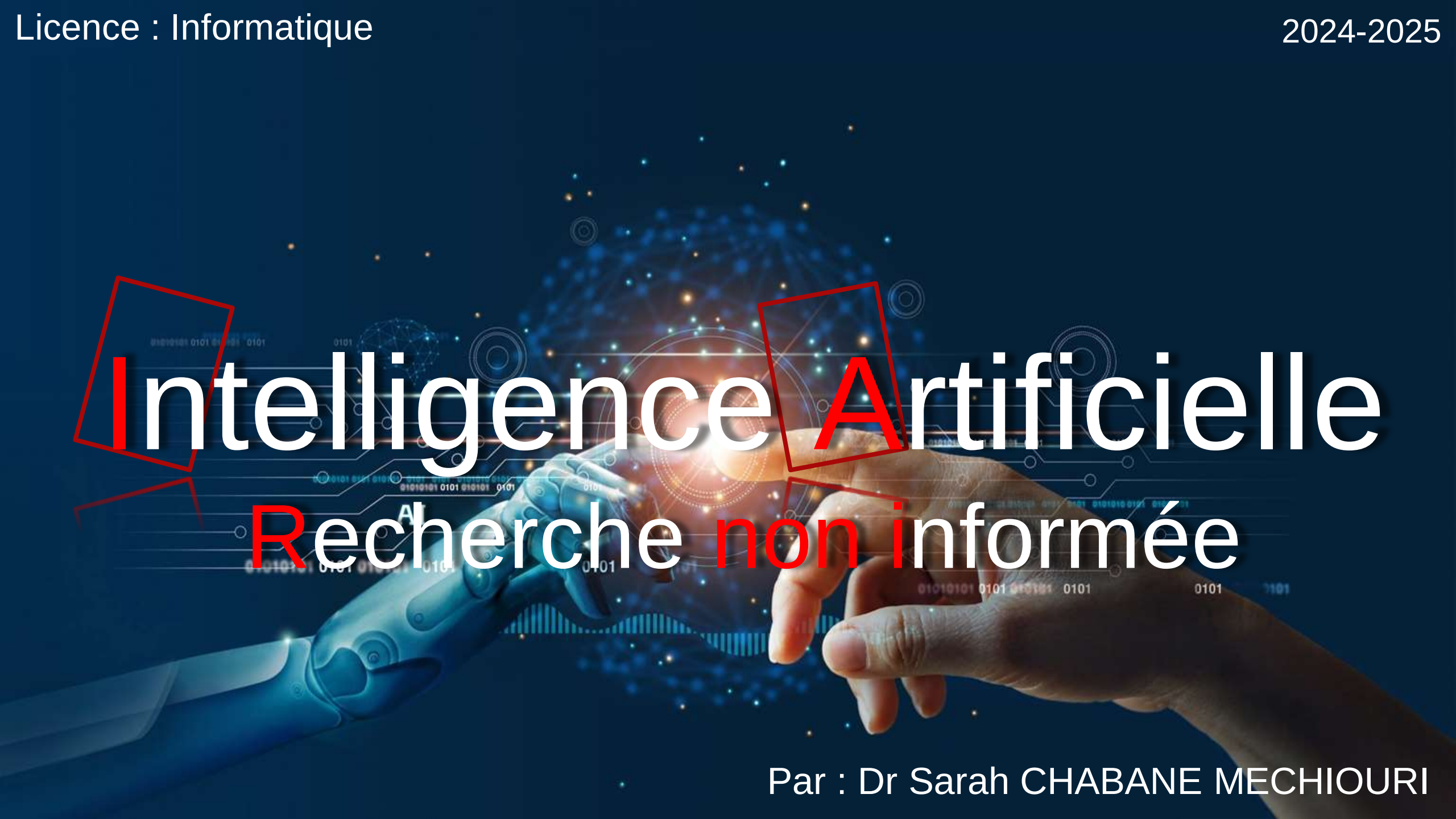
Licence : Informatique

2024-2025



Intelligence Artificielle

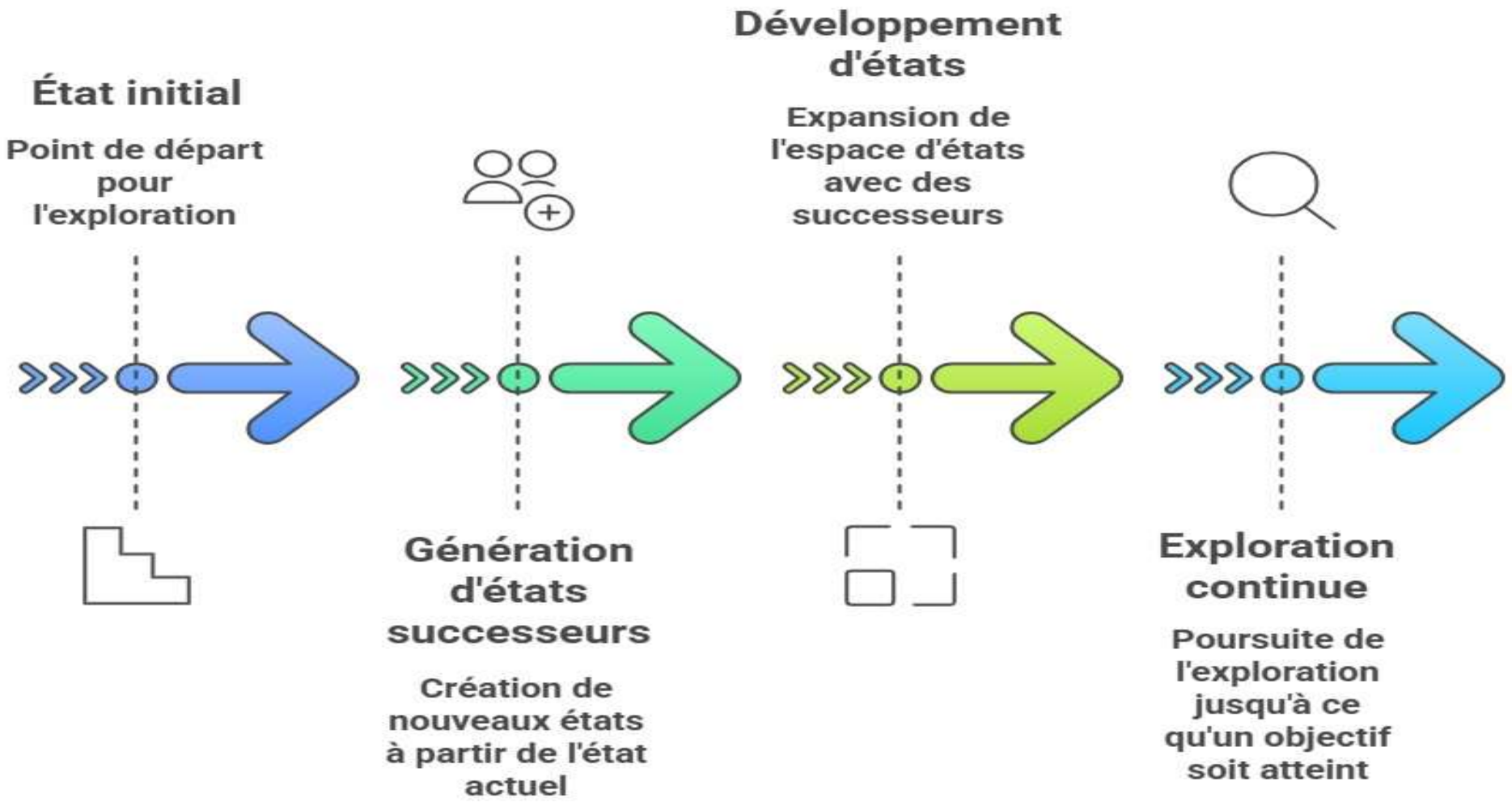
Recherche non informée

Par : Dr Sarah CHABANE MECHIOURI

CONTENU

- *Arbre de recherche*
- Stratégie de recherche
- Structure générale d'un algorithme de résolution
- Evaluation des algorithmes de recherche
- Stratégies aveugles (Non informées)
- Stratégies heuristiques (Informées)

EXPLORATION DE L'ESPACE D'ÉTATS



COMMENT EXPLORER ? ARBRE DE RECHERCHE

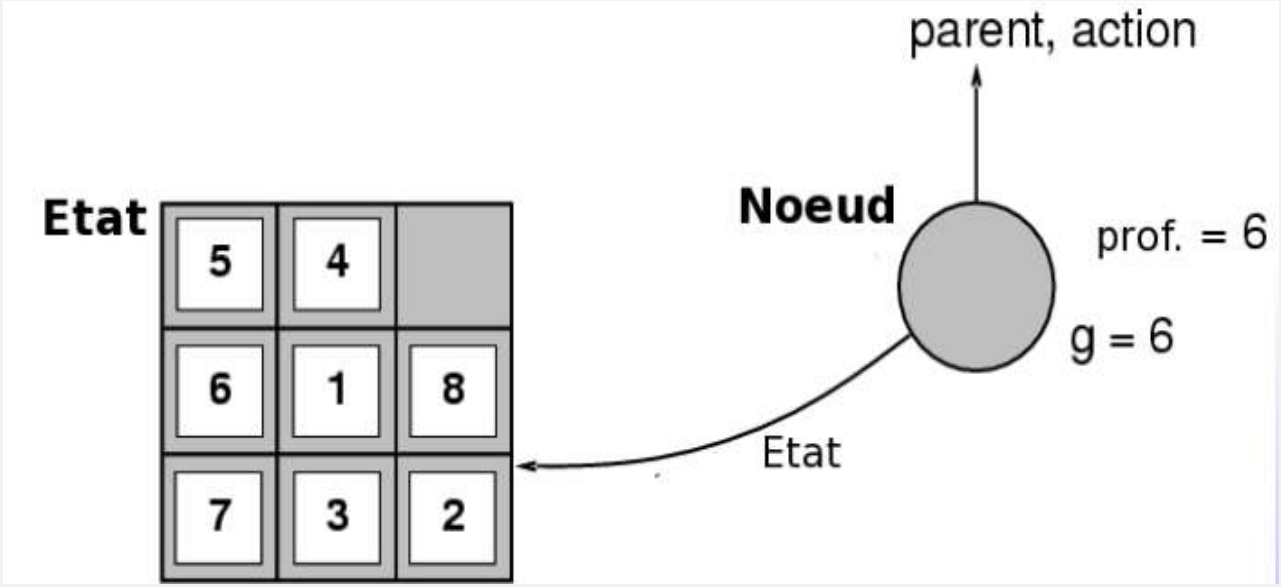
- Génération d'un **arbre de recherche**

On s'arrête lorsqu'on a choisi de développer un **nœud** qui est un état final.

IMPLÉMENTATION D'UN ALGORITHME DE RECHERCHE

- La structure de données **nœud** s qui est partie intégrante de l'arbre de recherche et qui inclue:

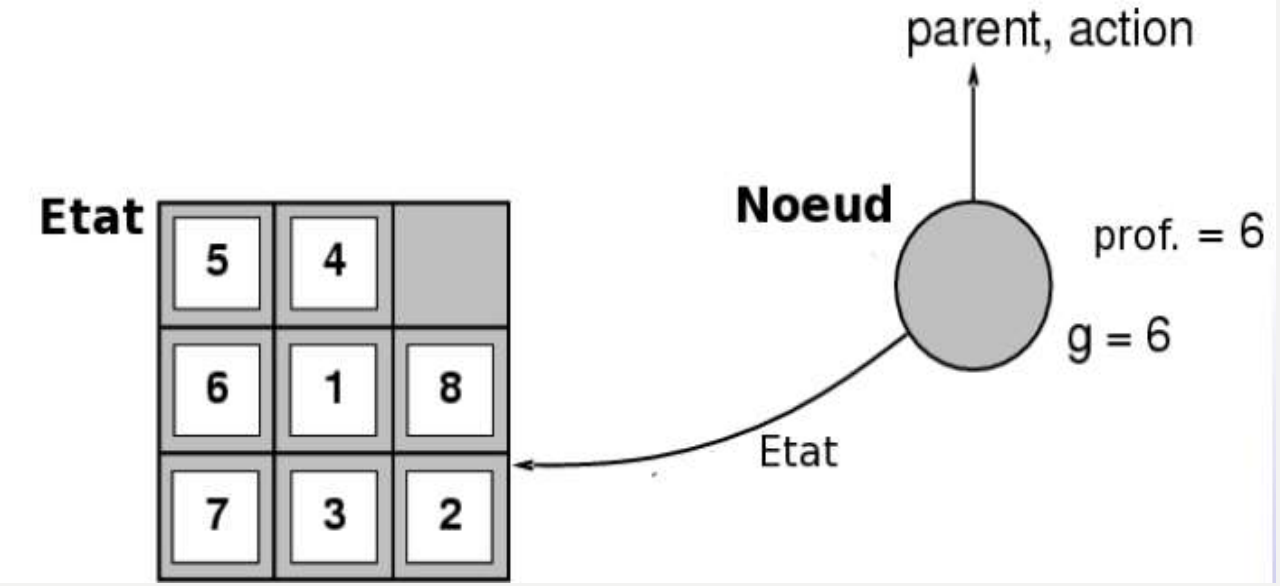
- L'état
- Le parent
- L'enfant
- La profondeur
- Le coût du chemin **(noté *g(x)*)**



IMPLÉMENTATION D'UN ALGORITHME DE RECHERCHE

Deux fonctions utiles :

- ***Expand*** crée de nouveaux nœuds
- ***Insert-Fn*** insère de nouveaux nœuds dans la liste à traiter



STRATÉGIES DE RECHERCHE

- Des algorithmes permettent de **guider la recherche** d'une solution en faisant des choix concernant les états à développer et en gérant le retour sur ces choix tout en évitant l'explosion combinatoire.

- **Explosion combinatoire :** la croissance exponentielle des possibilités ou des combinaisons qui doivent être prises en compte lors de la résolution d'un problème

STRUCTURE GÉNÉRALE D'UN ALGORITHME DE RÉSOLUTION

- **Point de départ :** on commence toujours par l'état initial (Situation initiale).
- **Déroulement :** on exécute les actions suivantes itérativement jusqu'à atteindre le but.
  - **S'il** n'y a plus d'états à traiter **alors** Renvoyer échec
  - **Sinon Choisir un des états à traiter**
    - **Si** l'état courant est un état but **alors** Renvoyer la solution correspondante
    - **Sinon**
      - Retirer cet état de l'ensemble des états à traiter et le remplacer par l'ensemble de ses états successeurs

UN ALGORITHME DE RECHERCHE GÉNÉRIQUE

```
fonction recherche(état initial, successeurs, test but, co

nœuds_ à_ traiter ← créer_liste(créer_noeud(état initial, [ ],

    boucle

si vide ?(nœuds_à_traiter) alors Renvoyer échec

nœud ← enlever premier_nœud (nœuds_à_traiter )

si test_but(état(nœud)) = vrai   alors renvoyer chemin(nœud), état(nœud))

        pour tout (action, état) dans successeurs(état(nœud))

    chemin ←  [action, chemin(nœud)]

    coût du chemin ← coût du chemin(nœud) + coût(ét

    s ← créer_nœud(état, chemin, coût_ du_ chemin)

    insérer(s, nœuds _à traiter)
```

Cet algorithme générique prend en entrée la description du problème à

initialiser une liste de nœuds à traiter

Si la liste de nœuds à traiter est vide → nous avons examiné tous les chemins

Si la liste contient encore des nœuds, nous sortons le premier nœud de la liste.

Si l'état de ce nœud est un état but. Renvoyer le chemin vers le but

Dans le cas contraire, la recherche se poursuit : nous produisons les successeurs du nœud et les insérons dans la liste de nœuds à traiter.

STRATÉGIE DE RECHERCHE

Qu'est ce qui change d'une stratégie de recherche à l'autre ?

- Les différents **attributs des nœuds** sont initialisés par la fonction **Expand**

- Une **stratégie de recherche** est définie par **l'ordre** dans lequel les nœuds sont développés, i.e. la fonction **Insert-Fn.**

EVALUATION DES ALGORITHMES DE RECHERCHE

Une stratégie s'évalue en fonction de 4 dimensions (Les métriques de comparaison des algorithmes):

- **Complétude** : est ce que cette stratégie trouve toujours une solution si elle existe ?
- **Complexité en temps** : le nombre de nœuds créés
  - nombre d'opérations nécessaires à son exécution dans le pire des cas.
- **Complexité en mémoire** : le nombre maximum de nœuds en mémoire
- **Optimalité** : est ce que la stratégie trouve toujours la solution la moins

EVALUATION DES ALGORITHMES DE RECHERCHE - SUITE

- La complexité en temps et en mémoire se mesure en termes de :
  - b : le facteur maximum de branchement de l'arbre de recherche,

    i.e. le nombre maximum de fils des nœuds de l'arbre de recherche.
  - d : la profondeur de la solution la moins coûteuse
  - m : la profondeur maximale de l'arbre de recherche

Attention m peut être ∞

ALGORITHMES DE RÉSOLUTION AVEUGLES (NON INFORMÉS)

Les stratégies de recherche non informées utilisent seulement les informations disponibles dans le problème.

- Recherche en largeur d'abord (Breadth-first search).
- Recherche en coût uniforme.
- Recheche en profondeur d'abord (Depth-first search).
- Recherche en profondeur limitée.
- Recherche en profondeur itérative(itérée).

RECHERCHE EN LARGEUR D'ABORD

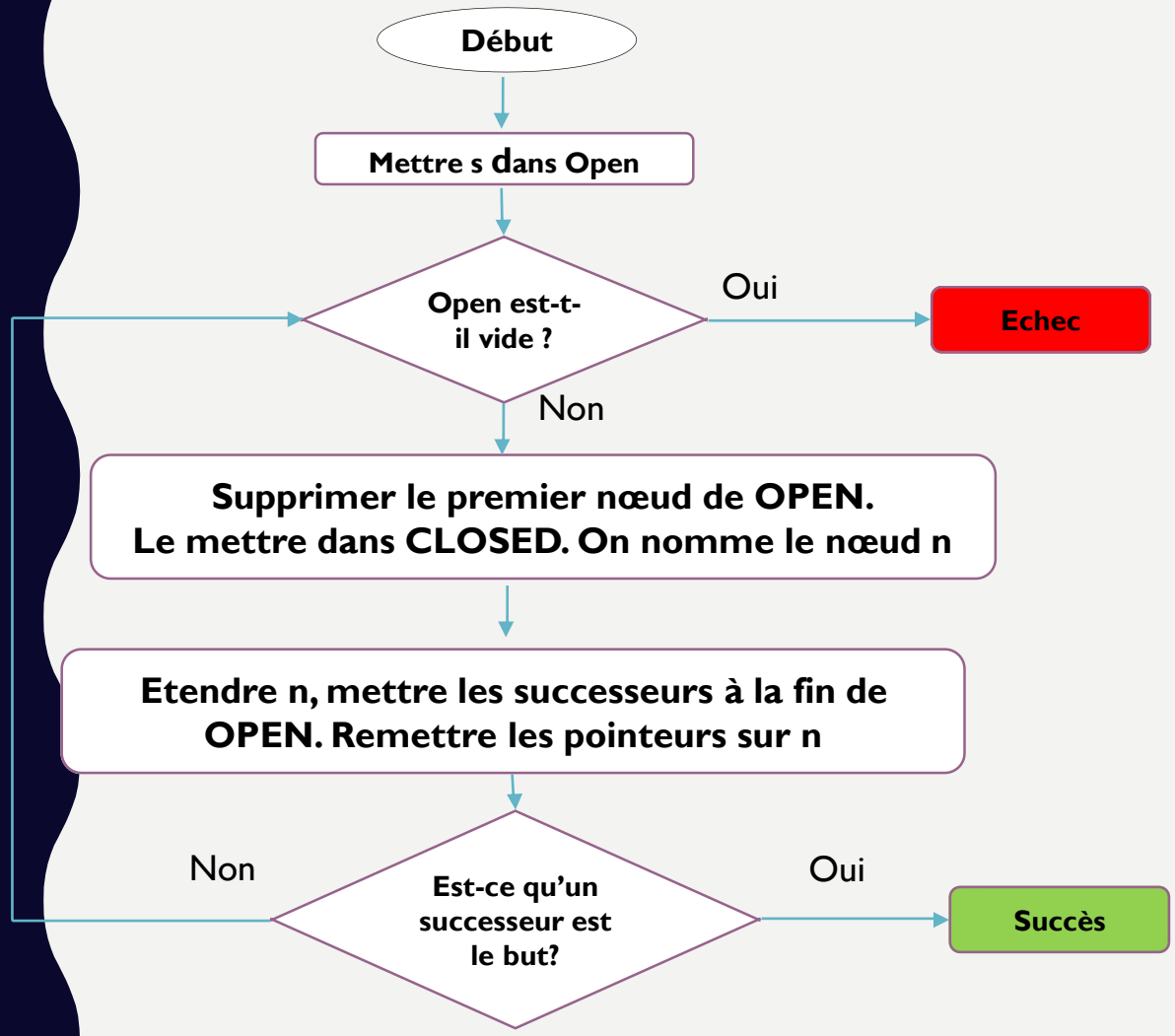
Principe

- Quand le système est dans un état donné, on engendre tous les états qui peuvent être obtenus à partir de cet état avec les opérateurs dont on dispose.
  - La fonction ***Insert-Fn*** ajoute les successeurs en fin de liste **(FIFO).**

- C'est un succès quand l'état engendré est l'état final recherché.

- Sinon on mémorise l'ensemble des états engendrés et on réitère pour chacun d'eux, tour à tour, le même procédé.

RECHERCHE EN LARGEUR D'ABORD



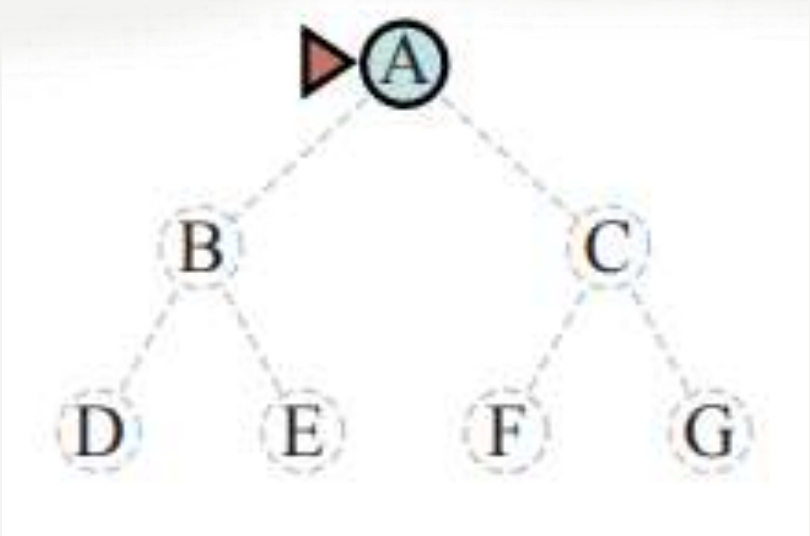
Algorithme

on a besoin de 2 listes:

- **OPEN** : représentant l'ensemble des états engendrés non traités.
- **CLOSED/VISITED** : sert à mémoriser l'ensemble des nœuds **déjà traités**.

RECHERCHE EN LARGEUR – EXEMPLE D'UN PARCOURS

- **Début:** OPEN={A}.



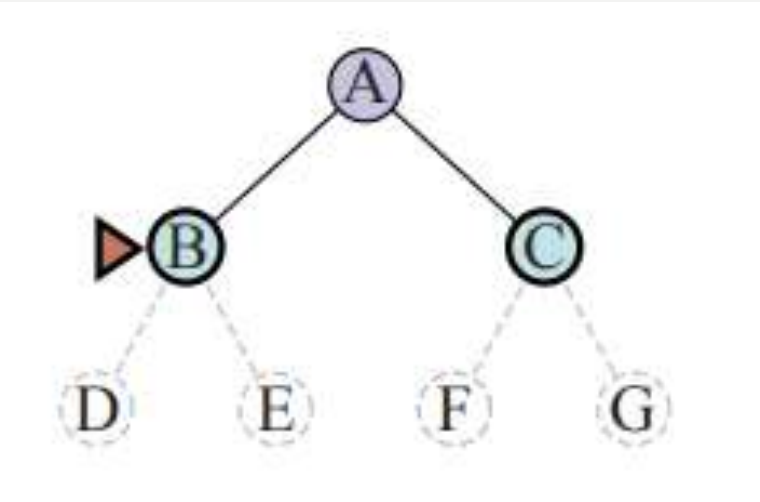
RECHERCHE EN LARGEUR – EXEMPLE D'UN PARCOURS

- **Etape 1 :**

  - OPEN={}, CLOSED={A}.

  - Etendre A. Successeurs = {B,C}

  -OPEN={B,C}.



RECHERCHE EN LARGEUR – EXEMPLE D'UN PARCOURS

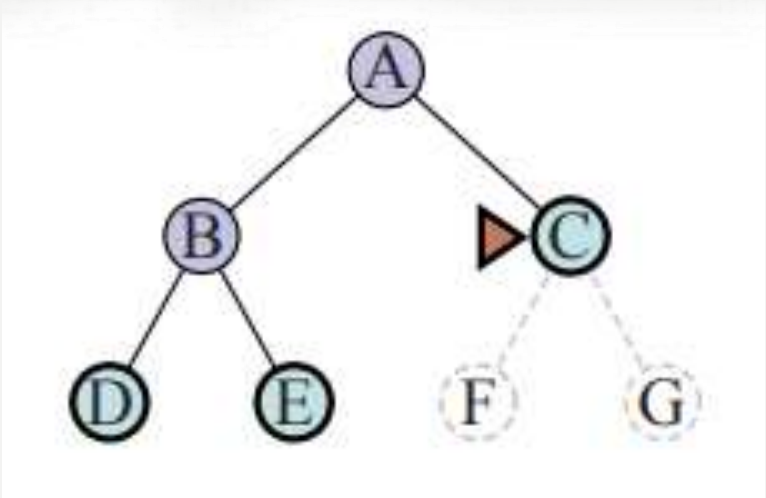
- **Etape 2 :**

  - OPEN={C}, CLOSED={A,B}.

  -Etendre B : successeurs = {D,E}

  -OPEN{C, D,E}.

  Important (FIFO : donc les nouveaux nœuds sont ajoutés à la fin de la liste)



RECHERCHE EN LARGEUR – EXEMPLE D'UN PARCOURS

- **Etape 3 :**

  - OPEN={D,E}, CLOSED={A,B,C}.

  -Etendre C : successeurs = {F,G}

  OPEN={D,E,F,G}.

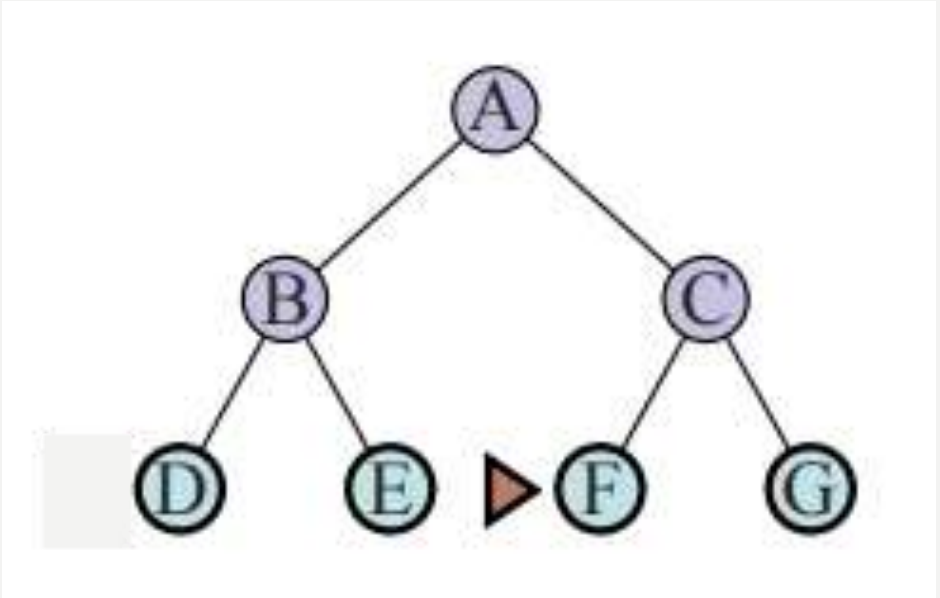
- **Etendre D. Pas de successeur !**

  Etendre E : Pas de successeur !

  Etendre F. Pas de successeur !

  Etendre G : Pas de successeur !

  OPEN={}, CLOSED={A,B,C,D,E,F,G}. Fin.



REMARQUE IMPORTANTE POUR L'ORDRE DE PARCOURS

- Si une règle ne définit pas les priorités des nœuds alors :
  - **Deux successeurs ont la même priorité** i.e. Nous pouvons prendre l'un ou l'autre sans altérer la recherche.
  - **Exemple** : Successeur de A (B,C) nous pouvons avoir l'ordre A-B-C ou A-C-B.

Par convention nous choisissons de gauche a droite.

PROPIÉTÉS DE LA RECHERCHE EN LARGEUR

- **Complet**, si *b* est fini ⇒ b : On suppose que chaque état a *b* successeurs
- **Complexité en temps :**

  $1 + b + b^2 + b^3 + \ldots + b^d = O(b^d)$ ⇒ d : la profondeur de la solution
- **Complexité en espace :** $O(b^d)$ (garde tous les nœuds en mémoire)
- **Optimale si coût = 1** pour chaque pas, mais non optimale dans le cas général

⇒ L'espace est le plus gros problème de la recherche en largeur.

RECHERCHE EN COÛT UNIFORME – ALGORITHME DE DJIKSTRA

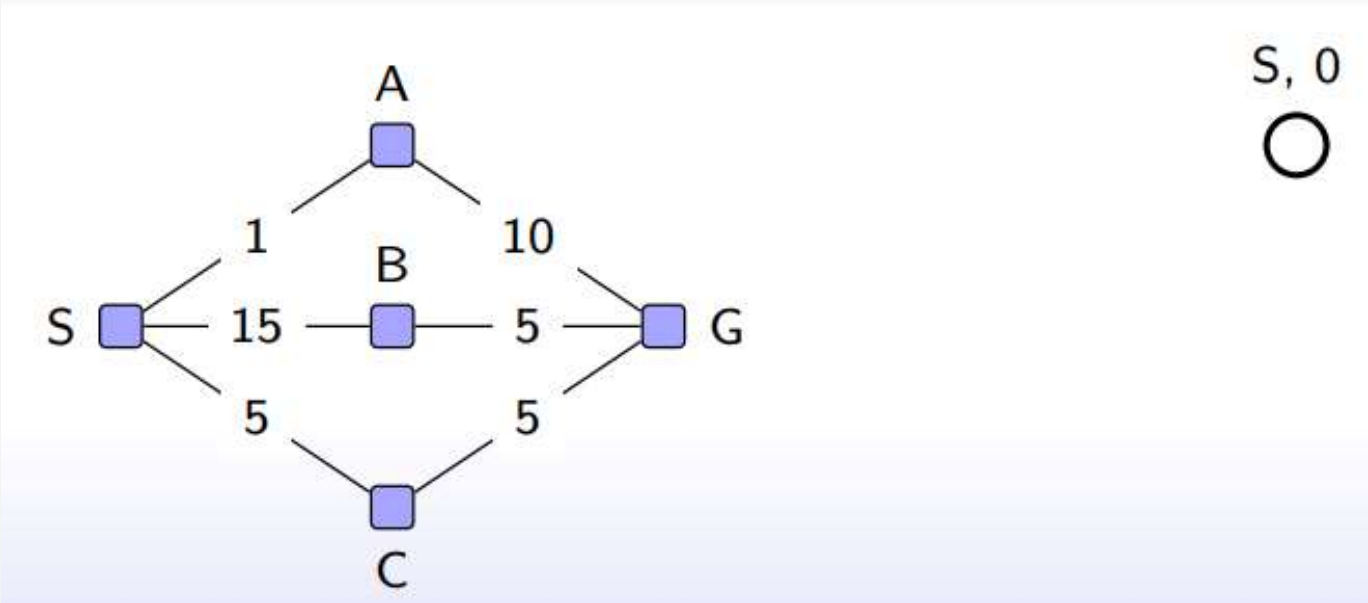
- Contrairement à la recherche en largeur, elle s'étend en vagues de chemins à coût uniforme.

- La fonction *Insert-Fn* ajoute les nœuds dans l'ordre du coût **g(x)** dans **OPEN** (Du plus petit coût au plus grand)**.**

g(x) : est le **coût** du chemin de **l'état initial** vers **x**.

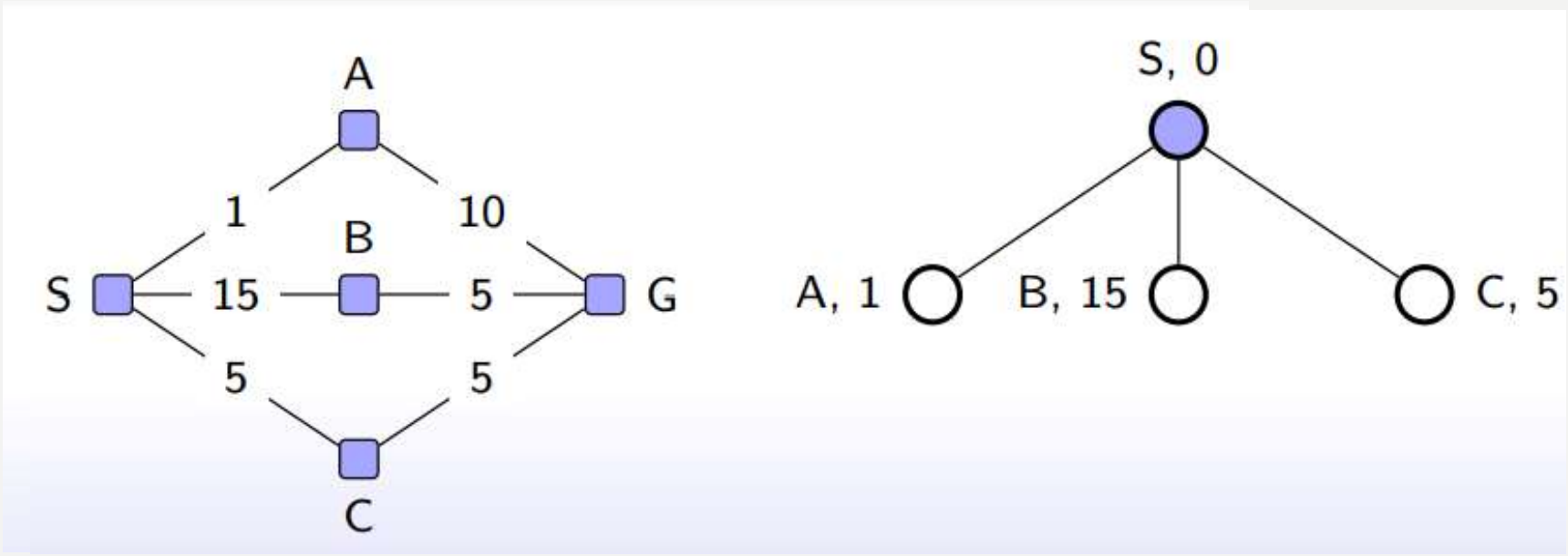
RECHERCHE EN COÛT UNIFORME-EXEMPLE

- **Etape 1 :** Open {S} Closed={}



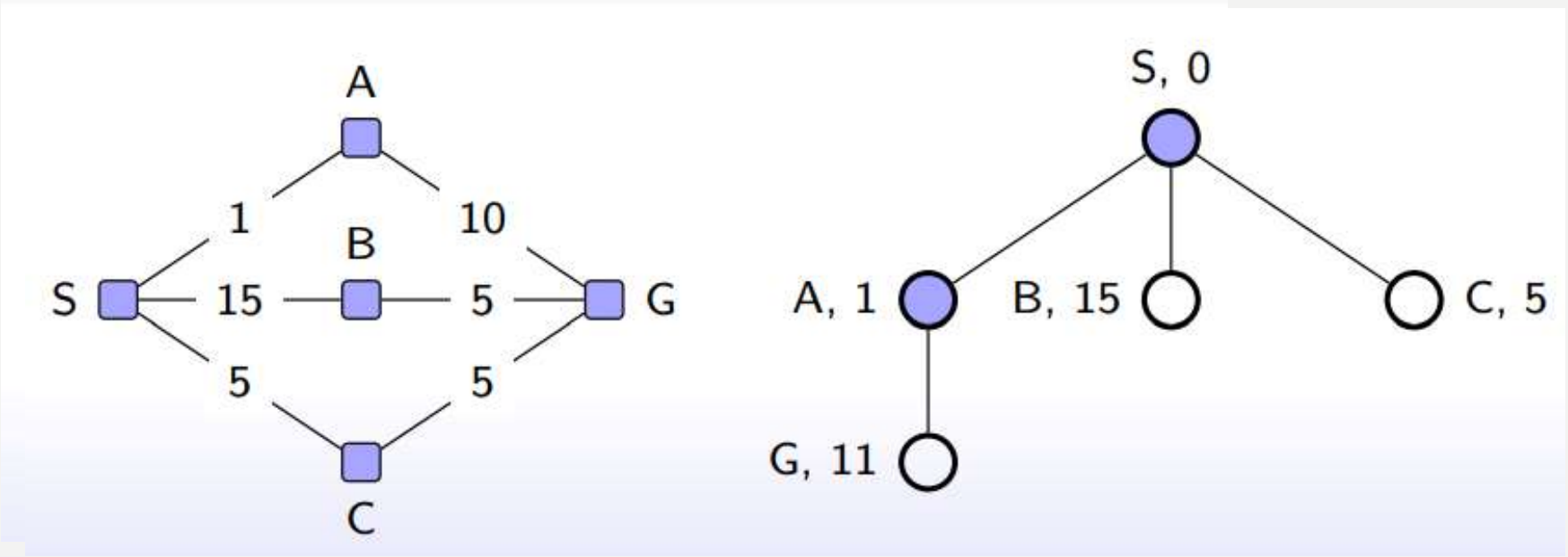
RECHERCHE EN COÛT UNIFORME-EXEMPLE

- **Etape 1 :** Open {S} Closed={}
- **Etape 2 :** Open{(A,1),(C,5),(B,15)},Closed={S}



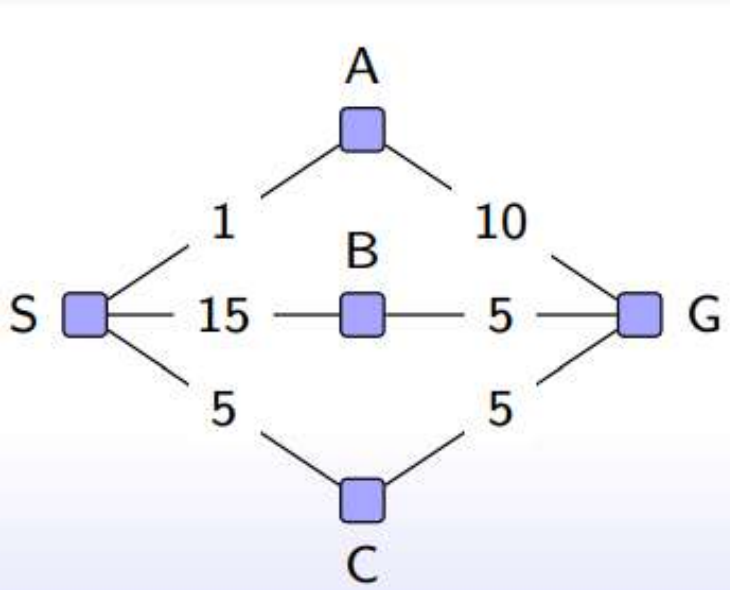
RECHERCHE EN COÛT UNIFORME-EXEMPLE

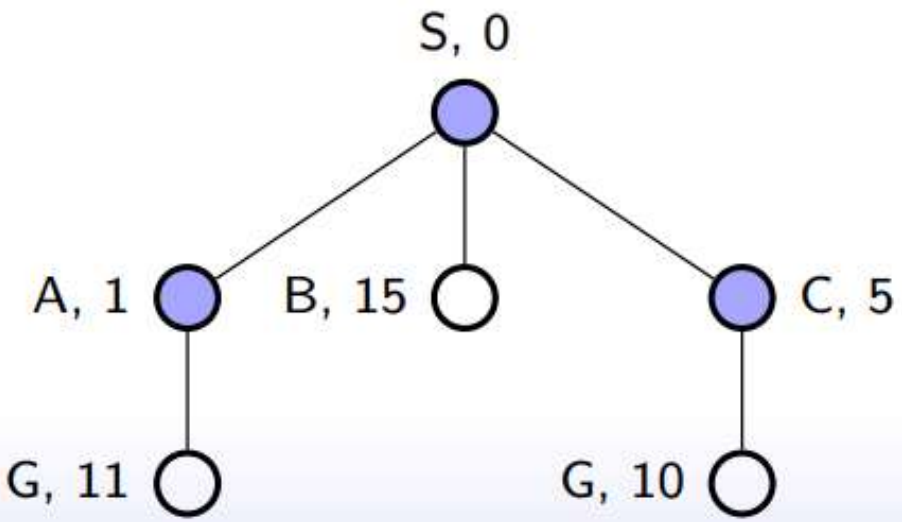
- **Etape 1 :** Open {S} Closed={}
- **Etape 2 :** Open{(A,1),(C,5),(B,15)},Closed={S}
- **Etape 3 :** Open{(C,5),(G,11),(B,15)},Closed={S,A}



RECHERCHE EN COÛT UNIFORME-EXEMPLE

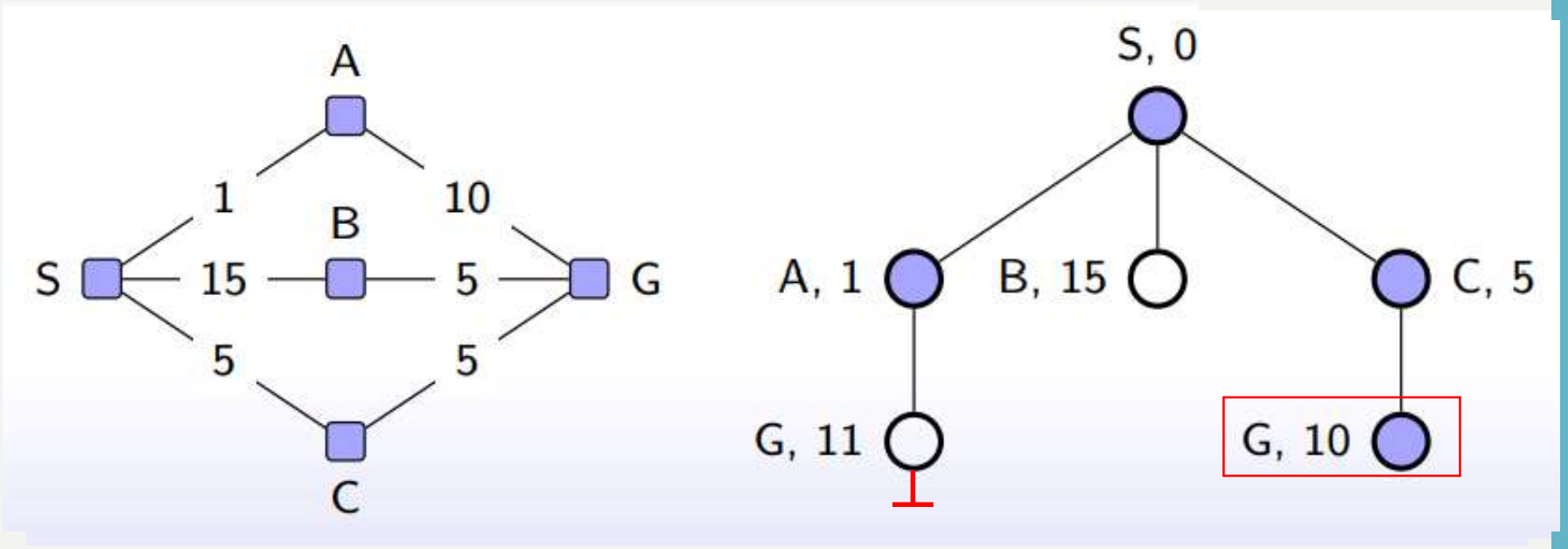
- **Etape 1 :** Open {S} Closed={}
- **Etape 2 :** Open{(A,1),(C,5),(B,15)},Closed={S}
- **Etape 3 :** Open{(C,5),(G,11),(B,15)},Closed={S,A}
- **Etape 4 :** Open{(G,10),(G,11),(B,15)},Closed={S,A,C}





RECHERCHE EN COÛT UNIFORME-EXEMPLE

- **Etape 1 :** Open {S} Closed={}
- **Etape 2 :** Open{(A,1),(C,5),(B,15)},Closed={S}
- **Etape 3 :** Open{(C,5),(G,11),(B,15)},Closed={S,A}
- **Etape 4 :** Open{(G,10),(G,11),(B,15)},Closed={S,A,C}
- **Etape 5 :** Closed={S,A,C,G}
- **La solution :** S,C,G. **L'ordre de parcours :** S,A,C,G **Closed (dernière étape)**



RECHERCHE EN COÛT OPTIMAL - PROPRIÉTÉS

- Equivalent à la largeur d'abord si le coût est toujours le même
- **Complet** si le coût de chaque pas est strictement supérieur à 0.

- **Complexité en temps** : $O(b^{1+\lfloor C^*/\varepsilon \rfloor})$: nombre de nœuds pour lesquels $g \leq C^*$

  où C^* est le coût de la solution optimale.

- **Complexité en espace** : idem que la complexité en temps

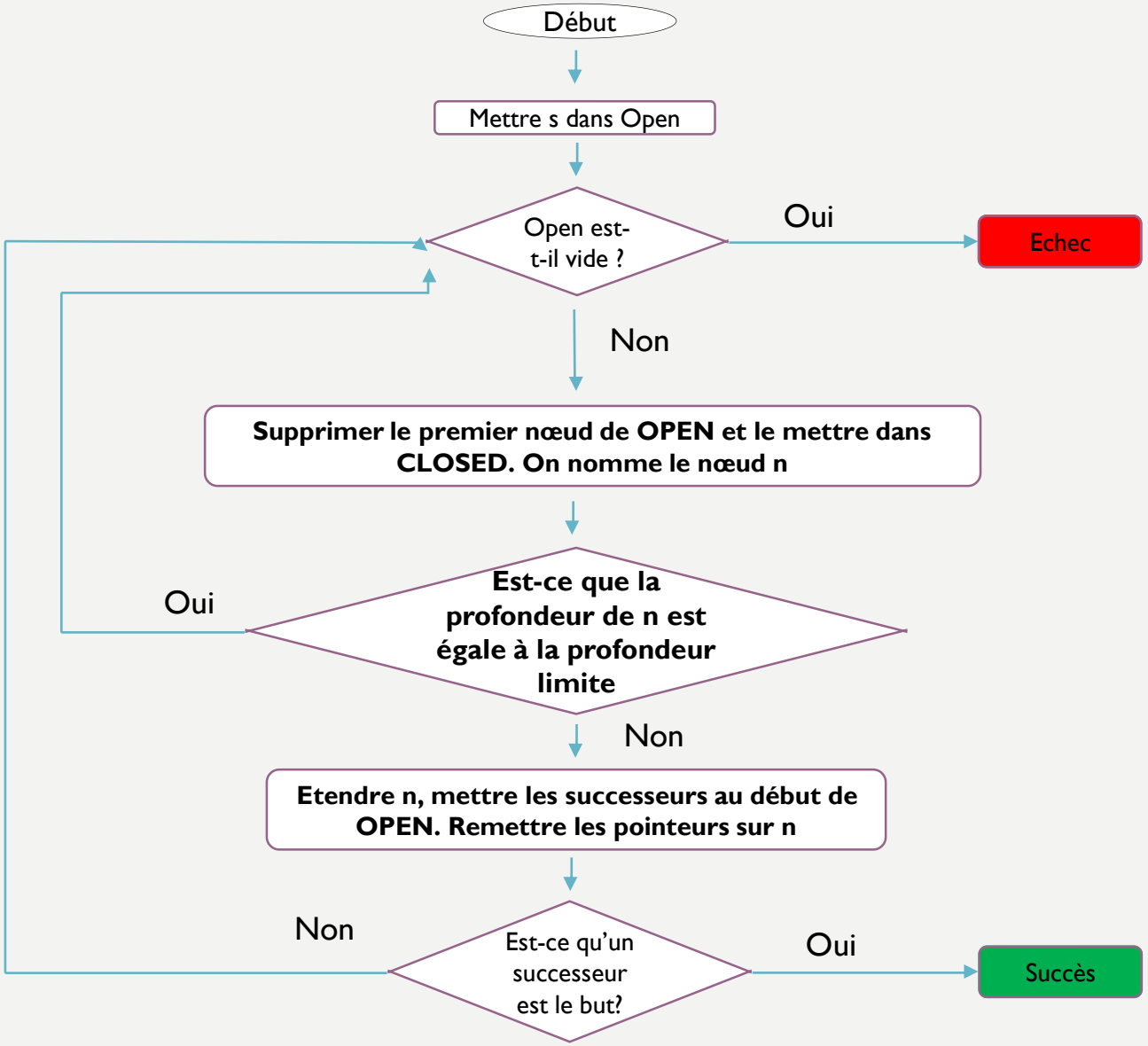
- **Optimale** car les nœuds sont développés en fonction de g.

RECHERCHE EN PROFONDEUR D'ABORD

- Le système étant dans un état donné, Il faut engendrer tous les états pouvant en être issus.

- La fonction *Insert-Fn* ajoute les successeurs en début de liste **OPEN (LIFO).**

RECHERCHE EN PROFONDEUR D'ABORD



RECHERCHE EN PROFONDEUR D'ABORD

- **Etape 1 :** OPEN = {A} CLOSED = {}
- **Etape 2 :** OPEN = {B,C} CLOSED = {A}
- **Etape 3 :** OPEN = {D,E,C} CLOSED = {A,B}
- **Etape 4 :** OPEN = {H,I,E,C} CLOSED = {A,B,D}
- **Etape 5 :** OPEN = {E,C} CLOSED = {A,B,D,H,I}
- **Etape 6 :** OPEN = {J,K,C}

CLOSED = {A,B,D,H,I,E}

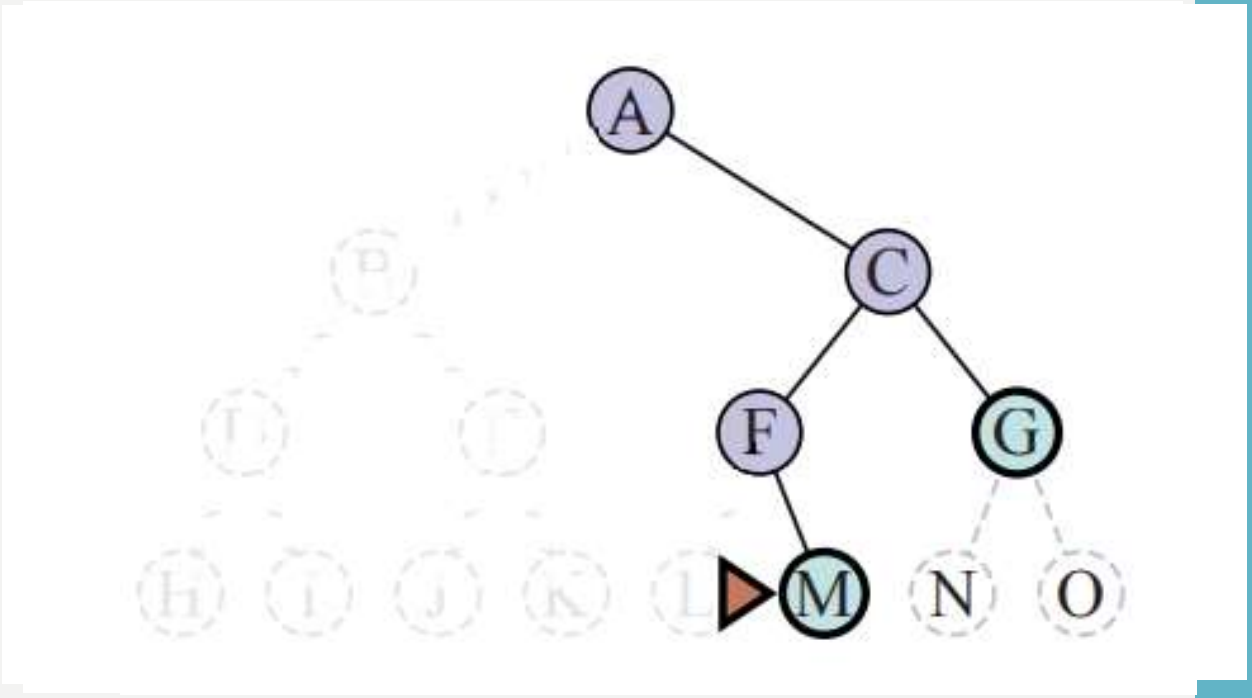
- **Etape 7** : OPEN = {C}

CLOSED = {A,B,D,H,I,E,J,K}

- **Etape 8 :** OPEN = {F,G}

CLOSED = {A,B,D,H,I,E,J,K,C}

- **Etape 9 :** OPEN = {L,M,G} CLOSED = {A,B,D,H,I,E,J,K,C,F}
- **Etape 10 :** OPEN = {M,G} CLOSED = {A,B,D,H,I,E,J,K,C,F,L}



Solution : A,C,F,M

Ordre de parcours : A,B,D,H,I,E,J,K,C,F,L,M

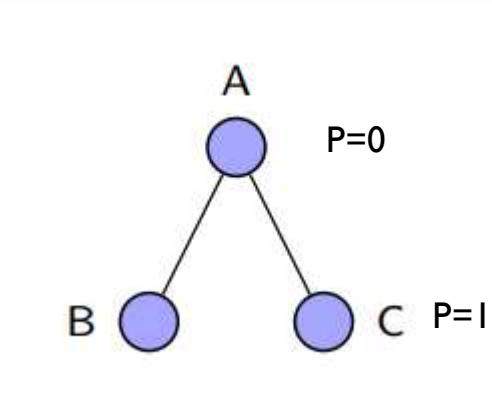
RECHERCHE EN PROFONDEUR : PROPRIÉTÉS

- **Non complet** dans les espaces d'états infinis ou avec boucle
- Il est possible d'ajouter un test pour détecter les répétitions
- **Complexité en temps** : $O(b^m) \Rightarrow$ m : la profondeur maximale de l'arbre de recherche
  - Très mauvais si m est beaucoup plus grand que b
- **Complexité en espace** : O(bm)
  - Linéaire!
- **Non optimale**

RECHERCHE EN PROFONDEUR LIMITÉE

- Algorithme de recherche en profondeur d'abord, mais avec une limite ℓ sur la profondeur.
- Les nœuds de profondeur ℓ n'ont pas de successeurs.

- Exemple : ℓ=1.



PROPRIÉTÉS

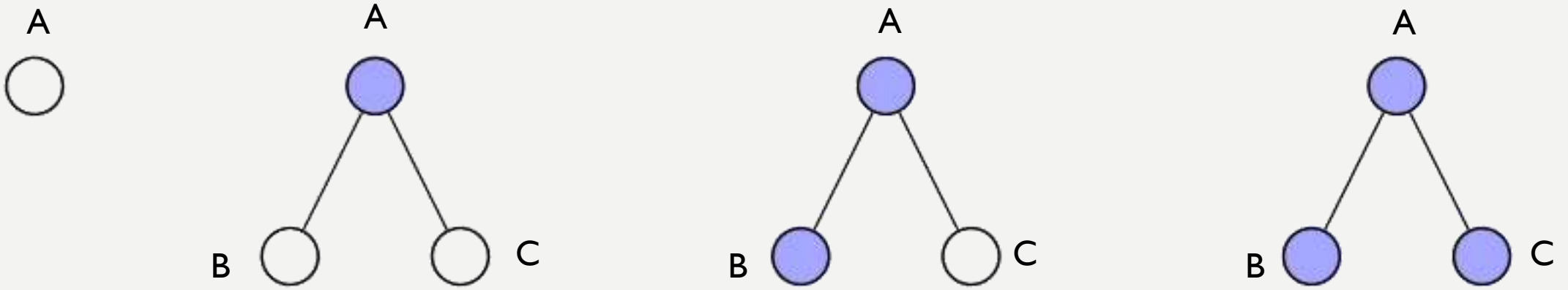
- **Complet** si $l \geq d$ d : la profondeur de la solution la moins coûteuse.
- Complexité en temps : $O(b^l)$
- Complexité en espace : O(bl)
- **Non optimale**

RECHERCHE EN PROFONDEUR ITÉRATIVE.

- Profondeur limitée, mais en essayant toutes les profondeurs: 0, 1, 2, 3, . . .
- Evite le problème de trouver une limite pour la recherche profondeur limitée
- Combine les avantages de largeur d'abord (complète et optimale), mais a la complexité en espace de profondeur d'abord.

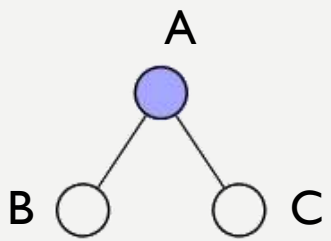
RECHERCHE EN PROFONDEUR ITÉRATIVE

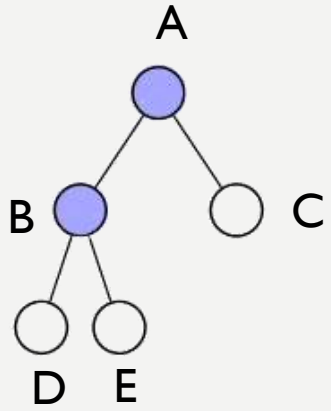
Itération : l=1

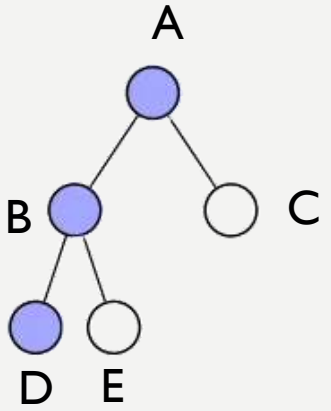


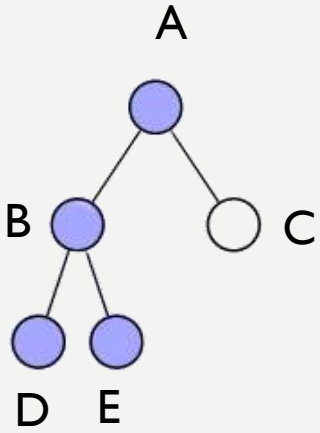
RECHERCHE EN PROFONDEUR ITÉRATIVE

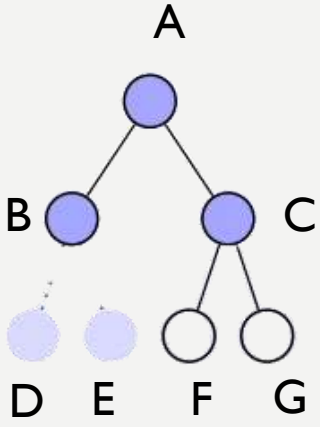
Itération : l=2

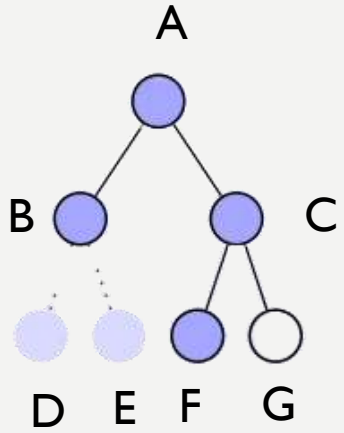


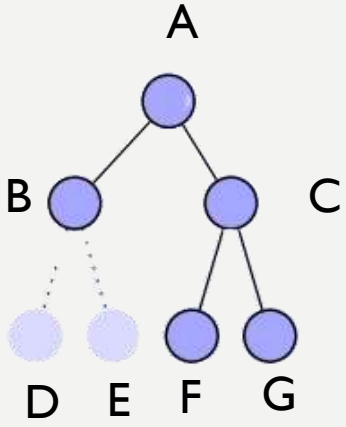












RECHERCHE EN PROFONDEUR ITÉRATIVE

- Peut paraître du gaspillage car beaucoup de nœuds sont étendus de multiples fois.
- Mais la plupart des nouveaux nœuds étant au niveau le plus bas, ce n'est pas important d'étendre plusieurs fois les nœuds des niveaux supérieurs
- **Complet**
- **Complexité en temps :**

  $(d + 1)b^0 + db^1 + (d - 1)b^2 + \ldots + b^d = O(b^d)$ -d : la profondeur de la solution la moins coûteuse
- **Complexité en espace** : O(bd)
- **Optimale** : oui, si le coût de chaque action est de 1. Peut être modifiée pour une stratégie de coût uniforme

Critères	Largeur d'abord	Coût uniforme	Prof. d'abord	Prof. limitée	Prof. itérative
Complétude	Oui	Oui	Non	Oui si $l \geq d$	Oui
Temps	$O(b^d)$	$O(b^{\lceil C^*/\epsilon \rceil})$	$O(b^m)$	$O(b^l)$	$O(b^d)$
Espace	$O(b^d)$	$O(b^{\lceil C^*/\epsilon \rceil})$	$O(bm)$	$O(bl)$	$O(bd)$
Optimalité - coût d'une action $= 1$	Oui	Oui	Non	Non	Oui
Optimalité - cas général	Non	Oui	Non	Non	Non

ETATS REDONDANTS

- Les algorithmes que nous venons de présenter peuvent visiter plusieurs fois le même état pendant la même recherche. Il y a **deux types** de **redondances** possibles :

1. Il y a des états qui peuvent être visités deux fois sur le même chemin (par exemple, au taquin, si vous jouez droite puis gauche, vous revenez à l'état initial).

   - **Solution :** Il suffit d'examiner au fur et à mesure les chemins des nœuds à insérer et de ne jamais ajouter à la liste de nœuds à traiter des nœuds dont les chemins contiennent plusieurs fois le même état. Il est possible de modifier les différents algorithmes de telle façon que cette modification ne change pas significativement la complexité en espace.

ETATS REDONDANTS

2. Il peut y avoir plusieurs chemins différent qui amènent au même état.

- **Important :** se rappeler de tous les états déjà visités et de ne garder qu'un seul chemin par état.
  - Il faut gérer une liste des états déjà visités et y ajouter les états quand ils sont visités pour la première fois. Avant de générer les successeurs d'un nœud, il faut vérifier si l'état successeur est dans la liste des états déjà visités (et si c'est le cas, nous ne générons pas de successeurs).
- **Complexité :**
  - **En largeur :** Cette modification ne changera pas trop la complexité en espace pour le parcours en largeur (qui souffre déjà d'une haute complexité en espace).
  - **Parcours en profondeur (itérée) :** qui gardent normalement peu de nœuds en mémoire cette modification va augmenter très significativement la complexité en espace.

EXERCICE 1

- Donner un état initial, un test de l'état final, une fonction successeur et une fonction de coût (c'est-à-dire le coût d'un chemin) à chacun des problèmes suivants. Choisissez une formulation suffisamment précise pour qu'elle puisse être implémentée sur machine.

1. Vous devez colorier une carte en utilisant seulement quatre couleurs de sorte que deux régions adjacentes n'aient pas la même couleur.

2. Un programme informatique affiche le message « enregistrement illégal » en entrée lorsqu'on spécifie un fichier contenant des enregistrements. Vous savez que les traitements des enregistrements sont indépendants les uns des autres. Vous voulez savoir lequel des enregistrements pose problème.

EXERCICE 2

Considérez un espace de recherche dans lequel l'état initial est 1 et la fonction successeur pour un nœud n retourne deux états contenant les entiers 2n et 2n + 1.

1. Dessiner la partie de l'espace de recherche contenant les nœuds de 1 à 15
2. Supposer que le but soit 11. Donner l'ordre de parcours des nœuds pour les algorithmes :
    (a) Largeur d'abord
    (b) Profondeur d'abord
    (c) Profondeur d'abord limitée à 2
    (d) Profondeur itérative