UNIVERSITÉ CÔTE D'AZUR

Introduction à l'Intelligence Artificielle (L2 Portail Sciences et Technologies)

Andrea G. B. Tettamanzi
Laboratoire I3S – Équipe SPARKS
andrea.tettamanzi@univ-cotedazur.fr

univ-cotedazur.fr

Séance 2
Résolution de problèmes

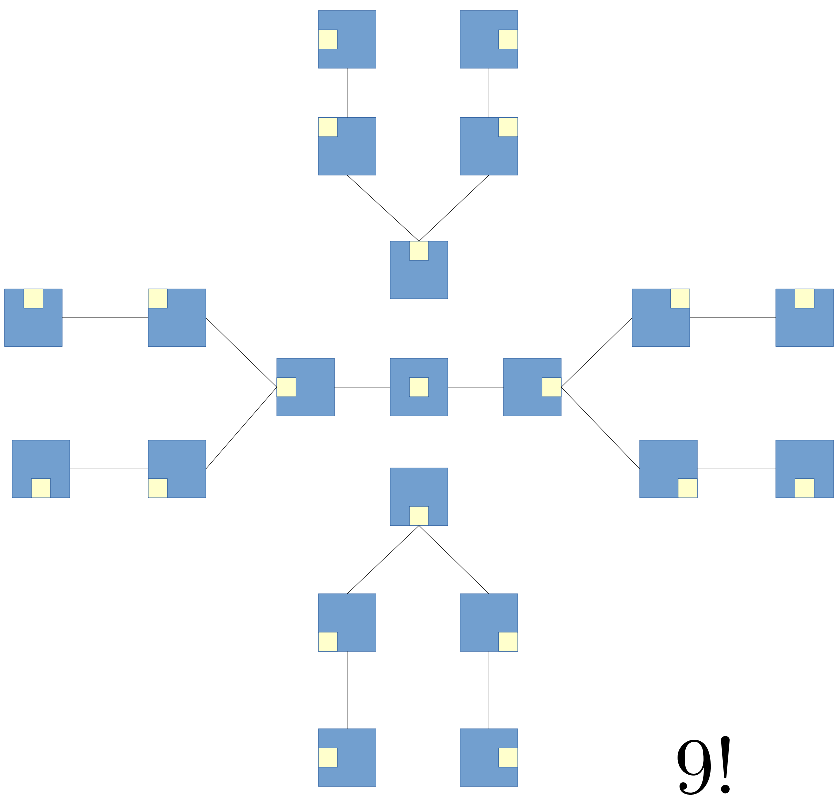
Plan pour cette séance

- Résolution par recherche
- Espace de recherche
- Algorithmes de recherche sur les graphes
- Stratégies de recherche aveugles
- Stratégies de recherche heuristiques
- Algorithme A*

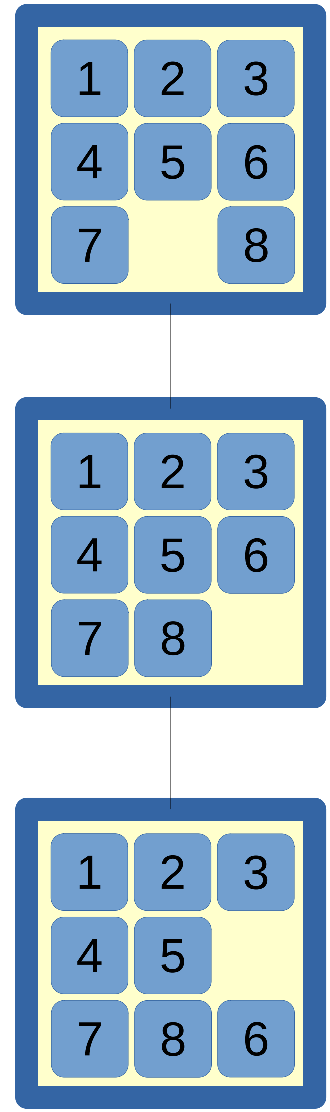
Problème

- État initial
- Actions disponibles à chaque état
- Modèle de transition : état $\times$ action $\rightarrow$ état
  - Espace des états (un graphe)
- Test objectif (l’état est-il une solution ?)
- Coût d’un pas (et d’une solution)

Jeu de taquin



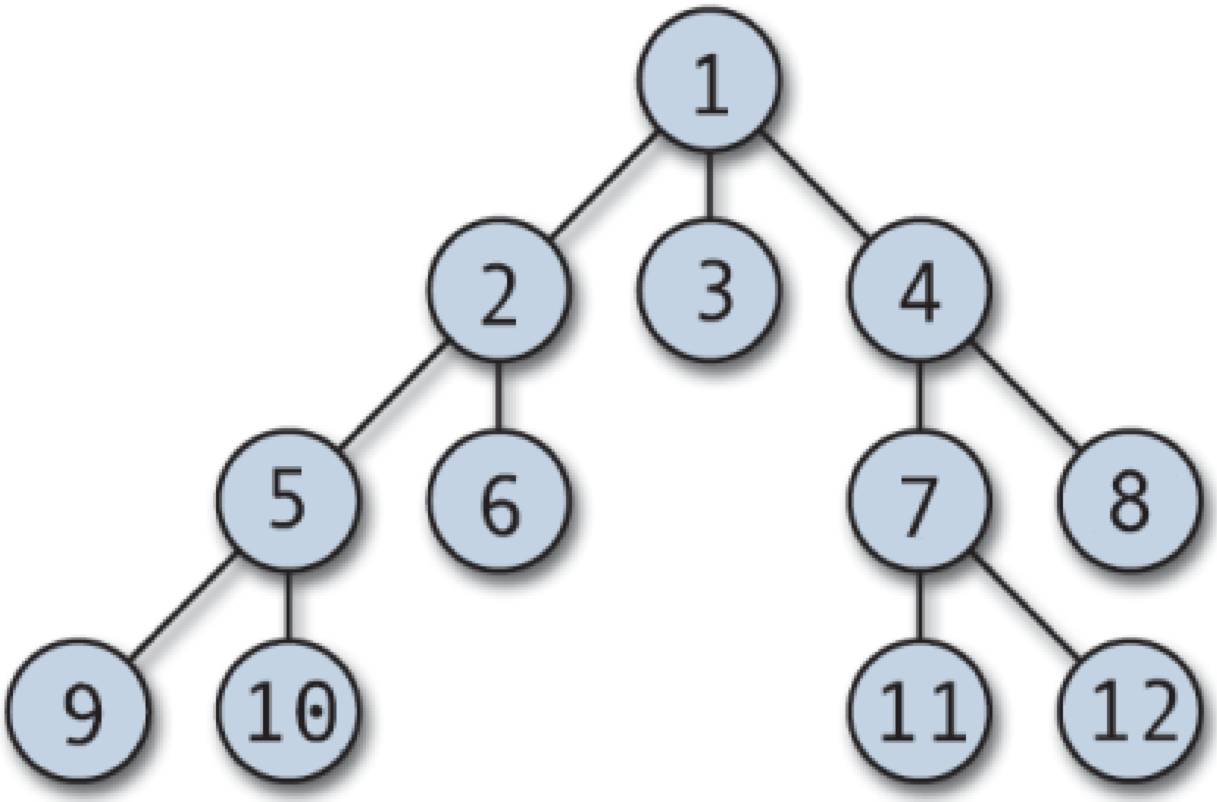
$$\frac{9!}{2} = 181440$$



Arbre

- Structure de données récursive
- Un arbre est formé par
  - Un nœud (dit « racine »), contenant
    - Des données ou une référence à des données
    - Des références (ou pointeurs) à des (sous-)arbres
  - Zéro ou plus sous-arbres
- Un nœud n'ayant pas des sous-arbres est dit « feuille »
- Les autres nœuds sont dits « nœuds internes »

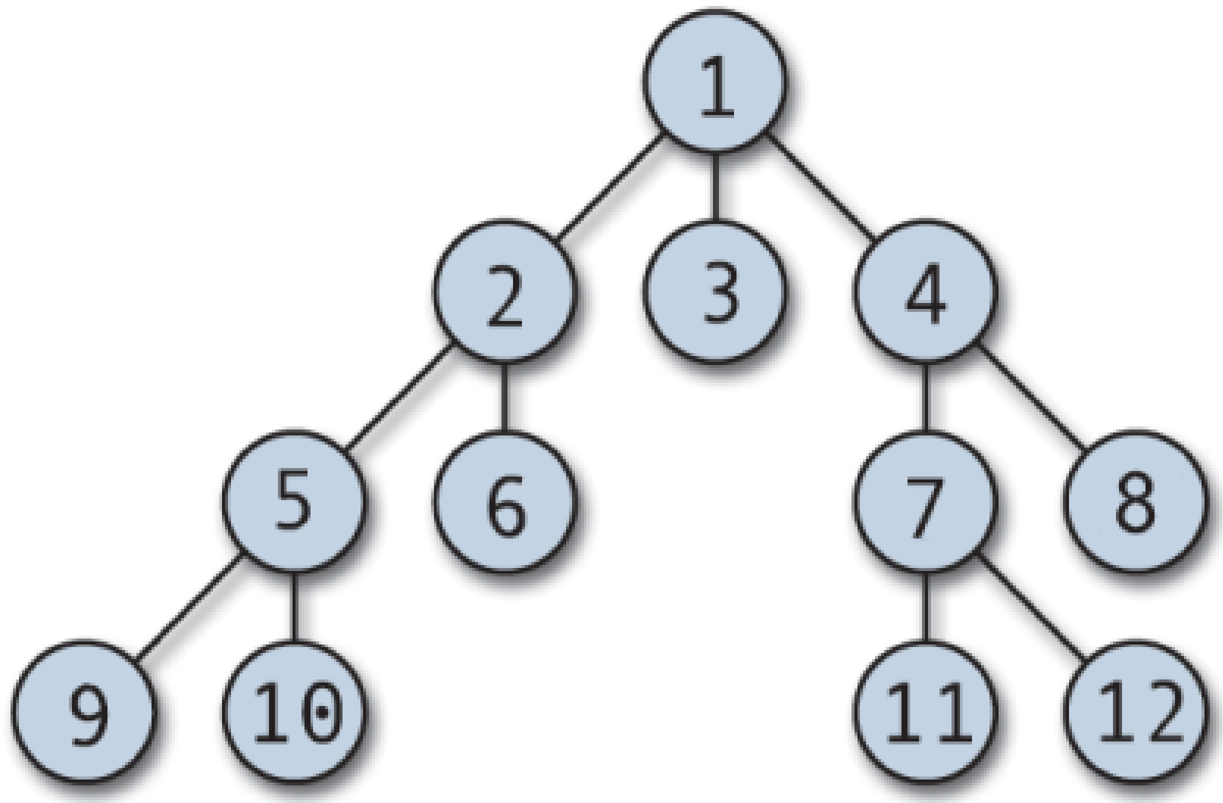
Arbre



Arbre

- La racine r de l'arbre est l'unique nœud ne possédant pas de parent
- Tout nœud x qui n’est pas la racine a
  - un unique parent, noté x.parent ou parent(x) (appelé « père » parfois)
  - 0 ou plusieurs fils ; x.fils ou fils(x) désigne l’ensemble des fils de x
- Si x et y sont des nœuds tels que x soit sur le chemin de r à y,
  - x est un ancêtre de y
  - y est un descendant de x
- Les feuilles n'ont pas de fils

Arbre



1 est la racine
9,10,6,3,11,12,8 sont les feuilles
11 est un descendant de 4, mais pas de 2
2 est un ancêtre de 10

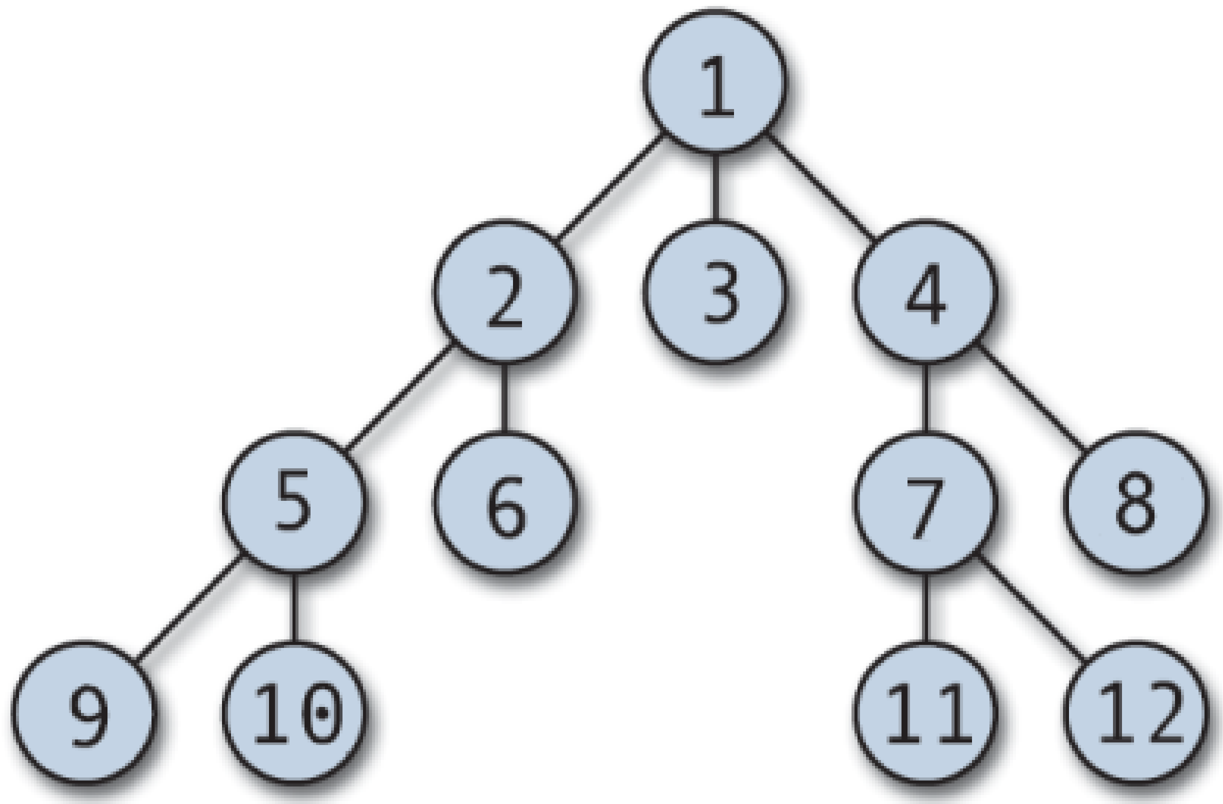
Arbre

- Quand il n’y a pas d’ambiguïté, on regarde les arêtes d’un arbre comme étant orientées de la racine vers les feuilles

- La profondeur d’un nœud (*depth*) est définie récursivement par
  - prof(v) = 0, si v est la racine
  - prof(v) = prof(parent(v)) + 1

- La hauteur d’un nœud (*height*) est la plus grande profondeur d’une feuille du sous-arbre dont il est la racine

Arbre

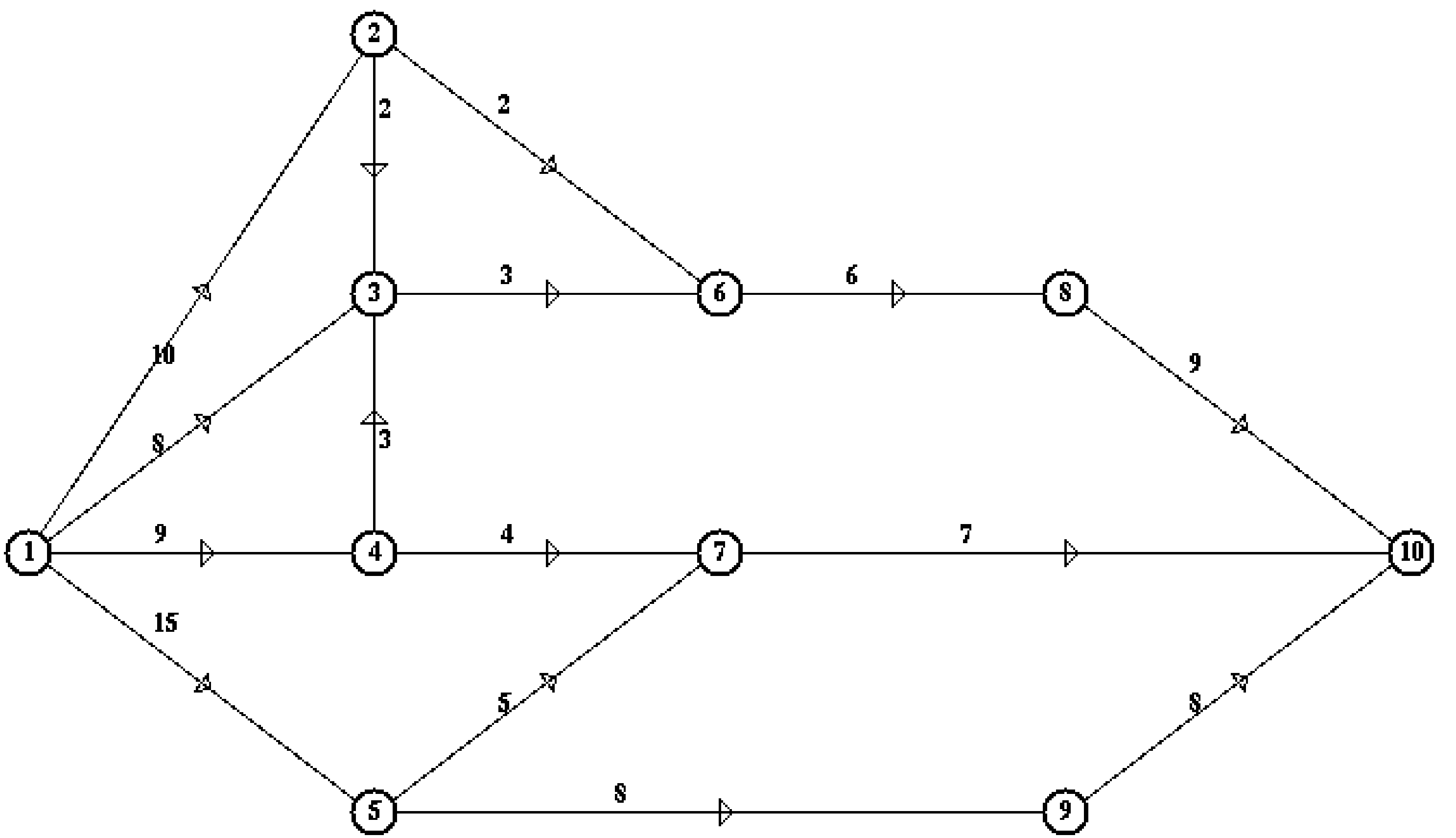


1 est la racine
2,3,4 sont à la profondeur 1
5,6,7,8 à la profondeur 2
La hauteur de 2 est 2, celle de 9 est 0, celle de 3 est 0, celle de 1 est 3

Graphe orienté

- Un Graphe Orienté $G = (X, U)$ est déterminé par la donnée :
  - d'un ensemble de sommets ou nœuds X
  - d'un ensemble ordonné U de couples de sommets appelés arcs.
- Si u = (i, j) est un arc de *G*, alors
  - i est l'extrémité initiale de u
  - j est l'extrémité terminale de u.
- Les arcs ont un sens (« orientés »).
  - L'arc u = (i, j) va de i vers j.
- Ils peuvent être munis d'un coût, d'une capacité etc. (arcs étiquetés)

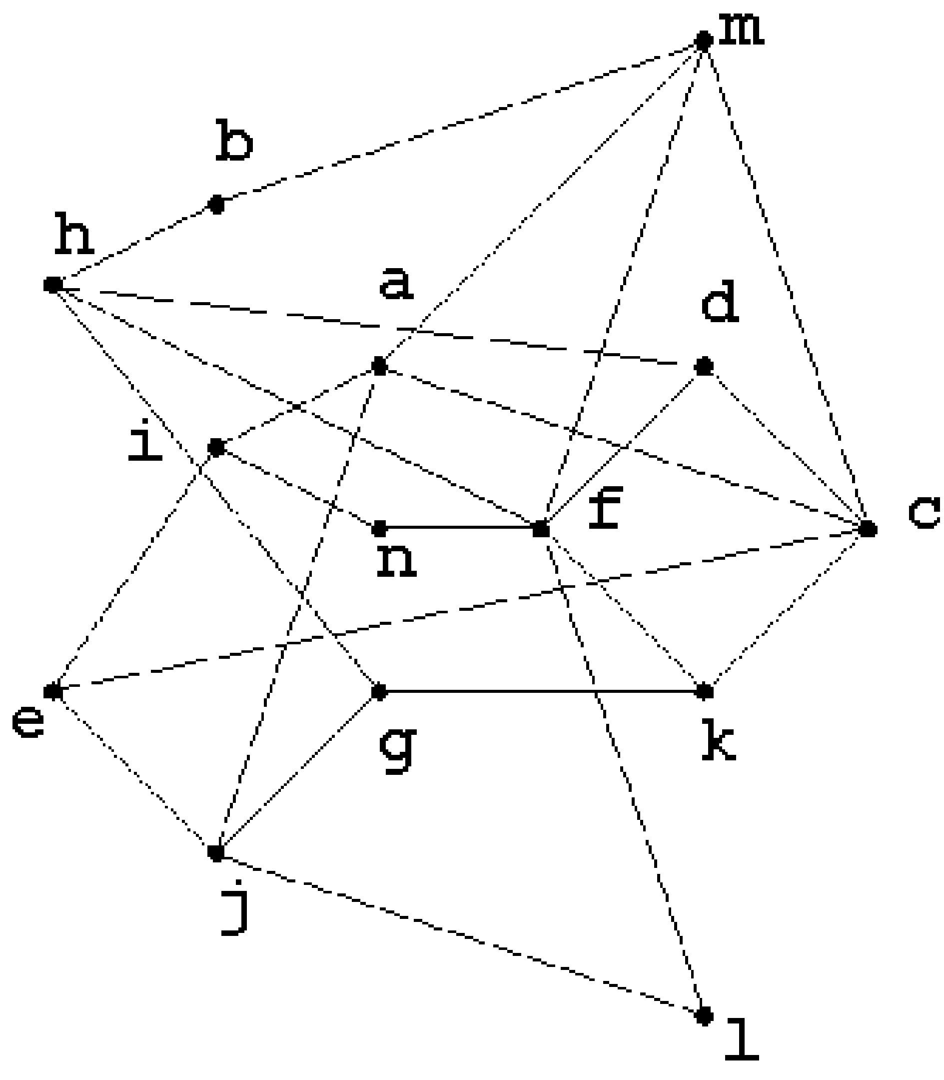
Graphe



Graphe non orienté

- Un Graphe Non Orienté $G = (X, U)$ est déterminé par la donnée :
  - d'un ensemble de sommets ou nœuds X
  - d'un ensemble de paires de sommets appelées « arêtes ».
- Les arêtes ne sont pas orientées

Graphe non orienté



Chemins et circuits

- Chemin de longueur q : séquence de q arcs $\{u_1, u_2, \ldots, u_q\}$ telle que
  - $u_1 = (i_0, i_1)$
  - $u_2 = (i_1, i_2)$
  - $u_q = (i_{q-1}, i_q)$

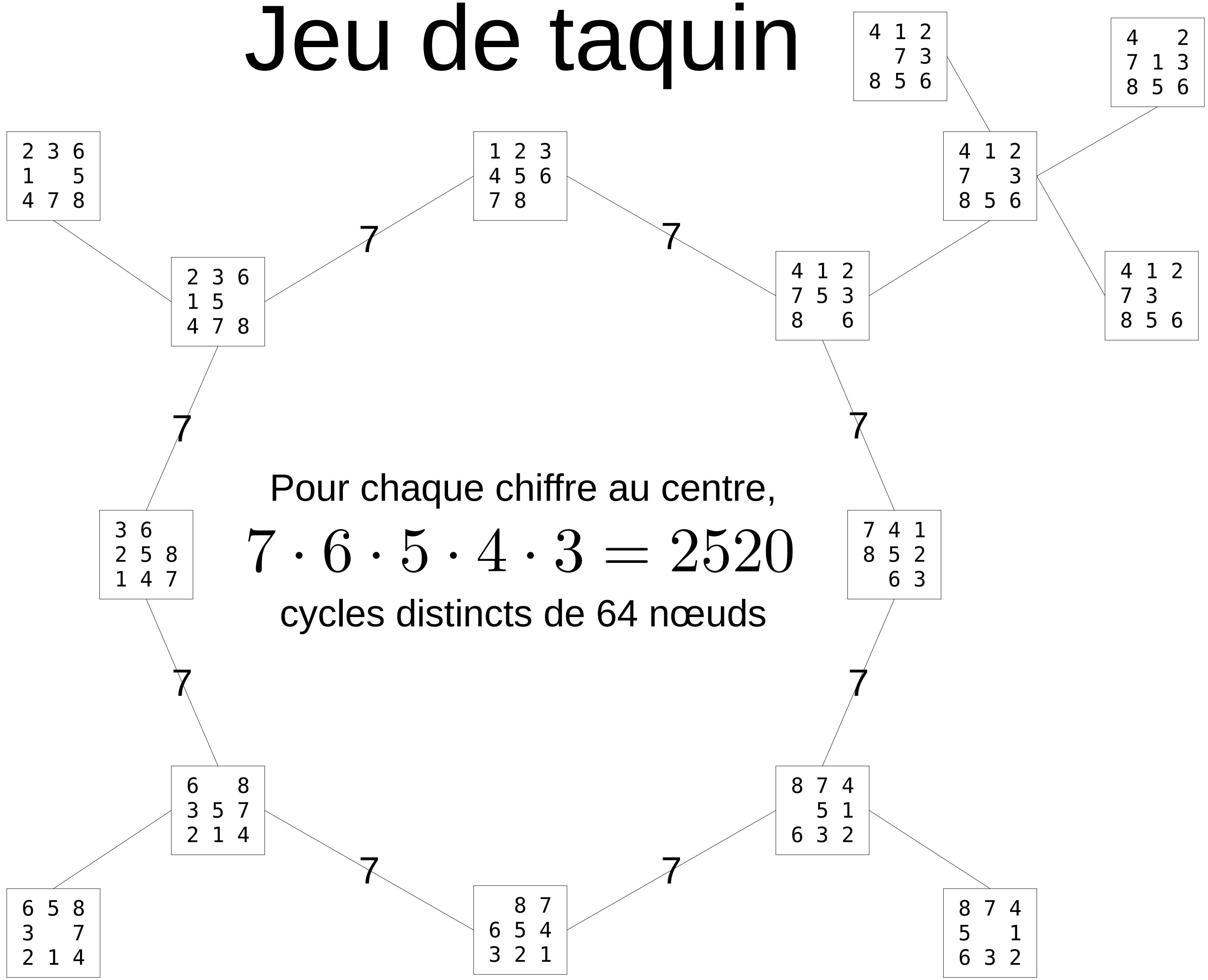
- Chemin : tous les arcs orientés dans le même sens
- Circuit : chemin dont les extrémités coïncident

Chaînes et cycles

- Chaîne de longueur q : séquence de q arêtes $\{u_1, u_2, \ldots, u_q\}$ telle que
  - $u_1 = (i_0, i_1)$
  - $u_2 = (i_1, i_2)$
  - $u_q = (i_{q-1}, i_q)$

- Cycle : chaîne dont les extrémités coïncident

Jeu de taquin

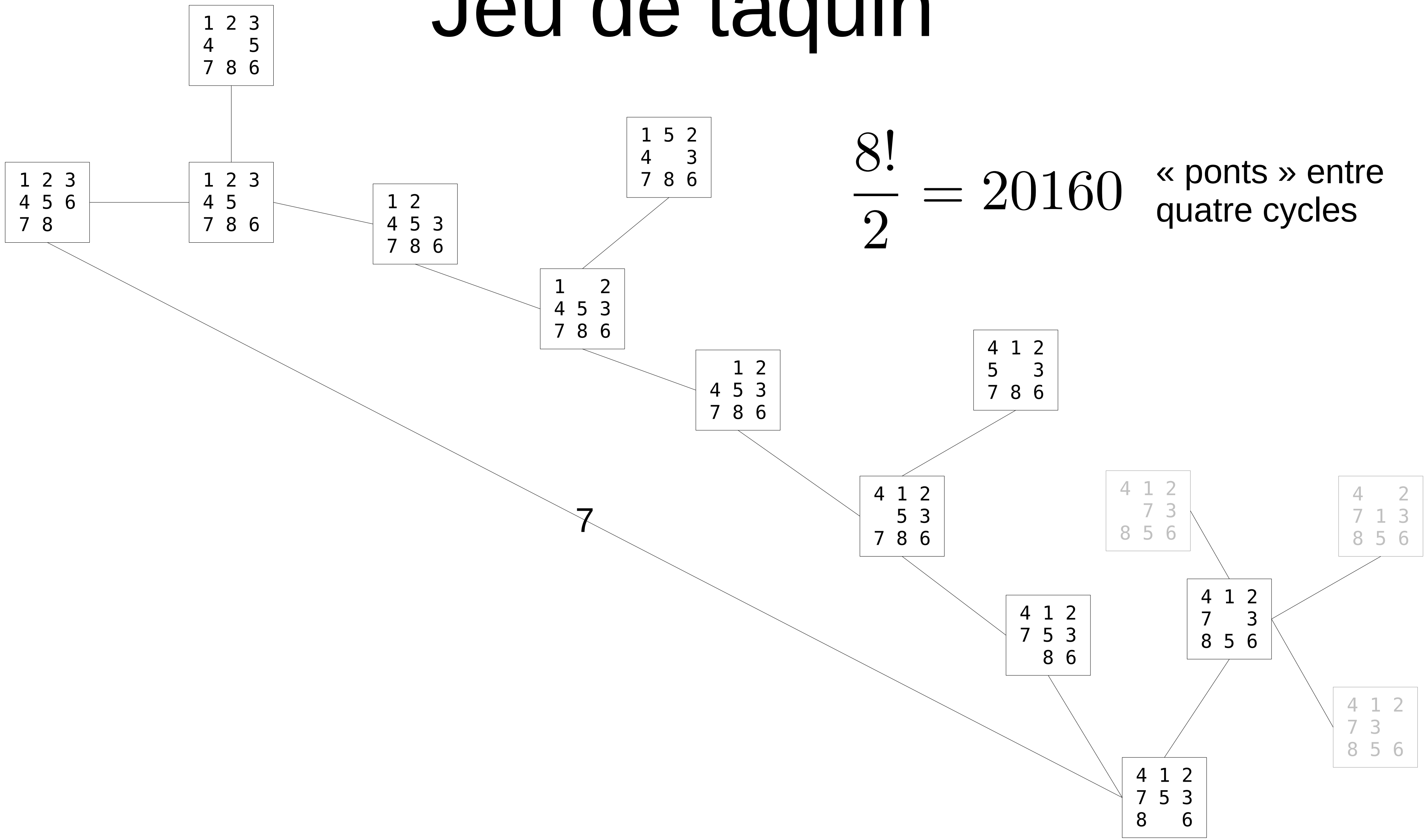


Pour chaque chiffre au centre,

$$7 \cdot 6 \cdot 5 \cdot 4 \cdot 3 = 2520$$

cycles distincts de 64 nœuds

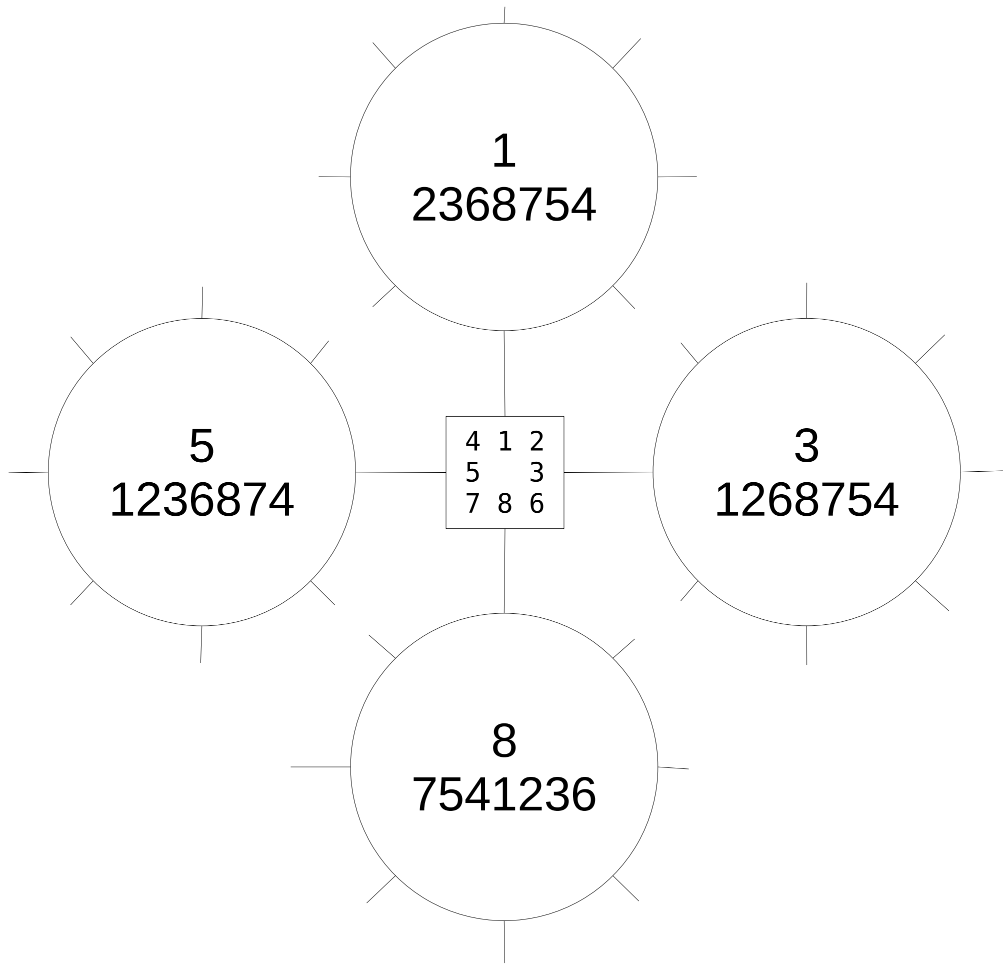
Jeu de taquin



$$\frac{8!}{2} = 20160$$

« ponts » entre quatre cycles

Jeu de taquin



Arbres comme graphes

- Un arbre est un graphe non orienté connexe et sans cycle

- Un graphe non orienté G ayant n sommets est un arbre si et seulement si il vérifie l'une des deux propriétés
  - G est connexe et possède n – 1 arêtes
  - G n'a pas de cycle et a n – 1 arêtes

Algorithme Général

Fonction RECHERCHE(problème), **renvoie** « échec » ou une solution

Front ← { état_initial }

Exploré ← { }

Répéter :

Si Front est vide, **renvoyer** « échec »

Choisir (et retirer) un nœud du front

Si le nœud contient un état final, **renvoyer** la solution correspondante

Ajouter le nœud à l'ensemble Exploré

Développer le nœud

Pour chacun de ses successeurs :

Si successeur ni dans Front ni dans Exploré :

Ajouter successeur au Front

Stratégies de recherche « aveugles »

- Aucune information sur la structure du problème
- On sait seulement tester si un état est final et générer un nouveau état en appliquant une action
- Les stratégies ne diffèrent que par l'ordre par lequel les nœuds sont développés
  - Exploration en largeur d'abord
  - Exploration à coût uniforme
  - Exploration en profondeur d'abord
  - Exploration en profondeur limitée
  - Exploration par approfondissement itératif

Exploration en largeur d’abord

Fonction LARGEUR(problème), **renvoie** « échec » ou une solution

Front ← file(état_initial)

Exploré ← { }

Répéter :

Si Front est vide, **renvoyer** « échec »

Défiler le premier nœud du front

Si le nœud contient un état final, **renvoyer** la solution correspondante

Ajouter le nœud à l’ensemble Exploré

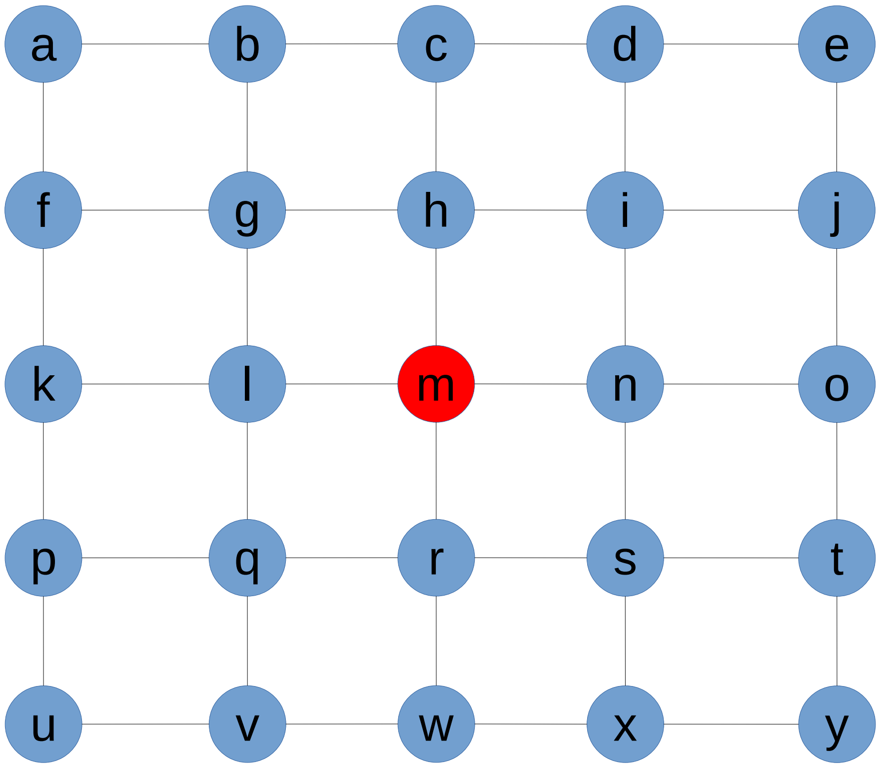
Développer le nœud

Pour chacun de ses successeurs :

Si successeur ni dans Front ni dans Exploré :

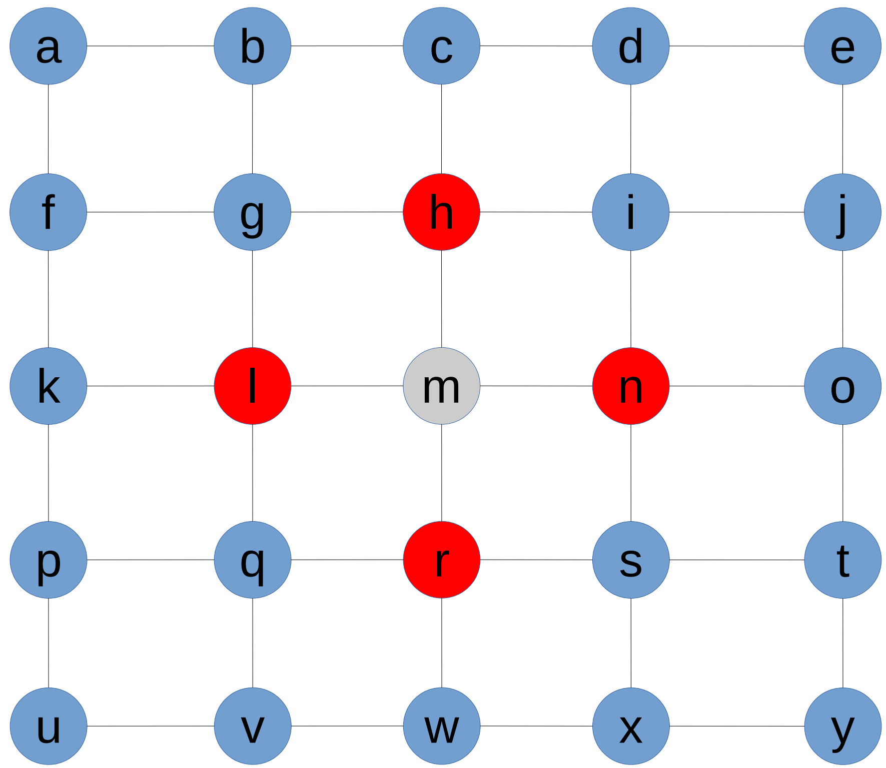
Enfiler successeur dans Front

Exploration en largeur d'abord



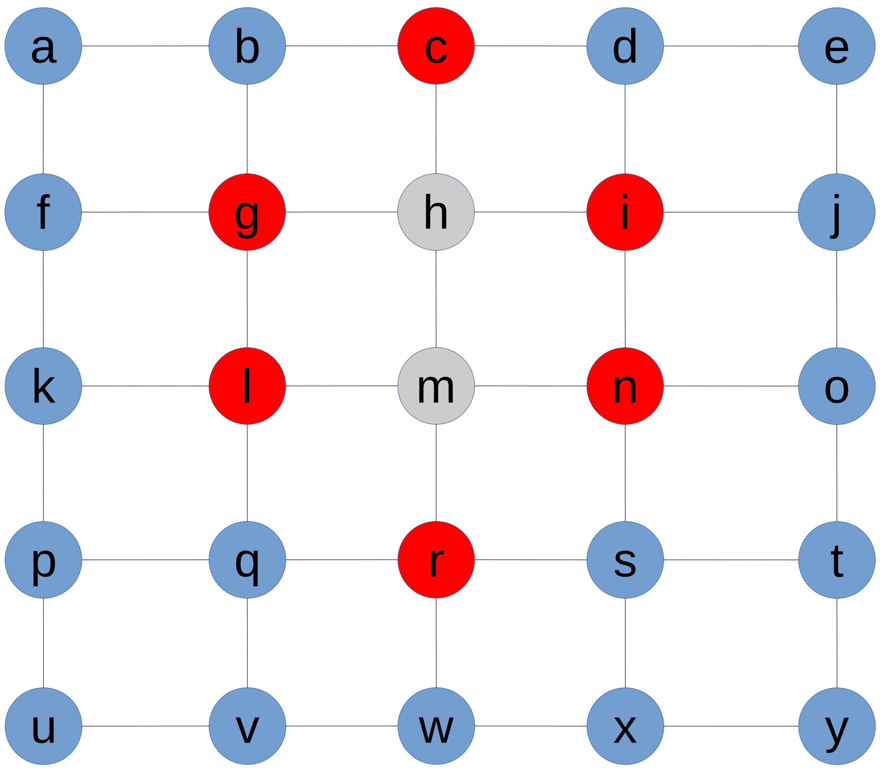
Front = [m]
Exploré = { }

Exploration en largeur d'abord



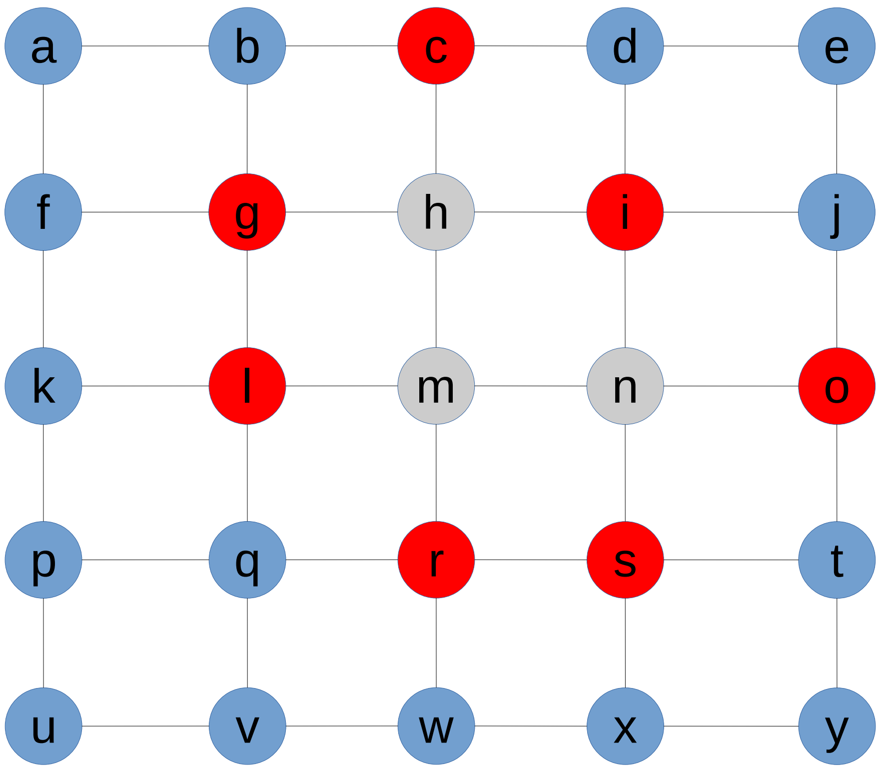
Front = [h, n, r, l]
Exploré = { m }

Exploration en largeur d’abord



Front = [n, r, l, g, c, i]
Exploré = { h, m }

Exploration en largeur d'abord

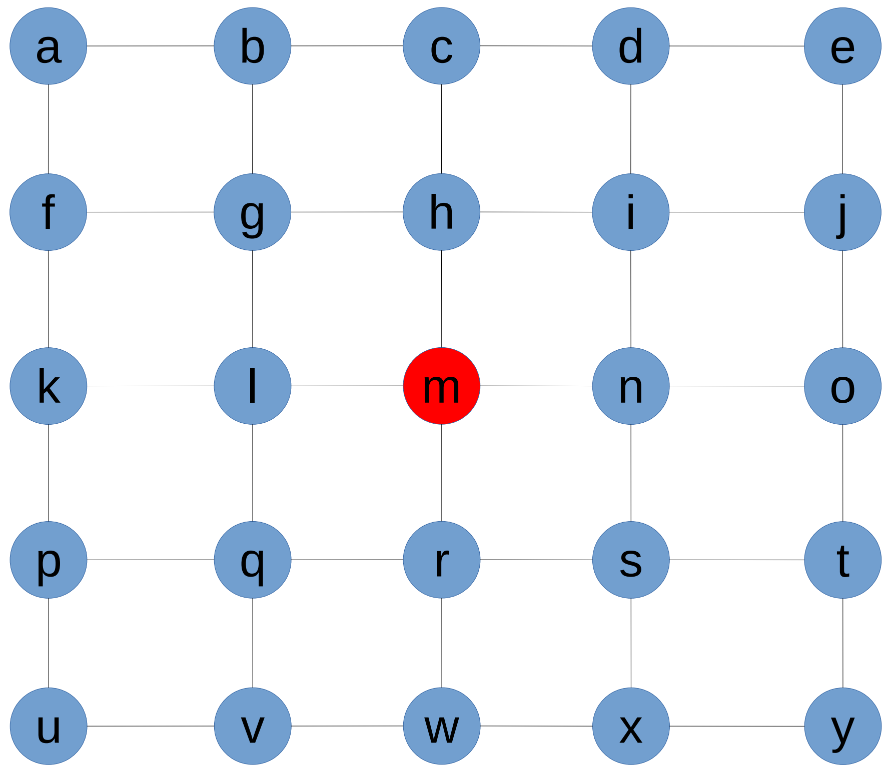


Front = [r, l, g, c, i , o, s]
Exploré = { h, m, n }

Exploration en profondeur d’abord

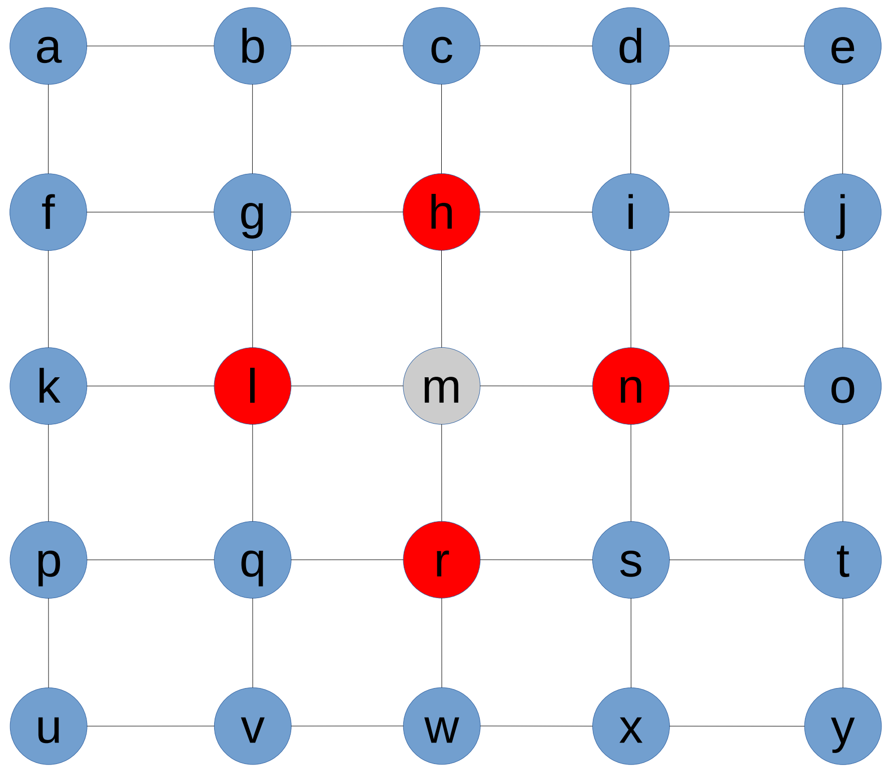
Fonction PROFONDEUR(problème), **renvoie** « échec » ou une solution
  Front ← pile(état_initial)
  Exploré ← { }
  Répéter :
    Si Front est vide, **renvoyer** « échec »
    Dépiler le nœud au sommet de la pile Front
    Si le nœud contient un état final, **renvoyer** la solution correspondante
    Ajouter le nœud à l’ensemble Exploré
    Développer le nœud
    Pour chacun de ses successeurs :
      Si successeur ni dans Front ni dans Exploré :
        Empiler successeur sur la pile Front

1)Exploration en profondeur d'abord



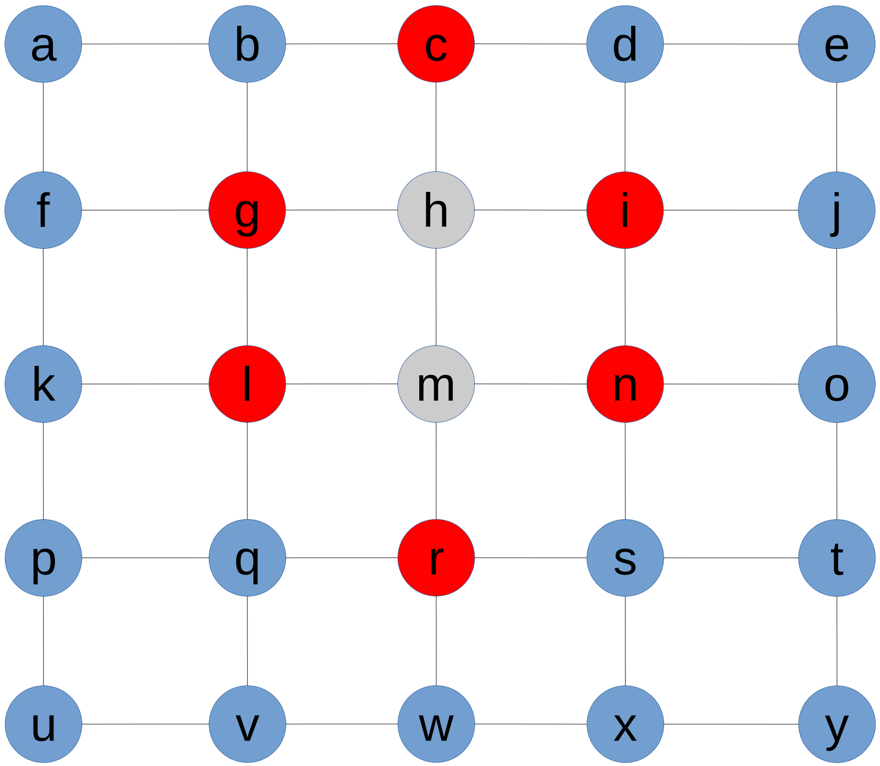
Front = [m]
Exploré = { }

1)Exploration en profondeur d'abord



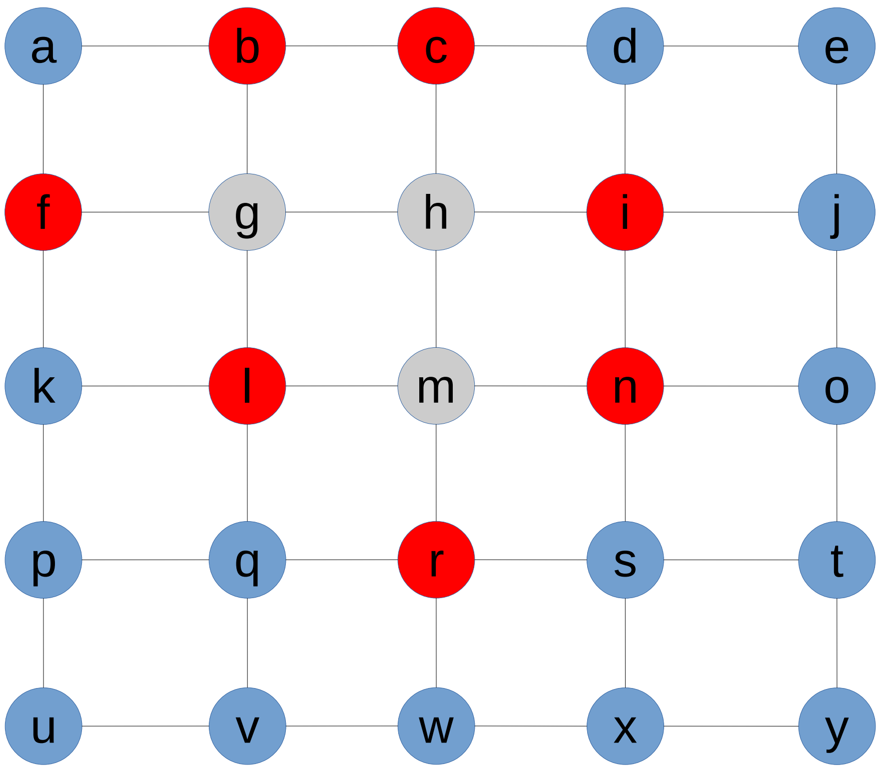
Front = [h, n, r, l]
Exploré = { m }

1)Exploration en profondeur d’abord



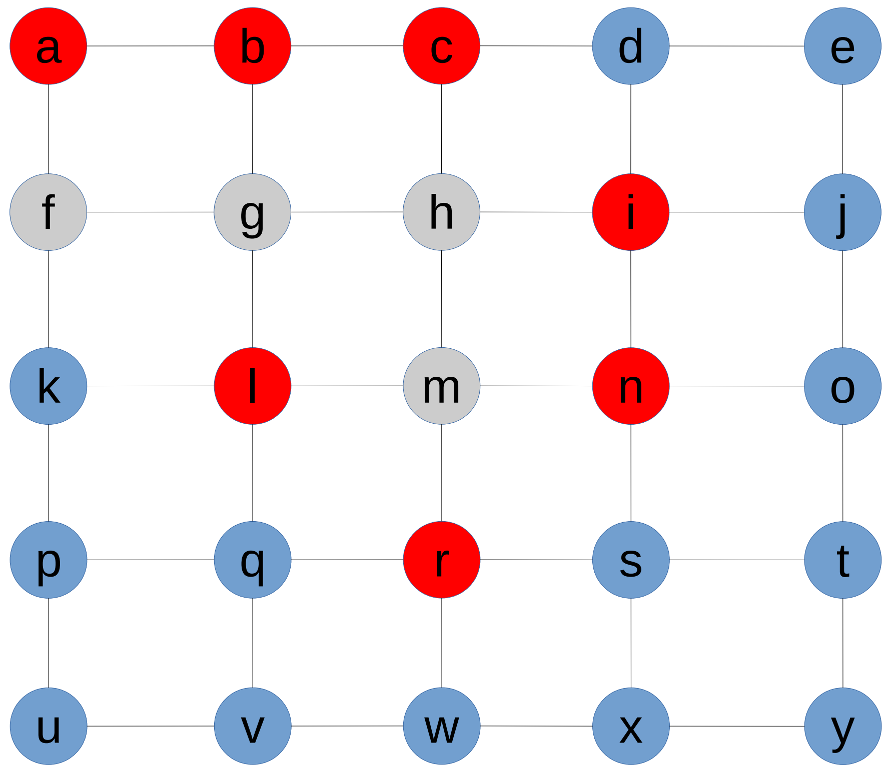
Front = [g, c, i, n, r, l]
Exploré = { h, m }

1)Exploration en profondeur d'abord



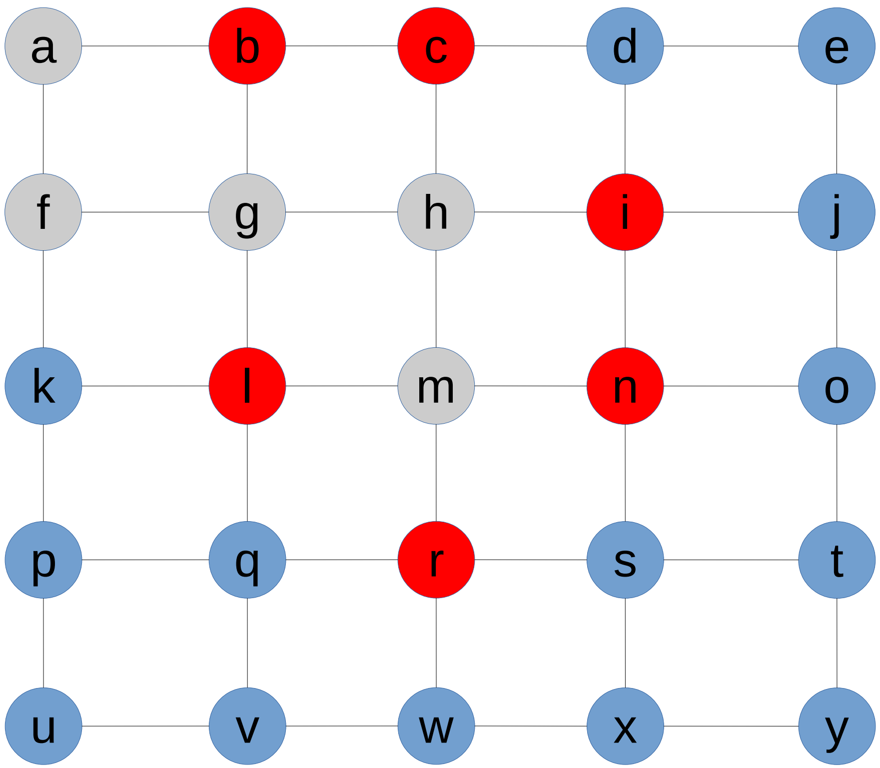
Front = [f, b, c, i, n, r, l]
Exploré = { g, h, m }

1)Exploration en profondeur d’abord



Front = [a, b, c, i, n, r, l]
Exploré = { f, g, h, m }

1)Exploration en profondeur d'abord



Front = [b, c, i, n, r, l]
Exploré = { a, f, g, h, m }

Exploration à coût uniforme

Fonction COUT_UNIFORME(problème), **renvoie** « échec » ou une solution

Front ← { état_initial }

Exploré ← { }

Répéter :

Si Front est vide, **renvoyer** « échec »

Choisir (et retirer) de Front le nœud n tel que coût(n) est le moindre

Si n contient un état final, **renvoyer** la solution correspondante

Ajouter le nœud à l'ensemble Exploré

Développer le nœud

Pour chacun de ses successeurs :

Si successeur ni dans Front ni dans Exploré :

Ajouter successeur au Front

Si successeur est déjà dans Front mais avec un coût supérieur

Mettre à jour coût (et chemin) du successeur

Recherche heuristique

- Idée : choisir le nœud à développer suivant une fonction d’évaluation, f(n)
- Coût du chemin jusqu’à n : g(n)
- Estimation du coût du chemin le moins cher pour aller de n à l’objectif : h(n)
- $f(n) = g(n) + h(n)$

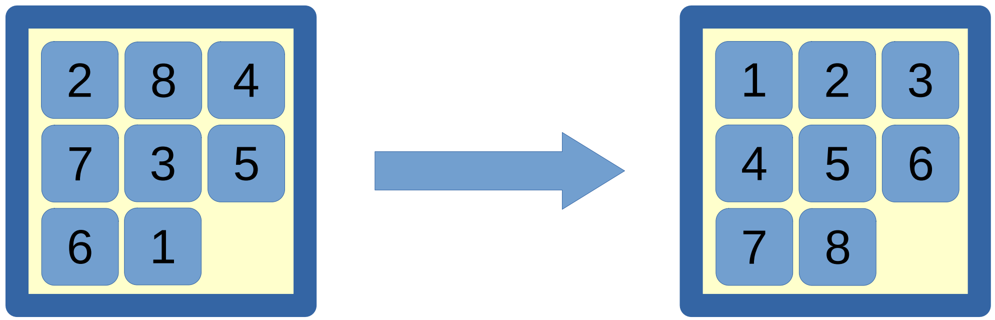
Algorithme A*

Fonction A*(problème), **renvoie** « échec » ou une solution
  Front ← { état_initial }
  Exploré ← { }
  Répéter :
    Si Front est vide, **renvoyer** « échec »
    Choisir (et retirer) de Front le nœud n tel que f(n) est le moindre
    Si n contient un état final, **renvoyer** la solution correspondante
    Ajouter le nœud à l'ensemble Exploré
    Développer le nœud
    Pour chacun de ses successeurs :
      Si successeur ni dans Front ni dans Exploré :
        Ajouter successeur au Front
      Si successeur est déjà dans Front mais avec un coût supérieur
        Remplacer successeur, avec son coût actuel

Optimalité de A*

- Une heuristique **admissible** ne surestime jamais le coût pour atteindre l'objectif
- Une heuristique est **cohérente** (ou **monotone**) si, pour tout node n et successeur n' obtenu par l'action a,
  $h(n) \leq c(n, a, n') + h(n')$
- Toute heuristique cohérente est aussi admissible
- Si h est cohérente, A* est optimal

A* et Jeu de taquin



Deux heuristiques :

1) Distance de Hamming : nombre de tuiles hors place

2) Distance de Manhattan : somme des distances de l’objectif pour chaque tuile

Tuile :	1	2	3	4	5	6	7	8	Total
Distance :	3	1	2	3	1	3	1	2	16

A* et Jeu de taquin