Méthodes faibles

Définition

- Les méthodes faibles sont basées sur des techniques de parcours de graphe ou d'arbre
- Elles peuvent être adaptées à de nombreux domaines
- Leur efficacité est généralement moins bonne que celle d'algorithme « ad hoc »

Les missionnaires et les cannibales

- Trois missionnaires et trois cannibales doivent traverser un fleuve
- La barque ne peut porter que deux personnes au plus et il faut au moins une personne pour la manoeuvrer
- Lorsque l'on arrive sur une rive, tous les passagers descendent avant que la barque reparte.
- Il ne doit jamais y avoir strictement plus de cannibales que de missionnaires sur une rive (sauf s'il n'y a plus de missionnaire évidemment)
- Comment faire traverser le fleuve?

Etats, variables d'état, espace d'états

- **Variables d'état –** Tout problème est relatif à un certain ensemble d'objets que l'on peut décrire sous forme de variables d'état du problème.
- **État d'un problème –** Un état du problème est l'ensemble des valeurs que prennent ses variables d'état à un instant donné.
- **Espace d'états –** L'espace d'états d'un problème est l'ensemble des états possibles pour le problème considéré.

Systèmes de production

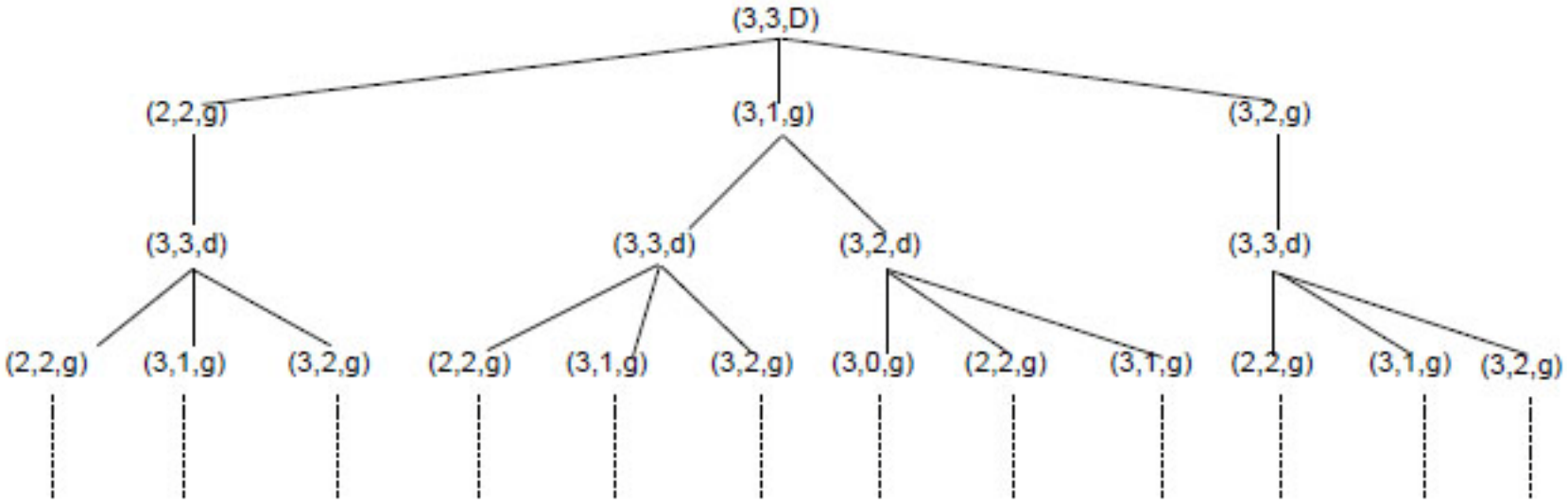
- **Système de production :** un ensemble de règles qui permettent à partir d'un état, de générer l'ensemble des états que l'on peut légalement atteindre.
- **Solution déductive :** aller de l'état initial à l'état final. (Chainage avant)
- **Solution inductive :** aller de l'état final à l'état initial (Chainage arrière)

Monotonie et commutativité

- **Monotonie:** *l'application d'une règle à t laisse les règles qui étaient applicables à t applicables à t+1.*
- **Commutativité partielle:** deux états X et Y, et (r1,r2...rn) tel que X->Xr1->....->Y
  Alors l'application d'une permutation (s(r1),...s(rn))donne aussi X->Xs(r1)->...->Y
- **Commutativité:** Monotonie et commutativité partielle.

Arbres

- **Arbres**: on relie chaque état généré à son prédécesseur sans faire de test d'occurrence.

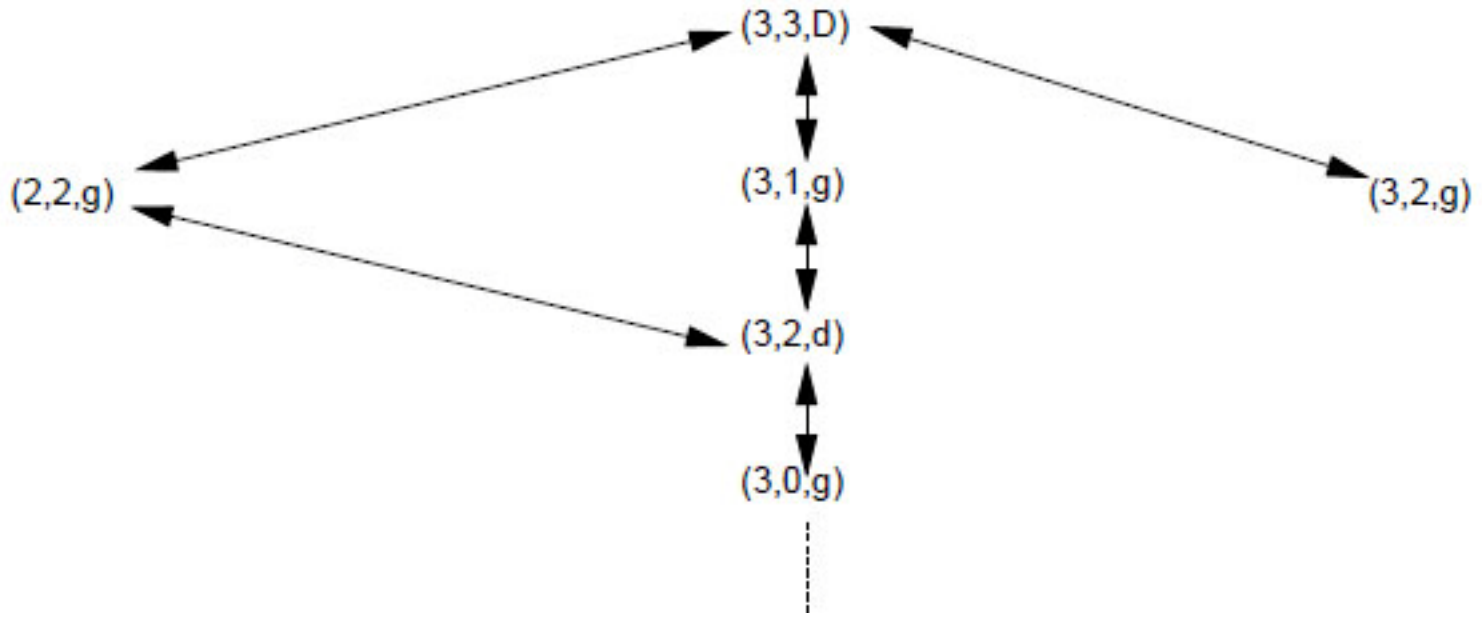


Arbres

- Avantages:
  - Rapidité (test d'occurrence couteux)
- Inconvénients:
  - La duplication d'états sature la mémoire
  - Risque de bouclages
- Très utilisé en théorie des jeux, même si on utilise souvent des techniques annexes de test d'occurrence (tables de transposition)

Graphes

- **Graphes:** lorsque l'on génère un nouvel état on vérifie qu'il n'a pas été déjà généré



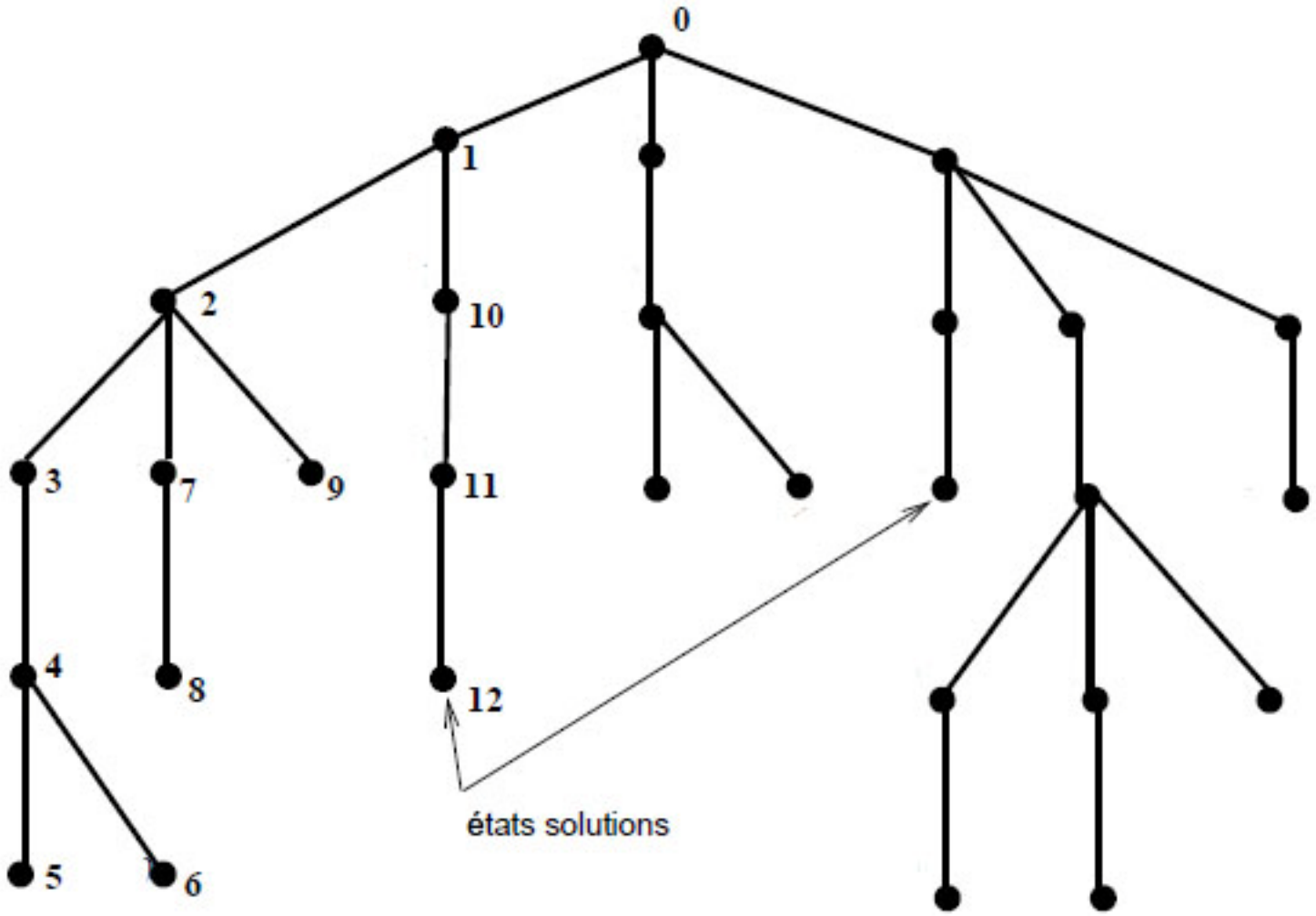
Algorithme du British Museum

- On génère un seul noeud en choisissant au hasard une règle de production.
- À partir de ce nouveau noeud on itère le processus jusqu'à l'obtention d'une solution.
- Si, à un instant donné, on aboutit à une impasse, on recommence l'opération depuis la racine de l'arbre.
- Algorithme du singe et de la machine à écrire

Recherche en profondeur et retour arrière

- Implantation plus intelligente de l'algorithme précédent.
- À chaque échec, on réalise un retour au noeud précédant le noeud où l'on a constaté l'impasse et on choisit une autre règle de production parmi celles qui n'ont pas été encore utilisées.

Recherche en profondeur et retour arrière



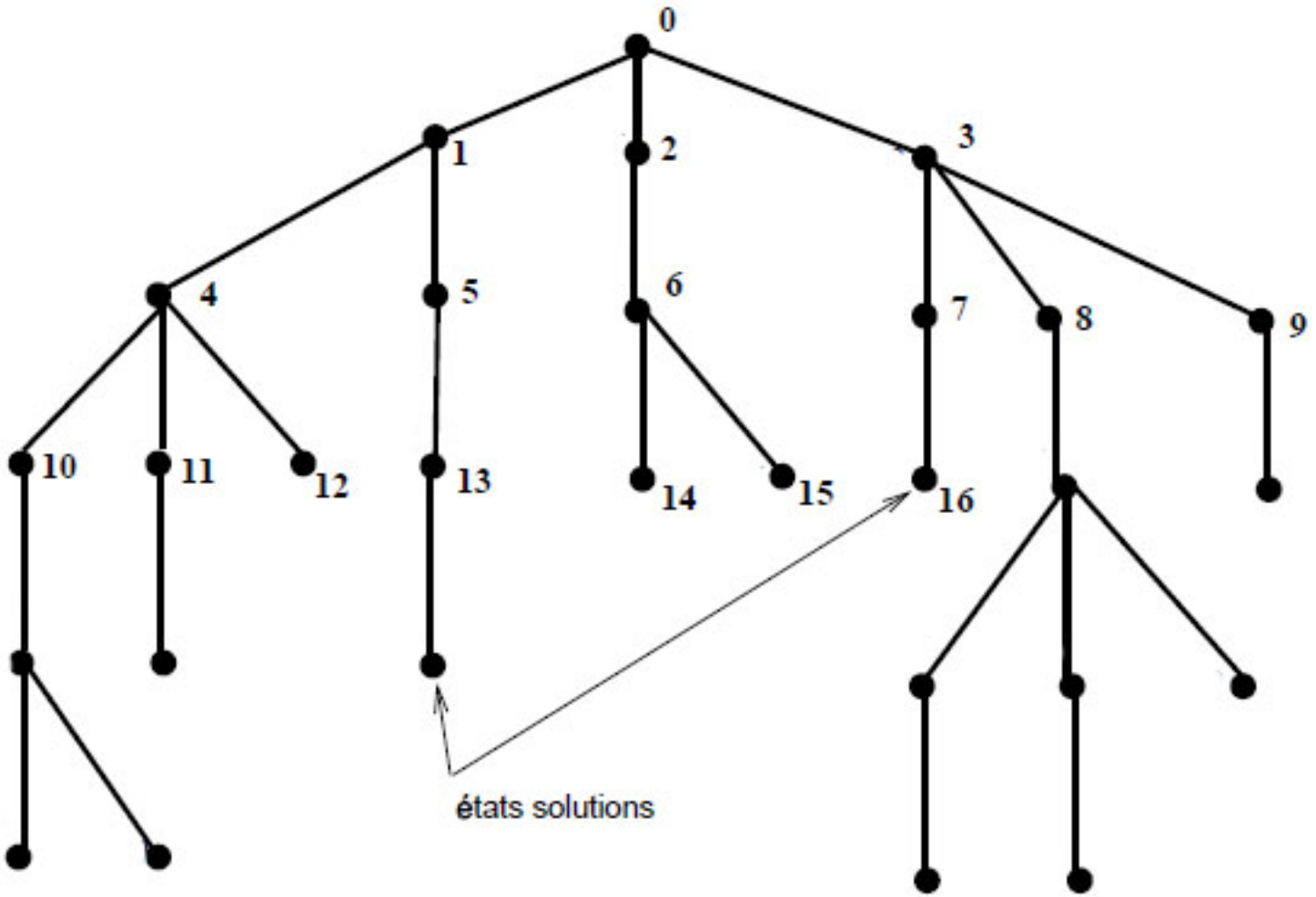
Recherche en profondeur et retour arrière

- Avantages:
  - Simple à implanter
- Inconvénients:
  - Plus efficace sur des graphes que sur des arbres (développement infini)
  - Ne sait pas profiter de la structure du problème
- Algorithme classique, utilisé par le langage PROLOG. Des variantes (iterative depth) utilisées en théorie des jeux

Recherche en largeur

- La recherche en largeur consiste à générer l'ensemble de tous les noeuds accessibles niveau par niveau.

Recherche en largeur



Recherche en largeur

- Avantages:
  - Cet algorithme garantit que l'on trouvera la solution, même s'il y a des cycles
  - Solution la moins profonde (la plus « courte »)
- Inconvénients:
  - Extrêmement couteuse en mémoire
  - Ne tient pas compte de la structure du problème.

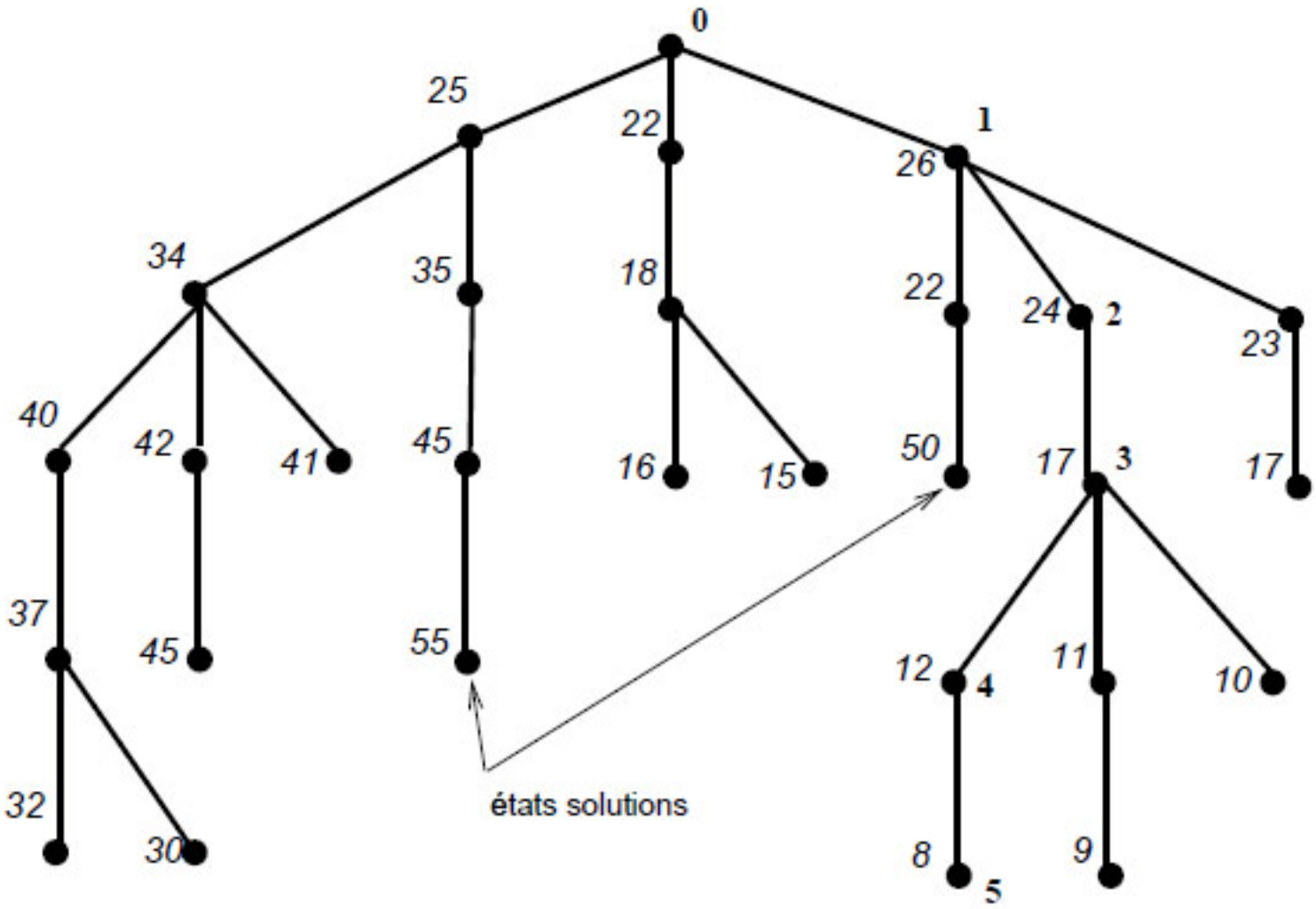
La notion d'heuristique

- Une heuristique a pour but de diriger la recherche dans l'arbre, de façon à réduire le temps de résolution du problème.
- Elle introduit des mécanismes qui dérivent de connaissances spécifiques au problème.
- On combine souvent l'utilisation d'une heuristique avec des méthodes de recherche ne garantissant plus la complétude de la résolution pour améliorer encore l'efficacité.

L’escalade

- L’escalade est une méthode de recherche en profondeur dans laquelle on introduit une fonction heuristique pour choisir à chaque étape le noeud à générer, au lieu de générer un noeud en prenant une règle de production au hasard.

L'escalade


0
25
22
1
26
34
35
18
22
24
2
23
40
42
41
45
16
15
50
17
3
17
37
45
55
12
4
11
10
32
30
états solutions
8
5
9

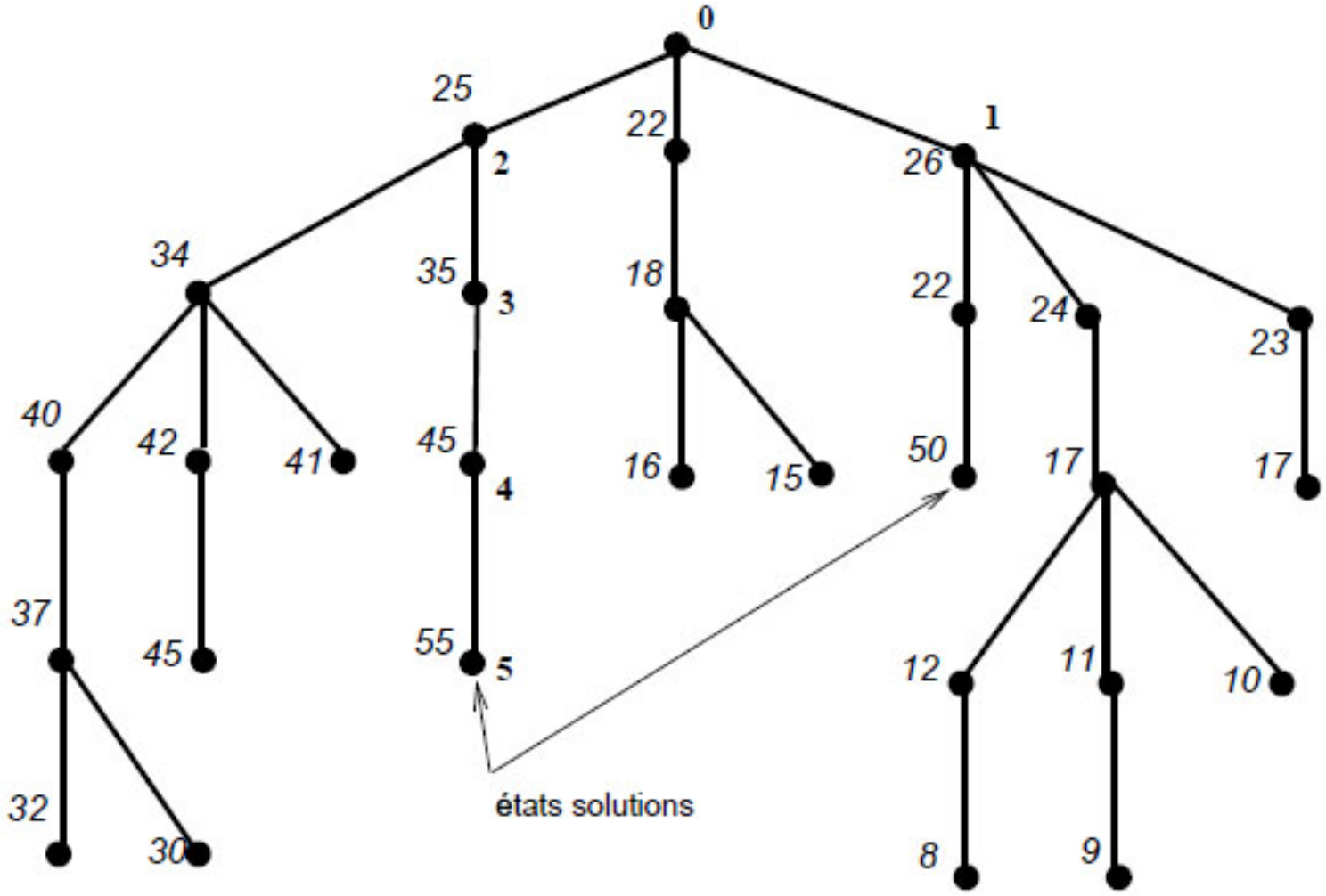
L'escalade

- Avantages:
  - Algorithme ultra-rapide
- Inconvénients:
  - Ne trouve pas systématiquement la meilleure solution
- Exemple: le problème du voyageur de commerce en choisissant systématiquement la ville la plus proche

Recherche meilleur en premier

- La recherche *meilleur en premier* est un compromis entre une recherche en largeur et une recherche en profondeur, en utilisant une fonction heuristique pour guider la génération des noeuds.
- On développe à chaque étape le meilleur noeud encore non développé. On inclut les noeuds générés dans l'ensemble des noeuds connus et on recommence le processus

Recherche meilleur en premier



Algorithme A*

- Algorithme de type meilleur en premier
- Applicable aux problèmes de cout additif
- Algorithme classique en robotique

Algorithme A*

- u_0 : l'état initial.
- T : l'ensemble des états terminaux.
- $P = \{p_1, p_2, \ldots, p_n\}$: l'ensemble des règles de production.
- h(u) : fonction heuristique qui estime le coût de passage de u à l'état final.
- k(u, v) : fonction qui donne le coût du passage par l'arc (u, v).

Algorithme A*

- G=u_0; g(u_0)=0; Tant que G faire
  - U=premier(G) ; G =G \ (u) ; D =D + (u)
  - Si u dans T fin_programme ; fin_si
  - Pour i allant de 1 jusqu'à n faire
    - On génère v à partir de u en appliquant pi
    - Si [v n'est pas dans D+G] ou [g(v) > g(u) + k(u, v)]
      - g(v)=g(u) + k(u, v)
      - f(v)=g(v) + h(v)
      - *U=père(v)*
      - *insérer-selon-f(G, v)*
    - fin_si
  - fin_faire
- fin_faire

Algorithme A*

- k(u, v) peut être prise identiquement nulle si on ne se préoccupe pas de la longueur du chemin menant à la solution.
- Si on cherche le chemin comprenant le moins d'arcs possibles, on prendra k(u, v) = 1 pour tout arc (u, v).
- h(u) : heuristique qui estime la distance entre l'état courant et l'état final.

L'heuristique h

- h(u) tente d'estimer le minimum sur tous les chemins (u, v_1, . . . , v_n, u_t), où ut est un état terminal, de la somme :
  - k(u, u_1) + k(u_1, u_2) + · · · + k(u_n, u_t)
- La valeur de ce minimum est noté $h^*(u)$, et il représente le coût du chemin minimal.

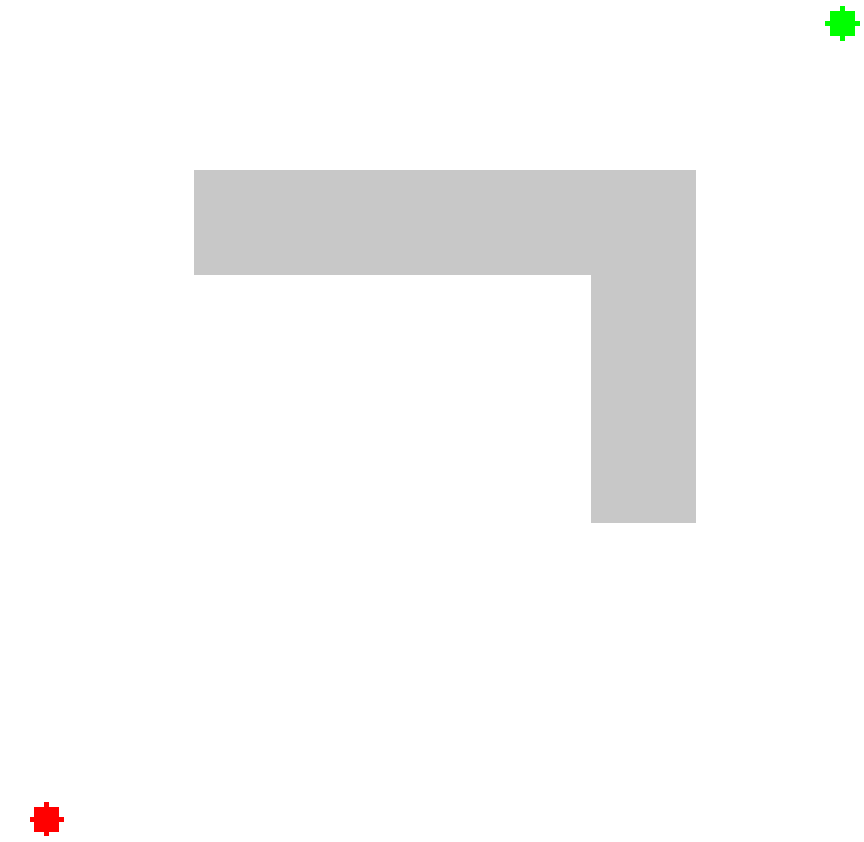
L'heuristique h

- **Heuristique parfaite**
  - Pour tout (u,v), h(u)=h(v) -> $h^*(u) = h^*(v)$
- **Heuristique presque parfaite**
  - Pour tout **(**u,v), h(u)<h(v) -> $h^*(u)<h^*(v)$
- **Heuristique monotone (consistante)**
  - Soit v fils de u, alors h(u)–h(v)<=k(u, v)
- **Heuristique minorante (admissible)**
  - Pour tout u, h(u)<=$h^*(u)$

Robot Motion Planning

- Trouver un chemin pour un robot dans un labyrinthe
  - k(u,v)=distance parcourue
  - h(u)=distance à vol d'oiseau entre le nœud u et la destination

Robot Motion Planning



L'heuristique h

- si h est parfaite, alors l'algorithme converge directement vers le but
- si h est minorante alors l'algorithme trouve toujours le chemin optimal
- Si h est monotone, h est minorante et garantit que pour tout noeud u développé, le chemin calculé est optimal (g(u) est effectivement égal au coût du plus court chemin menant à u).

Complexité de A* en fonction du nombre de noeuds N

- Dans le pire des cas, la complexité est 2^N.
- Si h est minorante, la complexité est N^2.
- Si h est monotone, la complexité est linéaire en N.
- Attention: une complexité même linéaire en N est généralement inutilisable. Ainsi dans le cadre du TSP, le nombre d'états N=n!

Complexité en fonction de K et M

- On cherche la complexité en fonction de
  - K facteur de branchement de l'arbre
  - M nombre minimal d'arcs entre initial et final
  - h parfaite: complexité linéaire en M.
  - h monotone: complexité en K^M.
  - h minorante:
    - $(1-r)h^* <= h : K^{aM}$
    - $h^*-(h^*)^{1/2} <= h : M^{1/2}\ K^{M^{1/2}}$
    - $h^*-\log(h^*) <= h : M^{\log(K)}$
    - $h^*-r <= h : MK^r$