*(e_o)$) et est donc le sommet recherché
- Si n est sélectionné, un (au moins) de ses descendants s est situé sur un chemin optimal vers un but. A* a de plus trouvé un chemin optimal vers s car sinon, il existerait à l'étape précédente un meilleur chemin de e_o à n, ce qui contredirait notre hypothèse
- On a $f(s) = g(s) + h(s) \leq g^*(s) + h^*(s)$

  car $g(s) = g^*(s)$ et $h(s) \leq h^*(s)$, donc

  $f(s) \leq f^*(s) = f^*(e_o)$ car s est sur un chemin optimal

Algorithme A* - démonstration

Preuve de la terminaison de A*

- Si le graphe est fini, c'est évident
- Si le graphe est infini, supposons que A* ne termine pas
  - Dans ce cas, A* continue l'extension de la frontière à l'infini, c'est-à-dire à une profondeur non bornée, car on a supposé que chaque sommet a un nombre fini de voisins
  - Le coût de chaque arc étant supérieur à ε, la valeur de g(n) pour chaque sommet n de la frontière finira par dépasser $f^*(e_0)$, ce qui contredit le lemme 1

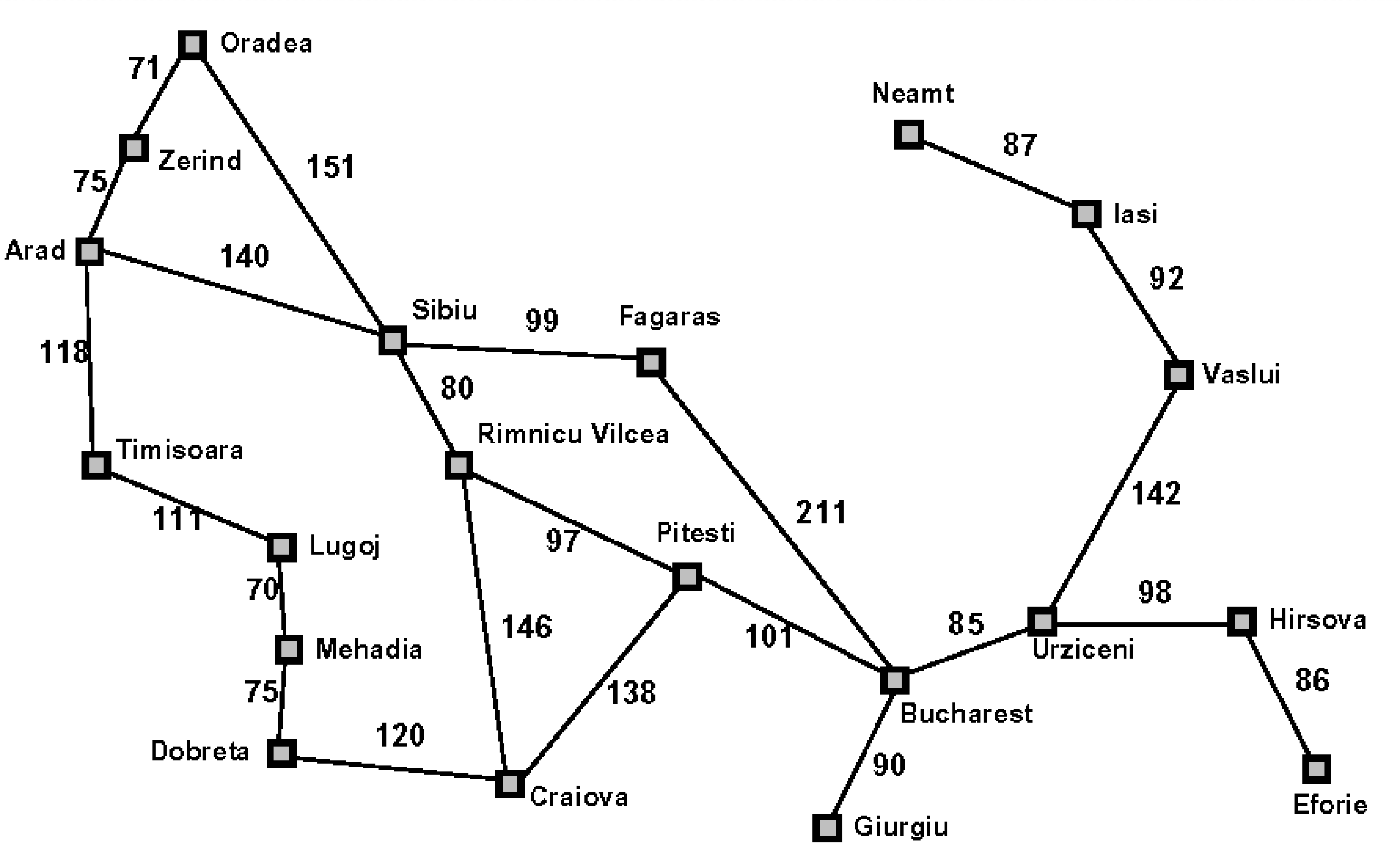
Algorithme A* - démonstration

<u>Preuve de l'admissibilité de A* par l'absurde</u>

- Supposons que A* termine sur l'état t' avec une solution non optimale, *i.e.* avec $f^*(t') > f^*(e_0)$ alors qu'il existe une solution optimale avec t tq $f^*(t) = f^*(e_0)$
- À la terminaison en t', on a $f(t') \geq f^*(t') > f^*(e_0)$ (car $g(t') \geq g^*(t')$)
- D'après le lemme 1, on sait qu'à l'étape précédente, il existait un état n dans OUVERT tel que $f(n) \leq f^*(e_0)$
- Donc A* n'a pas pu sélectionner t' (car $f(n) < f(t')$; tout nœud d'un chemin optimal sera étudié avant t')

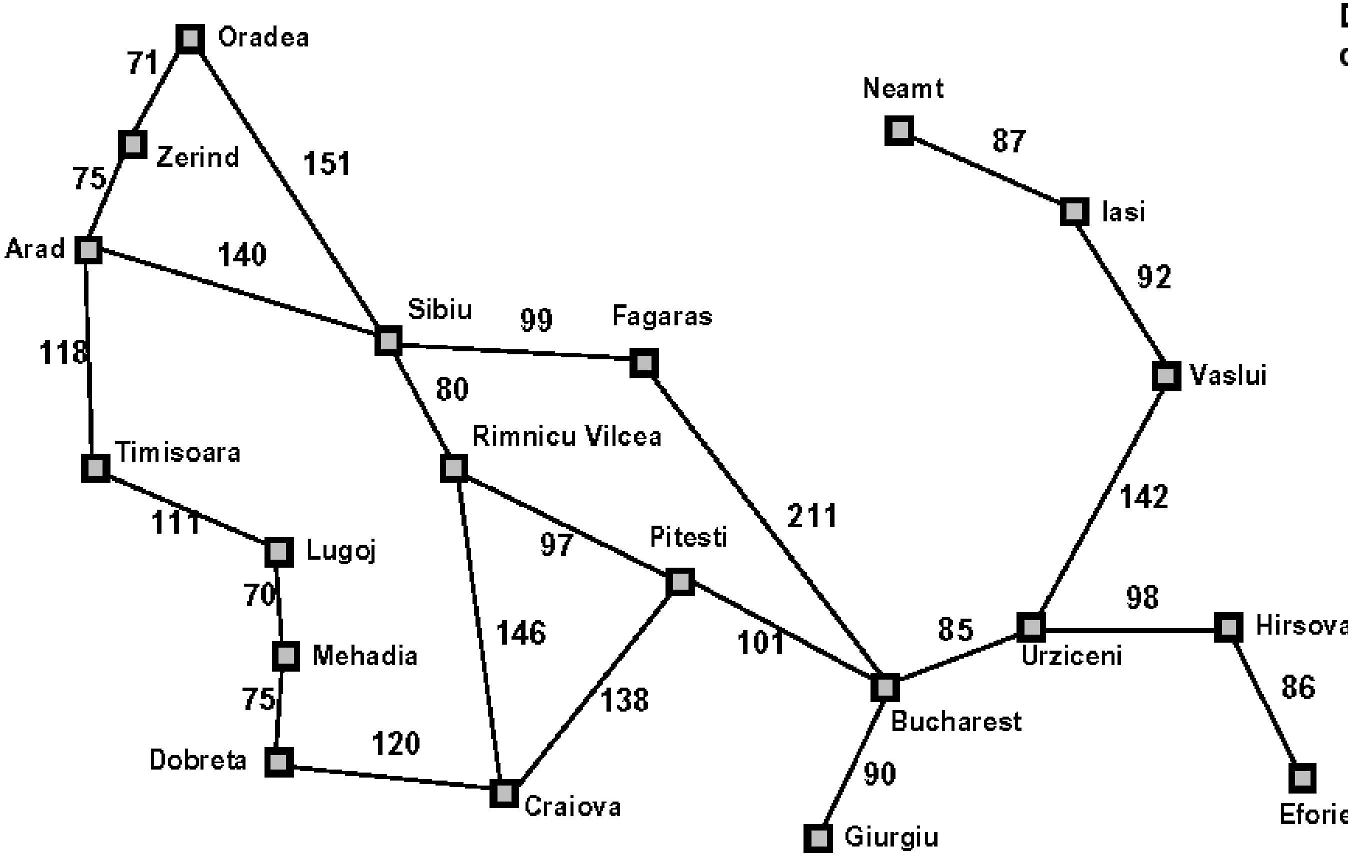
Algorithme A* - exercice

Voyage en Roumanie (d'après S. Russel et P. Norvig)



Algorithme A* - exercice

<u>Voyage en Roumanie d'Arad à Bucharest</u>



Distance "à vol d'oiseau" de Bucharest

Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Algorithme A* - exercice

<u>Voyage en Roumanie d'Arad à Bucharest</u>

- On souhaite aller à Bucharest à partir d'Arad en effectuant le moins de kilomètres
- On veut résoudre ce problème avec A*

- Formaliser le problème
- Expliciter la fonction g à utiliser
- **Quelle pourrait être une bonne fonction h ?**
- Est-elle minorante, monotone ?
- Une fois h donnée, dérouler l'algorithme

Algorithme A* - exercice - solution

Voyage en Roumanie d'Arad à Bucharest

Arad

366

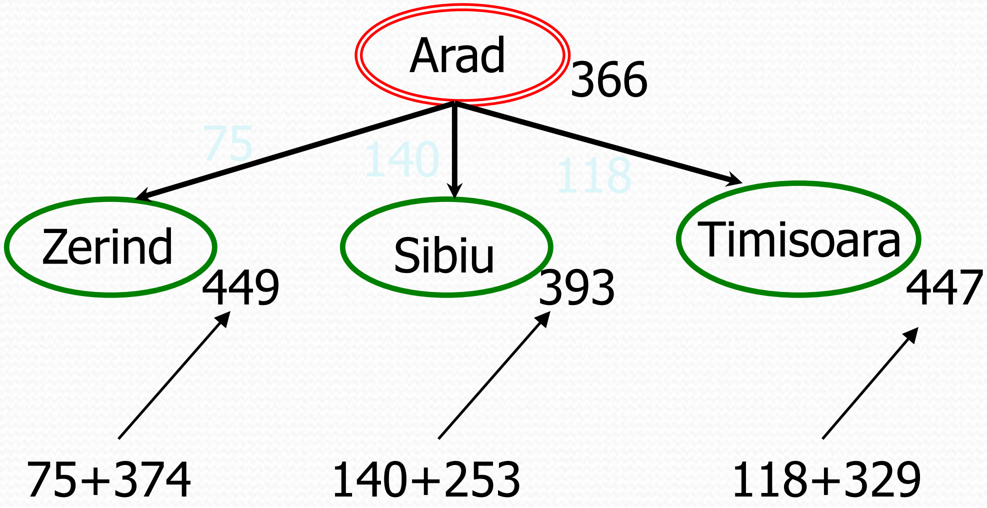
OUVERT

FERMÉ

$$f(e_{Arad}) = g(e_{Arad}) + h\,(e_{Arad}) \ = 0 + 366 = 366$$

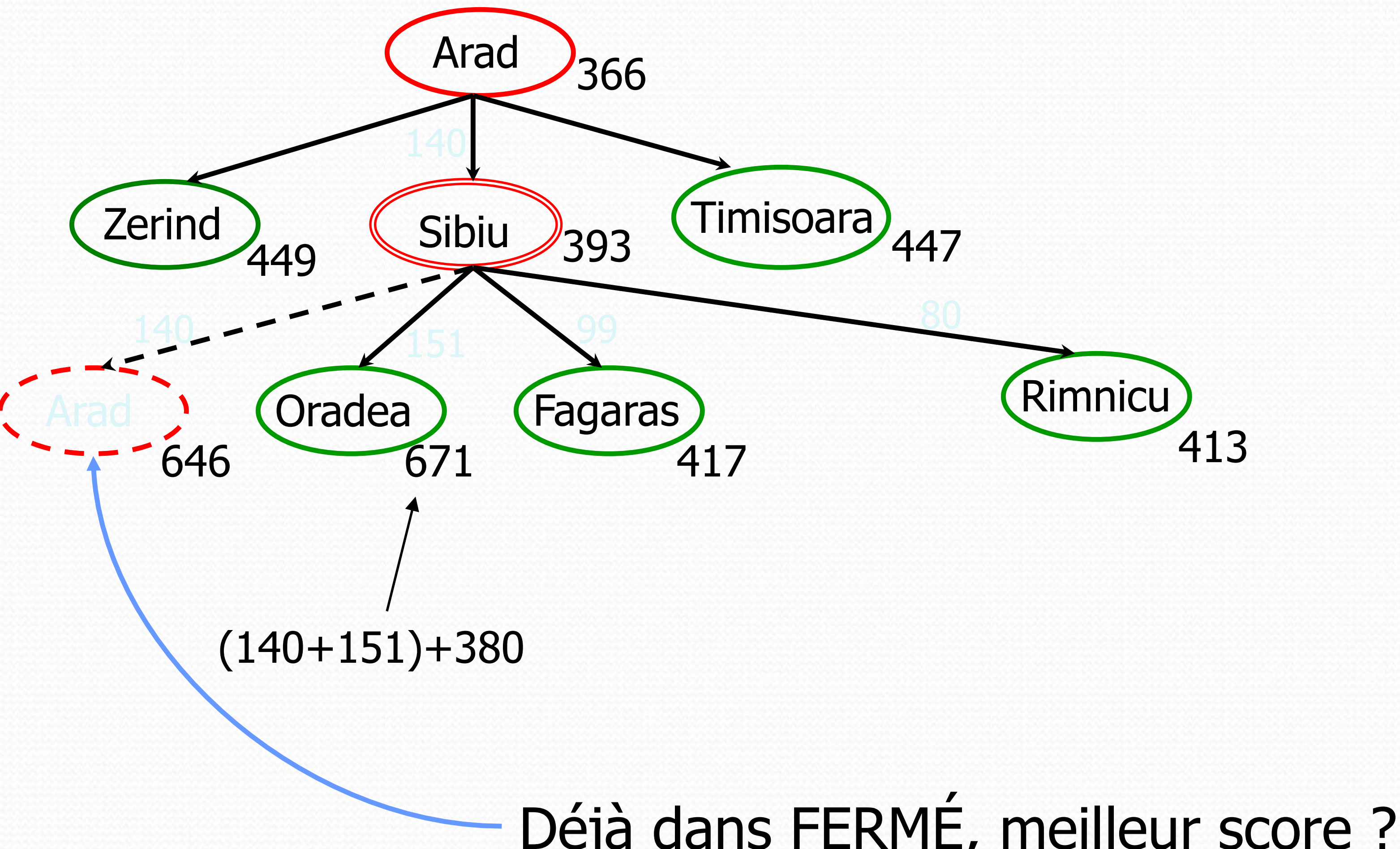
Algorithme A* - exercice - solution

Voyage en Roumanie d’Arad à Bucharest



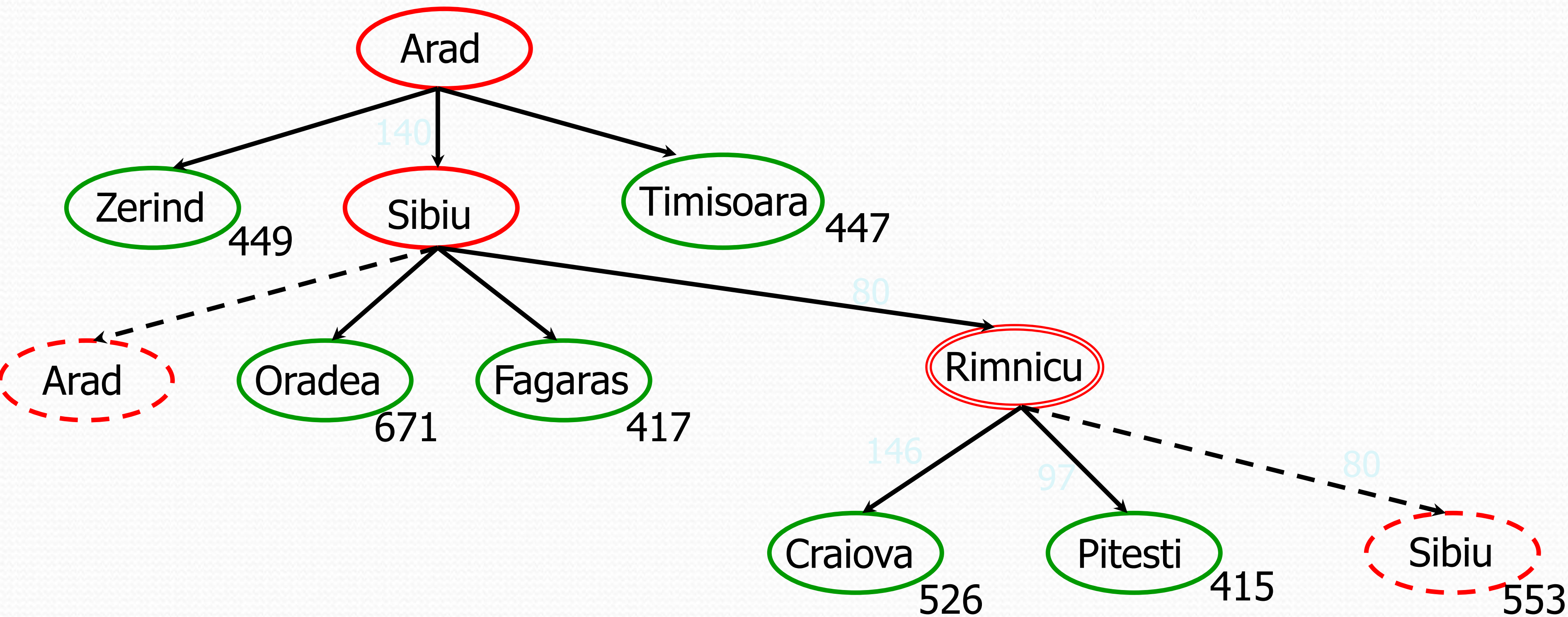
Algorithme A* - exercice - solution

<u>Voyage en Roumanie d'Arad à Bucharest</u>



Algorithme A* - exercice - solution

<u>Voyage en Roumanie d’Arad à Bucharest</u>



Algorithme A* - exercice - solution

Voyage en Roumanie d'Arad à Bucharest