Intelligence Artificielle

Mounir T. El Araki

mounir.elarakitantaoui@uic.ac.ma

Slides traduit de **MITOpenCourseware**

Intelligence Artificielle

- Introduction à l'intelligence artificielle
- Historique de l'IA
- Problèmes de recherche
  - Recherche dans les graphes
  - CSP (Problèmes de Satisfaction de Contraintes)
- LISP (LISt Processing) Scheme avec Racket
- Représentation de la connaissance (Knowledge) et Inférence
  - Logique propositionnelle et 1ière degré
  - Système basé sur les règles
- Introduction à la Planification

Organisation du cours

- 2 heures (**Lundi 14h00 à 15h50**)
- 11 Séances
- 1 contrôle continu (CC) (contenu partiel)
- 1 Examen final (EF) (tout le contenu)
- (CC)*40% + EF*60%

Contenu

S1	**18-févr.**	Introduction à l'IA et Historique
S2	**25-févr.**	Problématique de Recherche
S3	**4-mars**	Problématique de Recherche Aveugle
S4	**11-mars**	Problématique de Recherche informé et A*
S5	**18-mars**	Problèmes de Satisfaction de Contraintes
S6	**25-mars**	Travaux Dirigées
S7	**1-avr.**	Pas de cours
S8	**8-avr.**	Pas de cours
S9	**15-avr.**	Contrôle continu
S10	**22-avr.**	Problématique des Jeux (Min-Max)
S11	**29-avr.**	Introduction à la logique formelle
S12	**6-mai**	Logique Propositionnelle (Exercices)
S13	**13-mai**	Logique de premier ordre et Introduction à la Planification

Intelligence Artificielle

- Introduction à l'intelligence artificielle
- Historique de l'IA
- Problèmes de recherche
  - Recherche dans les graphes
  - CSP (Problèmes de Satisfaction de Contraintes)
- LISP (LISt Processing) Scheme avec Racket
- Représentation de la connaissance (Knowledge) et Inférence
  - Logique propositionnelle et 1ière degré
  - Système basé sur les règles
- Introduction à la planification

Introduction (définitions)

- Apprendre aux ordinateurs à être plus intelligents permettra sans doute d'apprendre à l'homme à être plus intelligent
  - (Patrick Winston : MIT AI Lab Dir.)
- L'IA est l'étude des idées qui permettent aux ordinateurs d'être intelligents
  - (Patrick Winston)
- L'IA est l'étude des facultés mentales à l'aide de modèles de type calculatoire
- L'IA a pour but de faire exécuter par l'ordinateur des tâches pour lesquelles l'homme dans un contexte donné est aujourd'hui meilleur que la machine

Introduction (définitions)

- L'IA est une **méthodologie** qui doit permettre de rendre les ordinateurs plus intelligents de façon à ce qu'ils montrent des caractéristiques normalement associées à l'intelligence dans les comportements humains, c'est-à-dire la compréhension du langage, l'apprentissage, la résolution de problèmes et le raisonnement (Stanford Knowledge System Lab. Dir :**Feigenbaum**)

- L'Intelligence Artificielle concerne la **conception** d'un être artificiel (machine) capable de posséder ou d'exhiber les capacités et caractéristiques propres à un cerveau humain

- Apprendre aux machines à **penser**

Types d'approches de l'IA

- IA Forte (approche cognitive: Strong AI)
  - La machine doit raisonner de la même façon que l'homme (utiliser les mêmes mécanismes de fonctionnement)
- IA Faible (approche pragmatiste: Weak AI)
  - La machine doit aboutir aux mêmes solutions que l'homme (peu importe la méthode employée)
  - Définitions difficiles , Qu'est ce que l'intelligence ?

L'intelligence humaine (Activités cognitives)

- Apprendre
  - élaborer un système de connaissances et pouvoir intégrer de nouvelles connaissances
- Raisonner, comprendre, déduire, anticiper, mémoriser
  - à partir du système de connaissances et des données de l'expérience pouvoir produire de nouvelles connaissances
- Communiquer et planifier
- Posséder une histoire
- Posséder une conscience
- Posséder des sentiments

Définitions

- L'IA est une branche de l'informatique et discipline des sciences cognitives
  - Mathématiques
  - Philosophie
  - Psychologie
  - Linguistique

- Comment l'IA procède-t-elle?
  - Modélisation et description des activités cognitives de manière abstraites
  - Réalisation de système artificiel répondant au model abstrait

Lien

- https://www.youtube.com/watch?v=RpZEqbZQUcA
- https://www.youtube.com/watch?v=hutQlyF750s
- https://www.youtube.com/watch?v=iARAMmS_43Y

Intelligence Artificielle

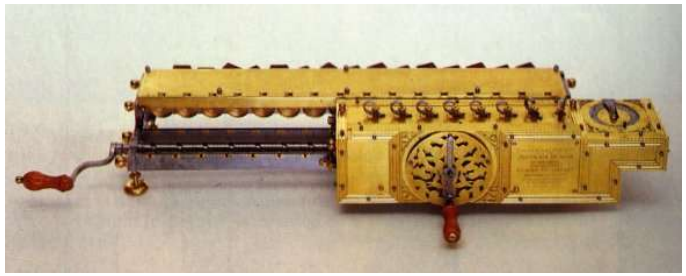
- Introduction à l'intelligence artificielle
- Historique de l'IA
- Problèmes de recherche
  - Recherche dans les graphes
  - CSP (Problèmes de Satisfaction de Contraintes)
- LISP (LISt Processing) Scheme avec Racket
- Représentation de la connaissance (Knowledge) et Inférence
  - Logique propositionnelle et 1ière degré
  - Système basé sur les règles
- Introduction à la planification

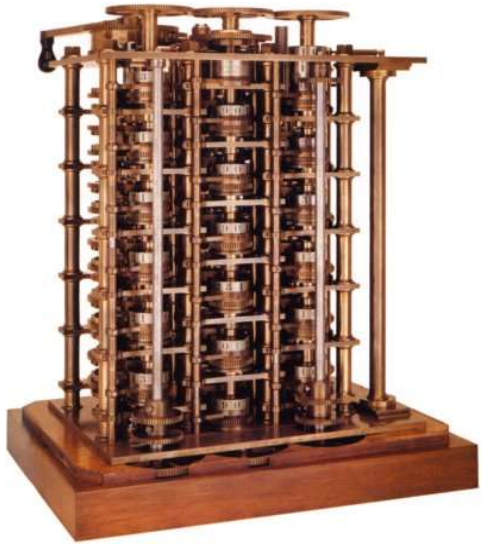
Historique (les débuts)

- Dans la mythologie grecque
  - Héphaïstos (fils de Zeus) a créer pour son service personnel deux servantes en or agissant comme des êtres vivants
- Le Golem dans la tradition juive (esclave puis héros)
  - Seigneur des Anneaux
  - Frankenstein

Historique (Mécanisation du calcul)

- Automatisation
  - 1623 Shickard
  - 1642 Pascal (la Pascaline)
  - 1670 Leibnitz
  - 1728 Falcon (premier cartes perforées)
  - Automates de Vancauson
  - 1805 Jacquard
  - Charles Babbage
    - 1830 Machine analytique





De l'automate à la machine Analytique de Babbage (mécanisation du calcul)

- Jacques Vancauson 1709 – 1782
  - 1738 le joueur de flûte traversière
  - 1739, joueur de tambourin et de flageolet, Canard digérateur
  - 1746, métier à tisser automatique
- Entre 1834 et 1836 Babbage définit les principaux concepts qui préfigurent ceux des ordinateurs :
  - Dispositif d'entrée / sorties (clavier, moniteur)
  - Organe de commande pour la gestion de transfert des nombres (unité de commande)
  - Magasin pour le stockage des résultats intermédiaires (registres) ou finaux (mémoire)
  - Moulin chargé d'exécuter les opérations (unité arithmétique et logique)
  - Dispositif d'impression (imprimante)

Historique (mécanisation du calcul)

- Le calcul passe par différentes étapes technologiques
  - Mécanique (engrenages)
  - Electrique (diode)
  - Electro-mécanique (relais)
  - Electronique (Transistor)

- Naissance de deux thèses paradoxales

  - Thèse 1 : le calcul ne fait pas partie de l'intelligence,
    - Intelligence Artificielle impossible

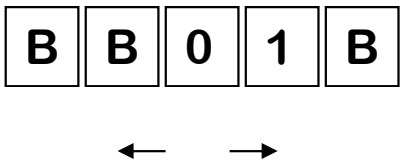
  - Thèse 2 : recréer des comportements humain (calcul) / animal (gestuelle)
    - Intelligence Artificielle possible

- Naissance d'une rivalité entre partisans et adversaires de l'IA
  - Minsky vs. Dreyfus

Historique (calculabilité)

- Notion de calculabilité
  - Fonction λ-calculable (Kleene, Church)
  - Fonction récursivement calculable (Gödel)
  - Thèse de Church-Turing
    - Fonction Turing-calculable $\Rightarrow$ machine de TURING

- Machine de Turing
  - Structure de stockage (bande linéaire) $\Gamma = \{ B, s_1, \ldots, s_n \}$
  - États $z = \{ z_0, \ldots, z_m, z_h \}$
  - Fonction de transition $\delta : (z - \{zh\}) \times \Gamma \rightarrow (z \times \Gamma \times \{G,D,I\})$

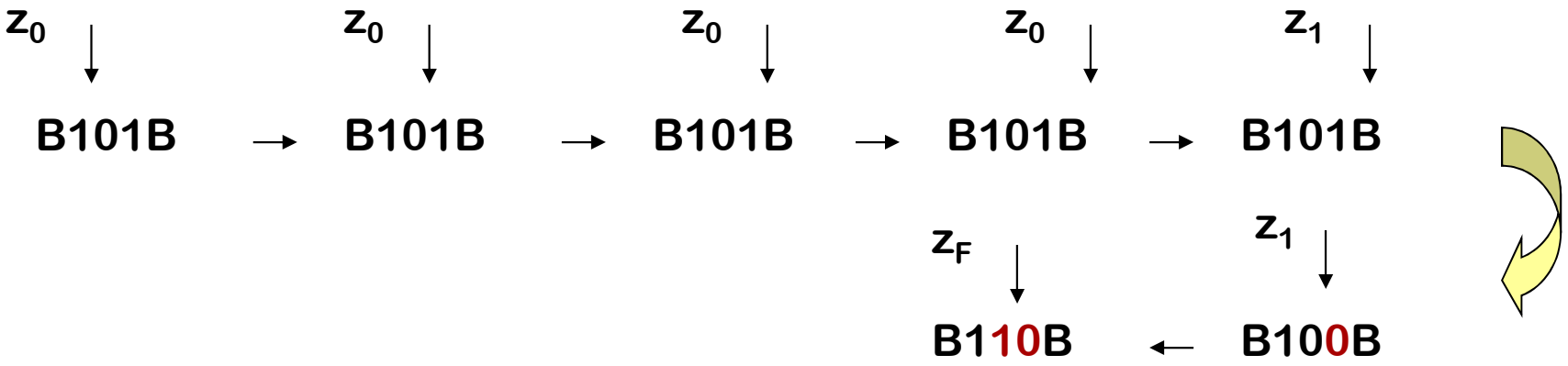


Historique (calculabilité)

- Machine de Turing : Ajouter 1

Γ = {0,1,B}

	B	0	1
z_0	(z_1, B, G)	$(z_0, 0, D)$	$(z_0, 1, D)$
z_1	$(z_F, 1, I)$	$(z_F, 1, I)$	$(z_1, 0, G)$



Historique (calculabilité)

- Machine de Turing : Détecter si le nombre de lettre dans un mot est paire.

	Blanc	A
z_0	(z_F, Blanc, D)	(z_1, A, D)
z_1	(z_0, A, D)	

Autre exemple L = ‘ $0^n 1^n$’

	0	I	X	Y	B
Z_0	(Z_1, X,D)			(Z_3,Y,D)	(Zf, B, G)
Z_1	(Z_1, 0,D)	(Z_2,Y, G)		(Z_1,Y ,D)	
Z_2	(Z_2, 0,G)		(Z_0, X,D)	(Z_2,Y, G)	
Z_3				(Z_3,Y,D)	(Zf, B,G)
Z_f					

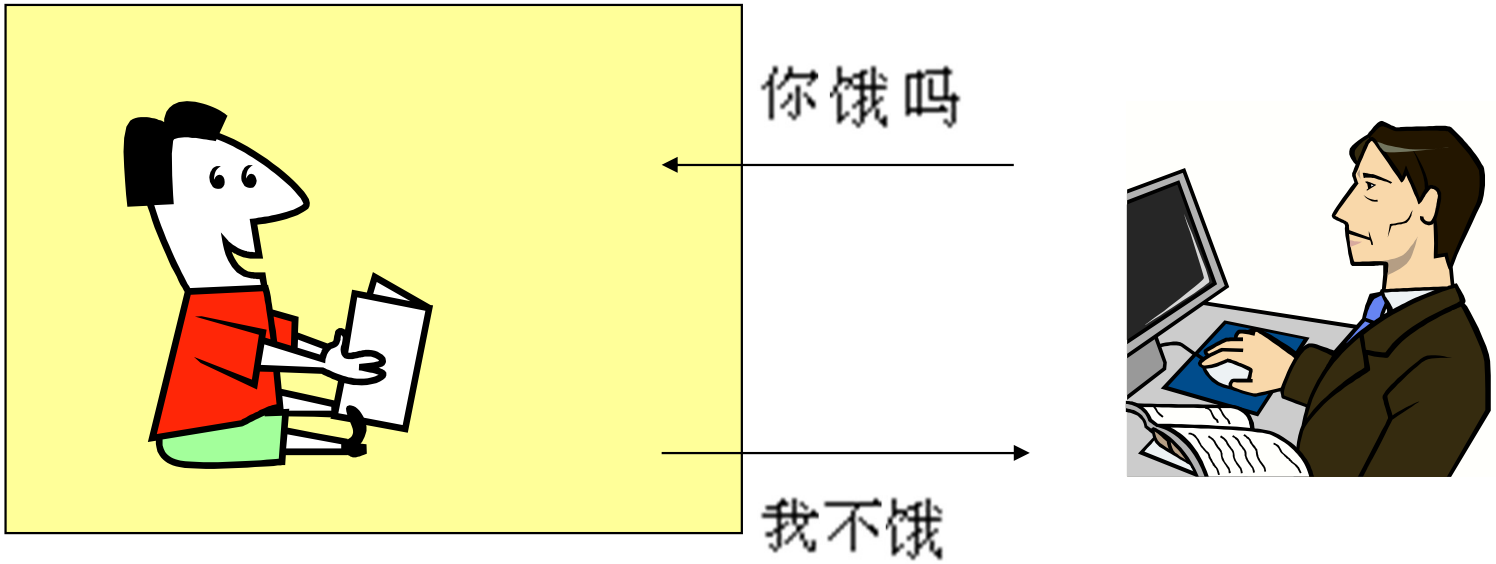
Historique (calculabilité)

- Le test de Turing
  - Un ordinateur peut-il tromper un humain ?
  - Une personne (A ou B) et une machine (A ou B) sont interrogées par une personne C
  - C doit déterminer qui de A ou B est la machine
  - ELIZA (Natural Language Processing)
    - http://www.manifestation.com/neurotoys/eliza.php3
  - **A.L.I.C.E. (Artificial Linguistic Internet Computer Entity)**
    - http://www.alicebot.org/

Historique (calculabilité)

John Searle (s'oppose à Turing)

La syntaxe est insuffisante pour produire le sens



Preuve de l'approche pragmatique de l'IA (Weak IA)

Historique (Fondements)

- Calcul
  - 1945 ENIAC (Mauchly, Eckert, Von Neumann)
  - 1948 Invention du transistor (Brattain, Bardeen et Shockley)
  - 1958 Invention du Circuit intégré (Kilby, TI)

- Naissance
- Débuts pendant la 2nde guerre mondiale
  - décryptage → traduction
  - Mise au point d'un traducteur automatique en 5 ans

- 1956 John McCarthy, Minsky, Newell, Simon
  - Automatiser le calcul et le **raisonnement**

- 1956 Newell, Shaw et Simon premier programme pour démontrer les théorèmes de la logique formelle
  - LOGIC THEORIST
  - GPS (General Problem Solver) 1959
  - NSS (programme de jeu d'échec)

- 1960 Création de **LISP** (McCarthy)

- 1974 Système Expert **Mycin** pour les diagnostiques médicaux

Domaines d'application

- Actuellement l'IA concerne :
  - La résolution de problèmes en général
    - Algorithme A*, recherche arborescente, CSP, heuristique, recherche locale, calcul des propositions et des prédicats, programmation génétique
  - La reconnaissance de formes / son (computer vision / Speech recognition)
  - Le traitement automatique du langage naturel (TALN) (NLP)
  - La robotique
  - Les réseaux de neurones
  - La recherche dans les bases de données
  - L'aide à la décision
  - Les jeux de stratégie
  - La démonstration des théorèmes
  - Les systèmes experts (diagnostic médical, de panne, etc …)
  - Les systèmes multi-agents

Intelligence Artificielle

- Introduction à l'intelligence artificielle
- Historique de l'IA
- **Problèmes de recherche**
  - Recherche dans les graphes
  - CSP (Problèmes de Satisfaction de Contraintes)
- LISP (LISt Processing) Scheme avec Racket
- Représentation de la connaissance (Knowledge) et Inférence
  - Logique propositionnelle et 1ière degré
  - Système basé sur les règles
- Introduction à la planification

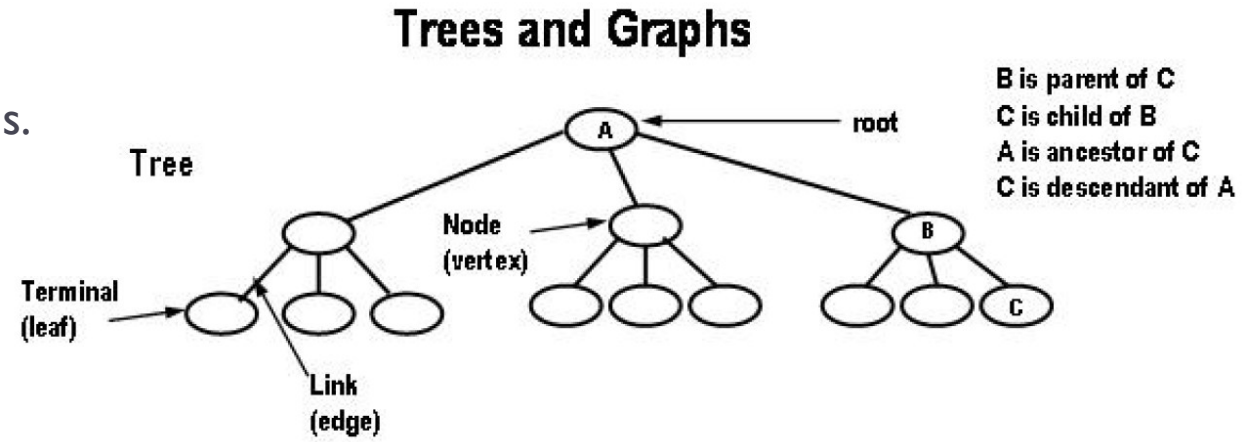
Recherche

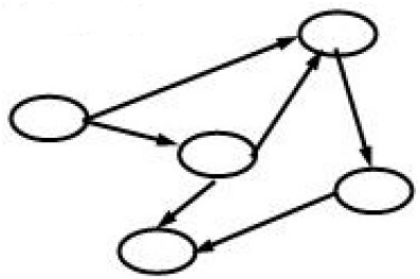
- La recherche : permet l'exploration des alternatives

- Background sur les graphes et les arbres
- Non informé (« aveugle ») vs informé (« heuristique »)
- N'importe quel chemin vs le chemin optimal
- Implémentation et Performance

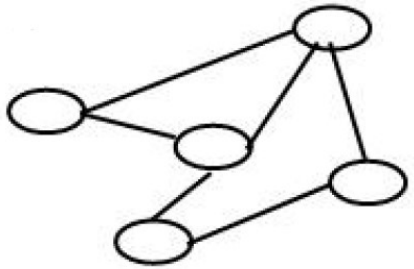
Arbres et Graphes

- Un arbre :
  - Nœuds (vertex) et des liens.
  - Graphe sans cycle
- Racine
- Fils / Parent
- Descendant / Ancêtre



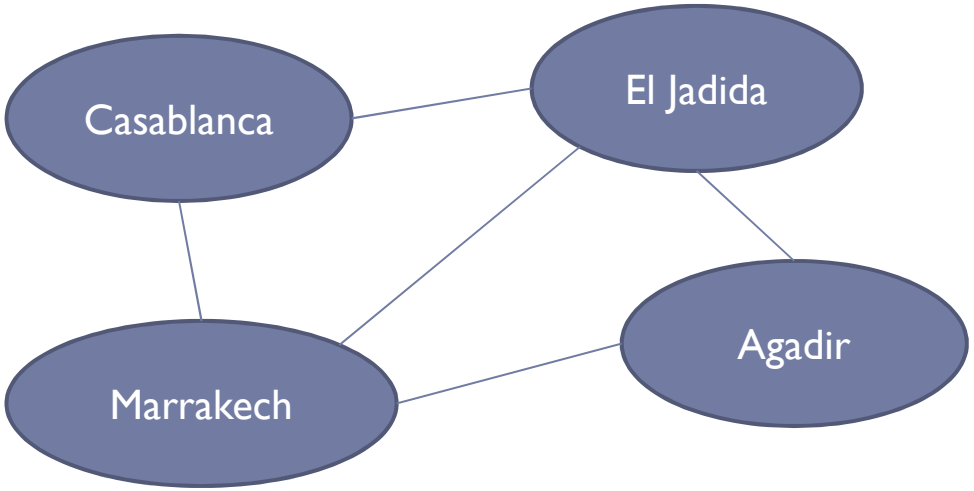


- Graphe Dirigé (chemins d'un seul sens)

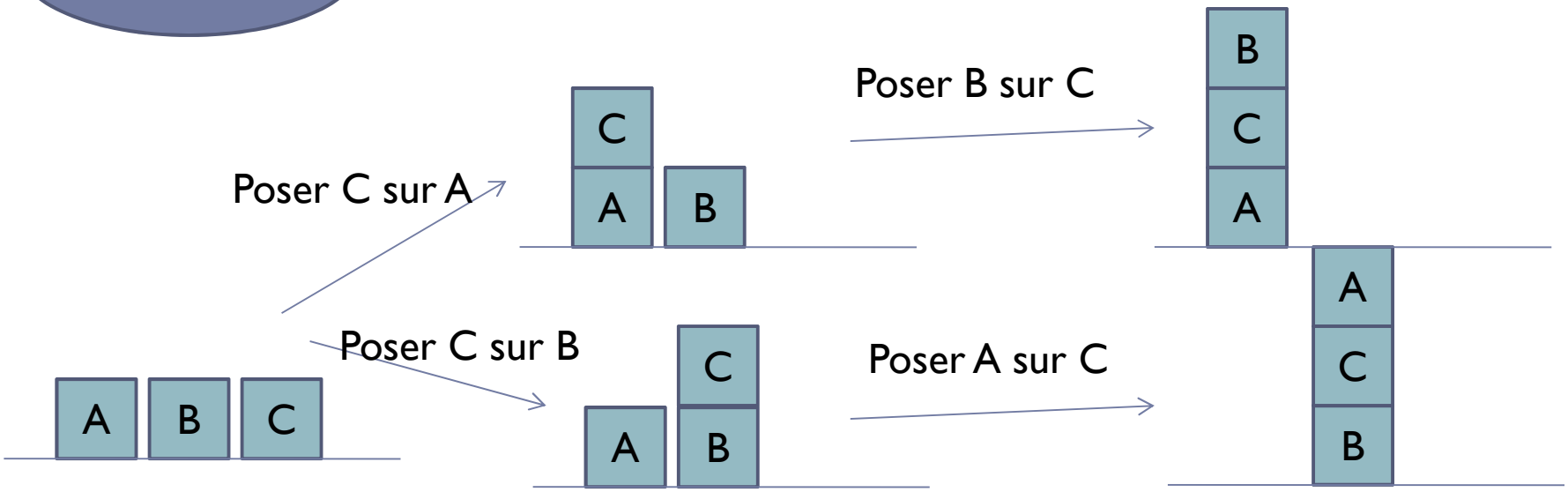


- Graphe non dirigé (chemins à double sens)

Exemple de graphes



- Un chemin dans un graphe : du nœud initial à l'objectif est un plan d'actions

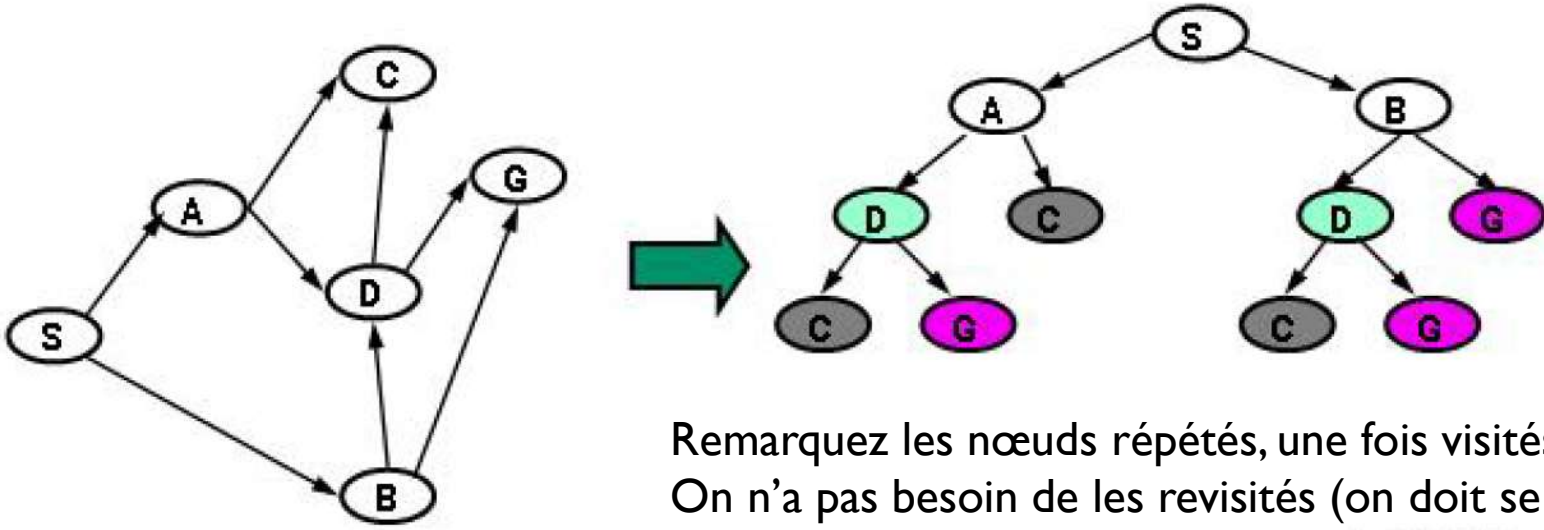


Paradigme de résolution des problèmes

- Quelles sont les états? (tous les aspects importants du problème)
  - Arrangements de pièces (pour assemblage)
  - Positionnement des camions (pour la planification de la distribution)
  - Ville (pour planifier un voyage)
- Quelles sont les actions (Opérateurs)? (Déterministes et discrets)
  - Assembler deux pièces
  - Part pour une ville
  - Déplacer le camion à une nouvelle position
- Quelles est le test de l'objectif (Goal Test)? (conditions pour le succès)
  - Toutes les pièces en place
  - Tous les packages distribués
  - Arrivée à la ville destination

Graphe de recherche comme arbre de recherche

- Arbres sont des graphes dirigés sans cycles et avec des nœuds ayant <= 1 parent
- Problèmes de recherche dans les graphes → Problèmes de recherche dans les arbres
  - Remplacer un lien non dirigé par 2 liens dirigés
  - Ne pas générer de cycles (loops) dans les chemins (ou se rappeler des nœud visités)



Remarquez les nœuds répétés, une fois visités
On n'a pas besoin de les revisités (on doit se rappeler)

Terminologie

- Etat : Fait référence aux **nœud du graphe**, autrement dit états du domaine du problème, par exemple: ville, un arrangement de blocks ou de pièces d'un puzzle.

- Nœud de recherche : Fait référence au **nœud de l'arbre** de recherche généré par l'algorithme de recherche.

  - Plusieurs nœuds peuvent faire référence au même état.
  - Chaque nœud représente un état dans la réalité.
  - Un nœud représente un chemin (depuis l'état initial de la recherche à l'état associé au nœud)
  - Parce que les nœuds de recherche font partie de l'arbre de recherche, ils ont un nœud ancêtre unique. (excepté pour la racine)
  - Se référer au graphe et arbre précédent.

Classes de recherche

Classe	Nom	Opérations
N'importe quel chemin Non informé (aveugle)	Recherche en profondeur (Depth First Search) Recherche en largeur (Breadth First Search)	Exploration systématiques de tout l'arbre de recherche jusqu'à trouver un nœud objectif (Goal)
N'importe quel chemin Informé	Recherche meilleur en premier Best- First	Utilisation de mesures heuristiques pour estimer un état; ex: estimation de la distance à l'objectif
Optimal Non informé	Coût uniforme	Utilisation de la mesure longueur du chemin Trouver le plus court chemin
Optimal Informé	A*	Utilisation de la mesure longueur du chemin et des heuristiques Trouver le plus court chemin

Exemple de recherche (1/4)

- Problème des cruches d'eau
  - On dispose de 2 cruches d'eau et d'un robinet pour le remplir, la première à une capacité de 4 litres, et la seconde de 3 litres.
  - Les cruches ne disposent pas de repère de mesure
  - PB: Obtenir exactement 2 litres d'eau dans la 1ière cruche

Exemple de recherche (2/4)

- Etats : Ensemble des paires ordonnées (C_1, C_2) tel que:
  - C_1, quantité d'eau dans la cruche 1 (4 litres)
  - C_2 quantité d'au dans la cruche 2 (3 litres)
  - C_1 dans {0,1,2,3,4} et C_2 dans {0,1,2,3}

- Etat initial : Cruches vides → (C_1, C_2) = (0,0)

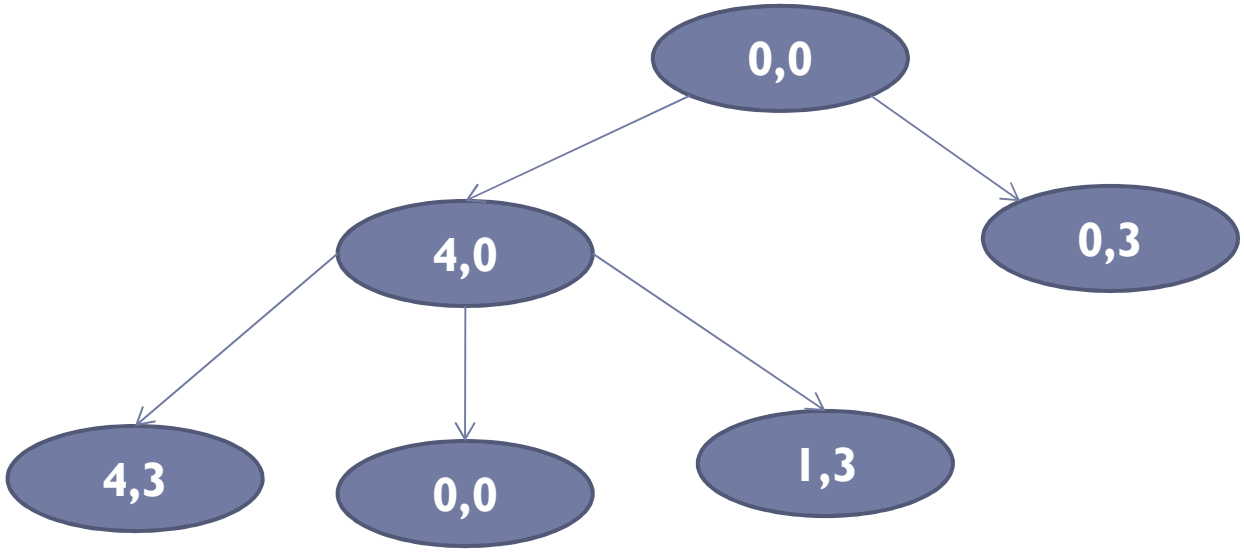
- Etat Objectif : (C_1, C_2) = (2,x); Quelque soit x on a réussi le test de l'objectif

Exemple de recherche (3/4)

- **Plan d'actions** :

  - P1 : remplir la cruche 1 (4 litres)
  - P2 : remplir la cruche 2 (3 litres)
  - P3 : vider la cruche 1 (4 litres)
  - P4 : vider la cruche 2 (3 litres)
  - P5: verser la cruche 1 dans la cruche 2 (4 litres dans 3 litres)
  - P6: verser la cruche 2 dans la cruche 1 (3 litres dans 4 litres)

Exemple de recherche (4/4)



Continuer jusqu'à trouver une solution

Algorithme de Recherche

- Un nœud de recherche est un chemin d'un état X jusqu'à la racine ex: (X, B, A, S)
- L'état du nœud de recherche est l'état le plus récent ex: X
- Soit Q une liste de nœuds de recherche ex: ((X, B, A, S) (C, B, A, S) …)
- Soit S l'état initial

1. Initialiser Q avec le nœud de recherche (S) seulement; Mettre Visité = {S}
2. Si Q est vide, Echec; Sinon, choisir un nœud de recherche N de Q
3. Si Etat(N) est l'objectif (Goal), return N (On a réussi à trouver l'objectif)
4. Sinon Enlever N de Q
5. Trouver tous les fils de Etat(N) ne se trouvant pas dans la liste Visité et créer les extensions de N pour chaque descendant.
6. Ajouter les extensions des chemins de N à Q; Ajouter les fils de l'Etat(N) à la liste Visité.
7. Go to Etape 2

- **Décisions critiques (Stratégies) :**
  - Etape 2 : Choisir un nœud de recherche N de Q
  - Etape 6: Ajouter les extensions des chemins de N à Q

Stratégie de recherche

- Recherche non-informé (aveugle)

- Recherche en Profondeur (Depth-First)
  - Choisir le 1^{ier} élément de Q
  - Ajouter les extensions au début de la liste Q
  - LIFO
  - Capacité de Q dépend de la profondeur de l'arbre

- Recherche en Largeur (Breadth-First)
  - Choisir le 1^{ier} élément de Q
  - Ajouter les extensions à la fin de Q
  - FIFO
  - Capacité de Q devient très grande, puisqu'on décale la recherche du plus long chemin.

Tester l'objectif

- L'algorithme s'arrête (à l'étape 3) quand Etat(N) est objectif = Goal, ou en général quant Etat(N) satisfait le test de l'objectif

- On pouvait tester l'objectif à l'étape 6 quand les chemins sont ajouté à Q. Cela pourrait améliorer le temps de la recherche plus rapidement.

- Pour des raisons de généralisation de l'algorithme avec celui de la recherche optimal, le test est fait dans l'étape 3.

Terminologie

- Visité : Un état M est visité en premier quand un chemin à M est ajouter à Q en premier.
  - En général, un état est dit « Visité » s'il apparait comme un nœud de recherche dans Q.
  - Un nœud visité est placé dans Q, mais pas encore examiné pour le test.

- Développé: ou (Etendu): Un état M est développé, quand c'est un état d'un nœud de recherche qui vient d'être enlevé de Q. A ce moment :
  - les descendants de M sont visités
  - Le chemin qui mène à M est étendu aux descendants éligibles de M
  - En principe un état peut être étendu plusieurs fois.
  - On parle d'extension du nœud de recherche qui amène vers M (au lieu de M)
  - Une fois le nœud est étendu, on a fini avec ce nœud.

- Cette distinction joue un rôle important dans la différenciation des algorithmes.

Les états visités

- Se rappeler des états visités améliore le temps de recherche dans les graphes, sans affecter l'exactitude de l'algorithme.
  - Permet d'éviter les cycles (loops)
  - Mais un espace additionnel est nécessaire pour stocker les états visités.

- Si le but est de trouver un chemin de l'état initial à l'état objectif, il n'y a pas d'avantage à ajouter des nœuds de recherche avec des états qui sont déjà des états d'autres nœuds de recherche.

- N'importe quel état joignable du nœud la deuxième fois pourrait être joignable du nœud la première fois

- On note que lorsqu'on utilise la liste « Visité », chaque état aura seulement un seul chemin (un nœud de recherche) dans Q.

- On reverra ce point pour la recherche optimale

Implémentation : les états visités

- Garder une liste de « Visité » n'est pas l'implémentation idéale.
- Si les états du graphe sont connu à l'avance comme un ensemble explicite.
  - L'espace alloué à « Visité » peut être un flag dans l'état lui-même.
  - Rend l'ajout et la vérification d'un état s'il est visité une opération consommant un temps constant.
- Si les états du graphe ne sont pas connu à l'avance (plus commun à l'IA), on peut utiliser une table de Hashage (Hash Table) pour détecter efficacement les états déjà visités
- A noter que, dans tous les cas, le coût de l'espace supplémentaire d'une liste « Visité » sera proportionnel au nombre d'états dans le graphe.
  - Qui peut être très grand dans certains problèmes.

Terminologie

- Heuristique : Quelque chose qui peut aider à trouver la solution rapidement, peut aider mais pas toujours, contrairement à trouver des solutions « garantis » ou « optimales ».

- Fonction Heuristique : en terme de recherche, une fonction qui calcul une valeur pour un état (mais ne dépend pas du chemin à cet état) qui peut aider à guider la recherche.

- Distance estimé à l'Objectif : cette type de fonction heuristique dépend de l'état et de l'objectif. Un exemple classique est la distance linéaire utilisé pour estimé un réseau de route. Ce type d'information peut aider à améliorer l'efficacité de la recherche

Stratégie de recherche

- Recherche non-informé (aveugle)
- Recherche en Profondeur (Depth-First)
  - Choisir le 1^{ier} élément de Q
  - Ajouter les extensions au début de la liste Q
  - LIFO
- Recherche en Largeur (Breadth-First)
  - Choisir le 1^{ier} élément de Q
  - Ajouter les extensions à la fin de Q
  - FIFO
- Recherche informé
- (Best-First)
  - Choisir le meilleur élément de Q d'abord (mesuré par une valeur heuristique de l'état)
  - Ajouter les extensions n'importe ou dans la liste Q
  - Laisser Q ordonné de façon à ce que l'on trouve efficacement le meilleur élément
  - Même aspect que la recherche en largeur

Implémentation : Chercher le meilleur nœud (Best)

- Il y a plusieurs possibilités pour trouver le meilleur nœud dans Q;
  - Scanner Q pour trouver la valeur minimale
  - Trier Q et choisir le premier élément
  - Laisser Q triée et faire des insertions laissant Q triée
  - Traiter Q comme une liste de priorité (Priority Queue)

- La meilleure solution dépend entre autres du nombre de fils en moyenne.

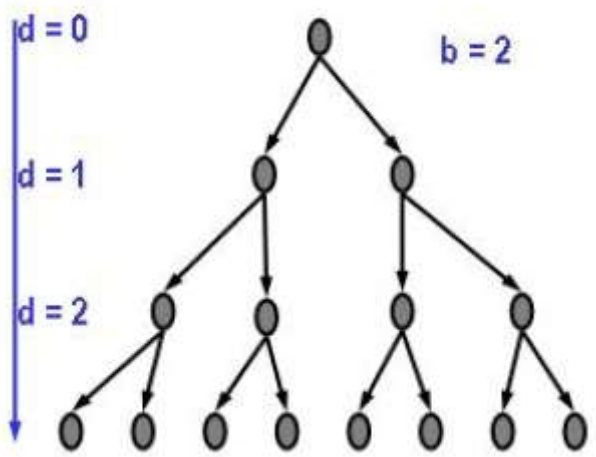
Temps d'exécution

(scénario le plus pessimiste : $T_{max} \approx$ max # Visité)

- Le nombre d'état dans un espace de recherche peut être exponentiel pour une 'profondeur' donnée. (ex: Nombre d'actions dans un plan, nombre de mouvement dans un jeu)

- Toute recherche, avec ou sans liste 'visité' doit visiter un état au moins une fois dans le cas le plus pessimiste.

- Toutes les recherches dans le cas le plus pessimiste auront un temps d'exécution proportionnel au nombre total d'états et donc à la profondeur d.



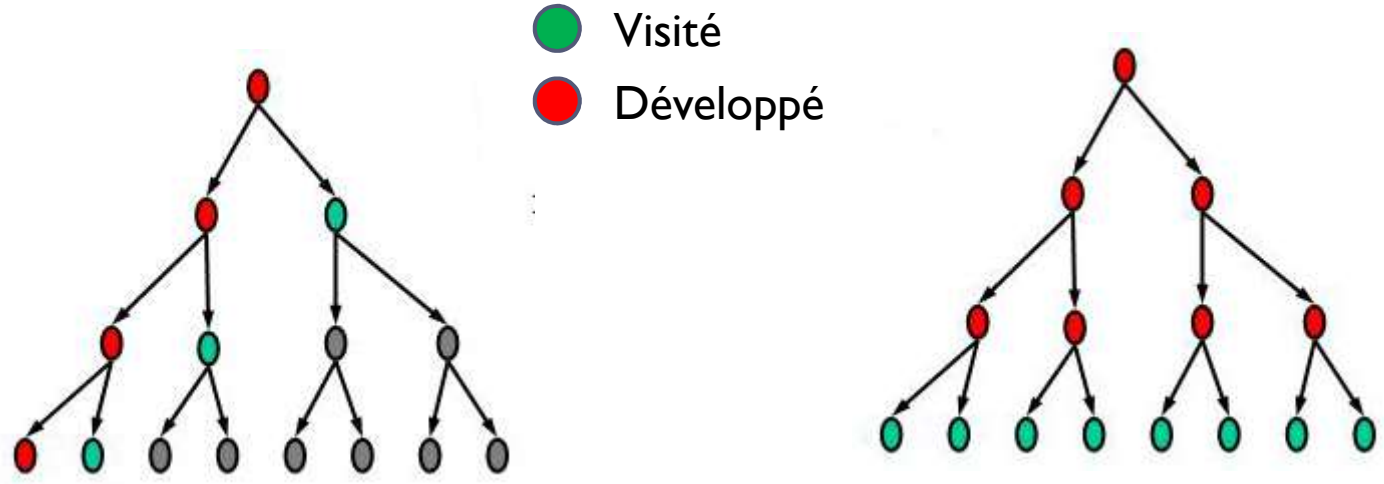
d est la profondeur
b est le facteur de branchement

$b^d < (b^{d+1}-1)/(b-1) < b^{d+1}$

Nombre d'états dans l'arbre

Espace

(scénario le plus pessimiste : max $Q_{Espace} \approx$ max(#Visité - #Développé)



En Profondeur max Q_{espace}
$(b-1)d \approx bd$

En Largeur max Q_{espace}
b^d

Coût et performance des méthodes de recherche (N'importe quel chemin)

- Recherche dans un arbre un facteur b de branchement et d de profondeur (sans utilisation d'une liste visité, on assume que l'espace d'états est l'arbre lui-même, donc on ne peut pas revisité un état)

Méthode de recherche	Pire Temps	Pire Espace	Chemin trouvé garanti
Recherche en profondeur	b^{d+1}	bd	Oui
Recherche en largeur	b^{d+1}	b^d	Oui
Recherche du meilleur	b^{d+1}	b^d	Oui

- Pire temps est proportionnel au nombre de nœuds ajouté à Q
- Pire espace est proportionnel à la longueur de Q maximale

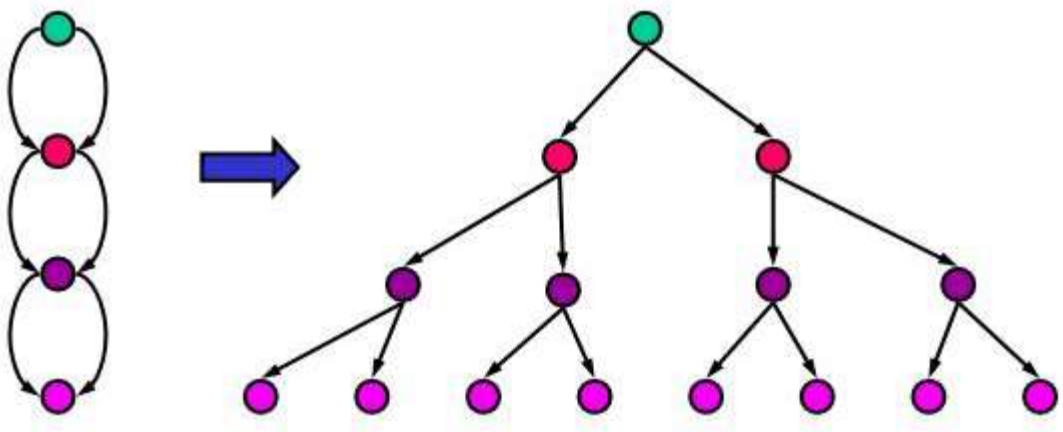
Coût et performance des méthodes de recherche (N'importe quel chemin)

- Recherche dans un arbre un facteur b de branchement et d de profondeur (avec utilisation d'une liste visité)

Méthode de recherche	Pire Temps	Pire Espace	Chemin trouvé garanti
Recherche en profondeur	b^{d+1}	~~bd~~ b^{d+1}	Oui
Recherche en largeur	b^{d+1}	~~b^d~~ b^{d+1}	Oui
Recherche du meilleur	b^{d+1}	~~b^d~~ b^{d+1}	Oui

- Pire temps est proportionnel au nombre de nœuds ajouté à Q
- Pire espace est proportionnel à la longueur de Q maximale (en plus de la liste visité)
  - **Pourquoi devrions nous utiliser une liste visitée?**

Etats vs. Chemins



- Considérons la recherche dans ce cas avec et sans liste visité?
  - Avec la liste visité (Pire temps de performance)→ Nombre d'états
  - Sans liste visité (Pire temps de performance) → Nombre de chemins
- Même sans cycle, la liste visité permet de réduire l'espace de recherche de nombre de chemins au nombre d'état.
- En plus de la détection de cycle, c'est aussi un tradeoff ajoutant de l'espace pour réduire le temps d' éxecution.

Espace

- Dans les vrai problèmes de recherche, la mémoire est une limite
- Imaginons une recherche sur un arbre avec un facteur de branchement 8 et une profondeur de 10. On assume qu'un nœud a besoin de 8 octet d'espace disque.
- La recherche en largeur requiert:
  - $(2^3)^{10} * 2^3 = 2^{33}$ Octets = 8 G octets

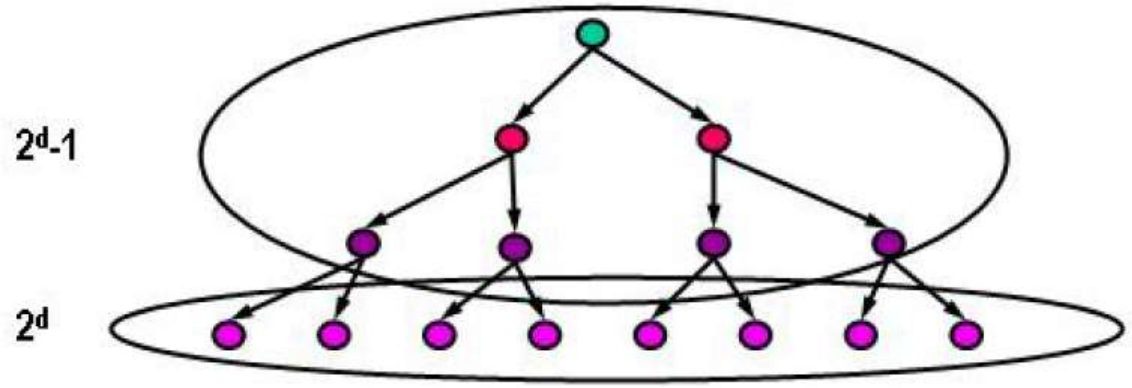
- Stratégie : Echanger le temps par la mémoire.
  - Simulation de la recherche en largeur par plusieurs applications de la recherche en profondeur, chacune jusqu'à une limite de profondeur. (PDS : Progressive Deepening Search)
  - 1- C=1
  - 2- DO DFS TO max Depth C. Si Path trouvé, return Path.
  - 3- ELSE, incrémente C et GOTO 2.

Recherche en Profondeur Progressive (Progressive Deepening Search)

- La recherche en profondeur requiert moins d'espace (linéaire par rapport à la profondeur) mais :
  - DFS peut s'exécuter pour toujours dans des espaces de recherches avec des chemins de longueur infinie.
  - DFS ne garanti pas de trouver un objectif non profond
- La recherche en largeur (BFS) garanti de trouvé un objectif non profond s'il existe mais :
  - BFS requiert beaucoup d'espace (exponentiel par rapport à la profondeur)
- La recherche en profondeur progressive (PDS) a les avantages de DFS et BFS
  - PDS requiert moins d'espace (linéaire par rapport à la profondeur)
  - PDS garanti de trouver un objectif non profond s'il existe

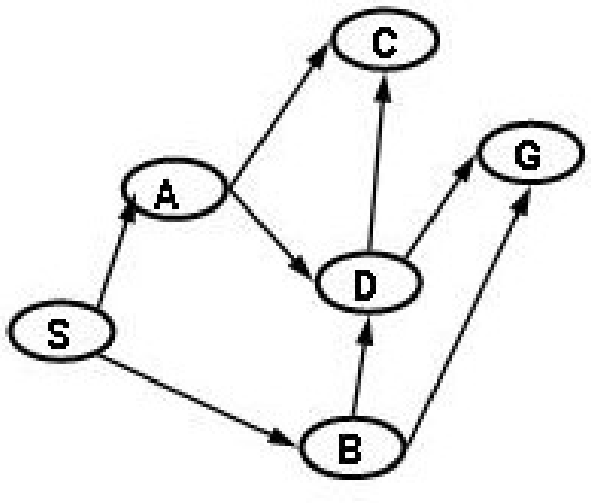
Recherche en Profondeur Progressive (Progressive Deepening Search)

- Est-ce que PDS n'est pas très pénalisante
  - On explore les même nœuds plusieurs fois.
- Dans les arbres exponentiel, le temps dominant est celui de la recherche profonde.
- Par exemple, si le branchement facteur est de 2, alors le nombre de nœuds à profondeur d est 2^d, alors que le nombre de nœud du niveau d'avant est de 2^{d-1}, alors la différence entre chercher l'arbre en entier et entre chercher le niveau le plus profond est au pire des cas un facteur de 2 en performance.



Exemple : Recherche en Profondeur
(Depth First)

	Q	Visité
1	(S)	S
2	(A S) (B S)	A, B, S
3	(C A S) (D A S) (B S)	C, D, B, A, S
4	(D A S) (B S)	C, D, B, A, S
5	(G D A S) (B S)	G, C, D, B, A, S

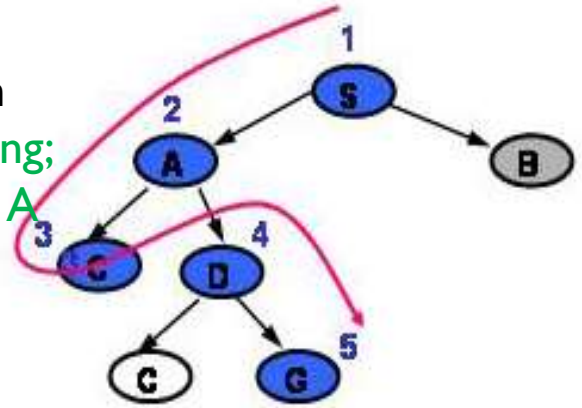


- Choisir le premier élément de Q, ajouter les extensions au début de Q
- Chemin en ordre inversé, l'état du nœud est la première entrée.

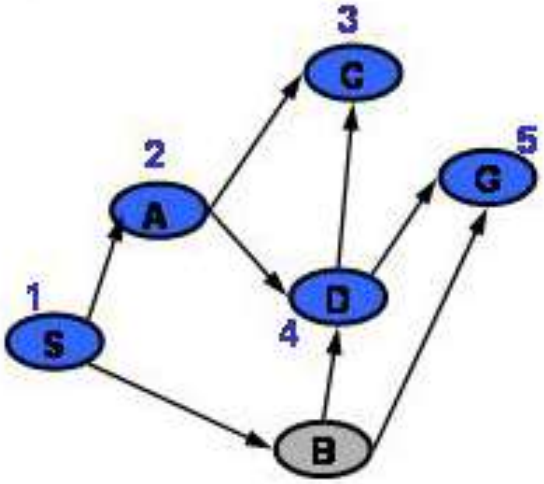
Exemple : Recherche en Profondeur
Autre angle de vue

- Les numéros indiquent l'ordre de retrait de Q (Développé)
- Bleu :Visité et développé
- Gris :Visité

Ici il y a un
Backtracking;
De C vers A



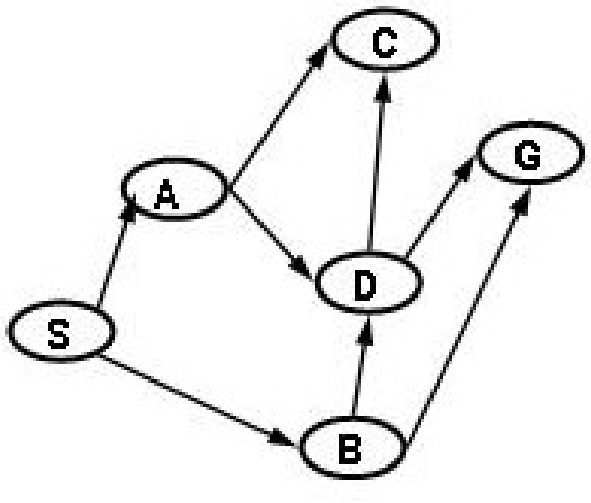
NB : ici C n'est
Pas visité



Exemple : Recherche en Profondeur

(Depth First : Sans la liste 'Visité')

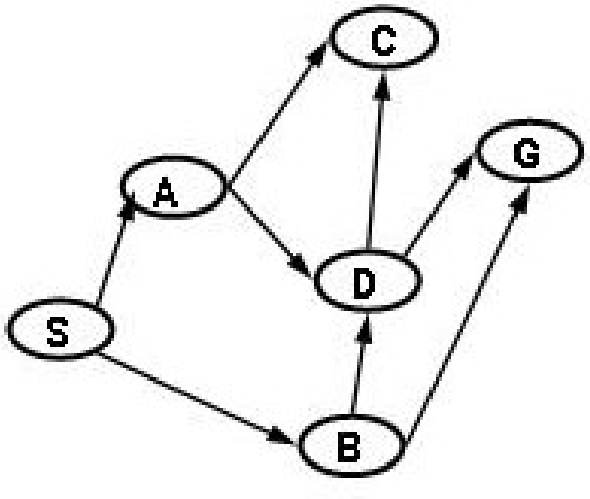
	Q
1	(S)
2	(A S) (B S)
3	(C A S) (D A S) (B S)
4	(D A S) (B S)
5	(C D A S) (G D A S) (B S)
6	(G D A S) (B S)



- Choisir le premier élément de Q, ajouter les extensions au début de Q
- Chemin en ordre inversé, l'état du nœud est la première entrée.
- On ne développe pas chemin, si le résultat est une boucle.

Exemple : Recherche en largeur
(Breadth First)

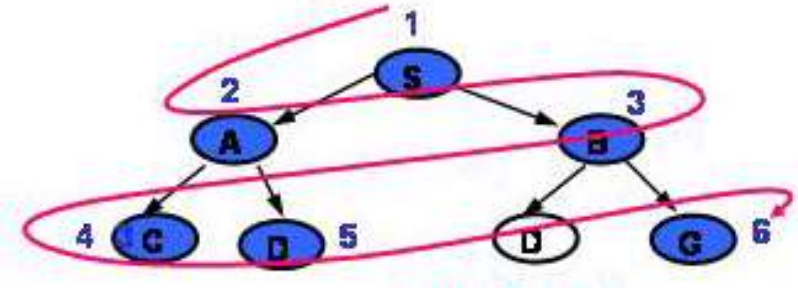
	Q	Visité
1	(S)	S
2	(A S) (B S)	A, B, S
3	(B S) (C A S) (D A S)	C, D, B, A, S
4	(C A S) (D A S) (G B S)*	G, C, D, B, A, S
5	(D A S) (G B S)	G, C, D, B, A, S
6	(G B S)	G, C, D, B, A, S



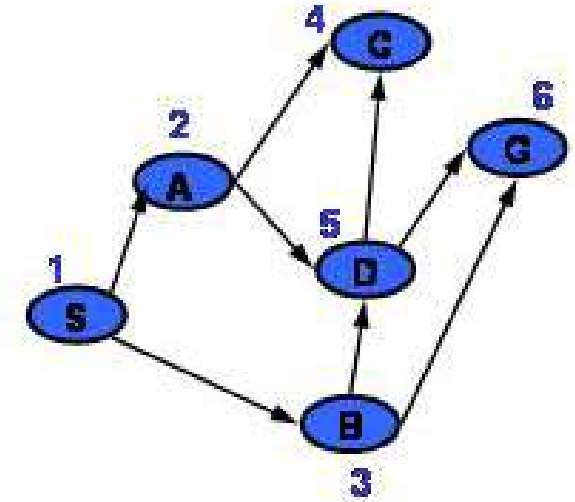
- Choisir le premier élément de Q, ajouter les extensions à la fin de Q
- Chemin en ordre inversé, l'état du nœud est la première entrée.

Exemple : Recherche en largeur
Autre angle de vue

- Les numéros indiquent l'ordre de retrait de Q (Développé)
- Bleu :Visité et développé
- Gris :Visité



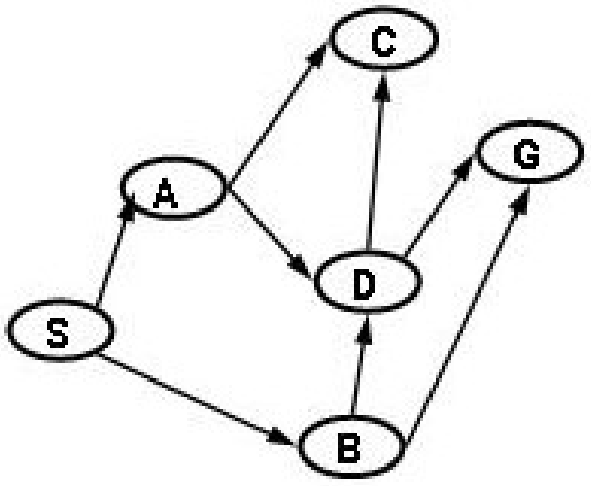
NB : D n'est
pas encore visité



Exemple : Recherche en largeur

(Breadth First : Sans la liste 'Visité')

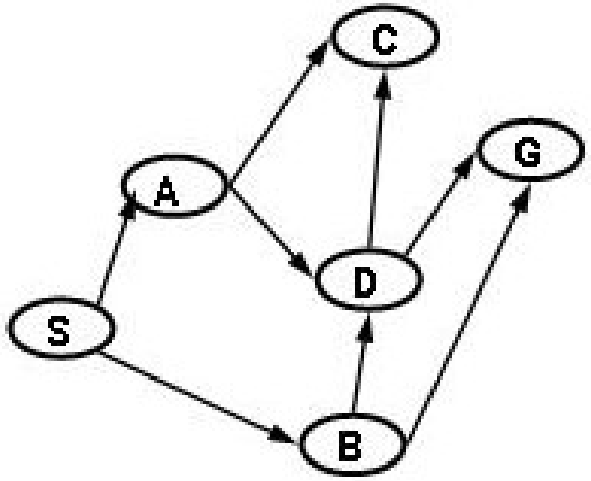
	Q
1	(S)
2	(A S) (B S)
3	(B S) (C A S) (D A S)
4	(C A S) (D A S) (D B S) (G B S)*
5	(D A S) (D B S) (G B S)*
6	(D B S) (G B S)* (C D A S) (G D A S)
7	(G B S)* (C D A S) (G D A S) (C D B S) (G D B S)



- Choisir le premier élément de Q, ajouter les extensions à la fin de Q
- Chemin en ordre inversé, l'état du nœud est la première entrée.

Exemple : Recherche du meilleur d'abord
(Best First)

	Q	Visité
1	(10 S)	S
2	(2 A S) (3 B S)	A, B, S
3	(1 C A S) (3 B S) (4 D A S)	C, D, B, A, S
4	(3 B S) (4 D A S)	C, D, B, A, S
5	(0 G B S) (4 D A S)	G, C, D, B, A, S



Valeurs heuristiques
A=2, C=1, S=10
B=3, D=4, G=0

- Choisir le meilleur (valeur heuristique) élément de Q, ajouter les extensions à Q (Dans n'importe quelle position)
- Chemin en ordre inversé, l'état du nœud est la première entrée.

Classes de recherche

Classe	Nom	Opérations
N'importe quel chemin Non informé (aveugle)	Recherche en profondeur (Depth First Search) Recherche en largeur (Breadth First Search)	Exploration systématiques de tout l'arbre de recherche jusqu'à trouver un nœud objectif (Goal)
N'importe quel chemin Informé	Recherche meilleur en premier Best- First	Utilisation de mesures heuristiques pour estimer un état; ex: estimation de la distance à l'objectif
Optimal **Non informé**	**Coût uniforme**	**Utilisation de la mesure longueur du chemin** **Trouver le plus court chemin**

Algorithme de Recherche utilisé

- Un nœud de recherche est un chemin d'un état X jusqu'à la racine ex: (X, B, A, S)
- L'état du nœud de recherche est l'état le plus récent ex: X
- Soit Q une liste de nœuds de recherche ex: ((X, B, A, S) (C, B, A, S) …)
- Soit S l'état initial

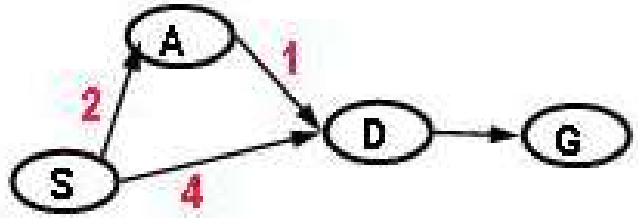
1. Initialiser Q avec le nœud de recherche (S) seulement; Mettre Visité = {S}
2. Si Q est vide, Echec; Sinon, choisir un nœud de recherche N de Q
3. Si Etat(N) est l'objectif (Goal), return N (On a réussi à trouver l'objectif)
4. Sinon Enlever N de Q
5. Trouver tous les fils de Etat(N) ~~ne se trouvant pas dans la liste Visité~~ et créer les extensions de N pour chaque descendant.
6. Ajouter les extensions des chemins de N à Q; Ajouter les fils de l'Etat(N) à la liste Visité.
7. Go to Etape 2

Ne pas utiliser Visité
Pour la recherche optimale

- **Décisions critiques (Stratégies) :**
  - Etape 2 : Choisir un nœud de recherche N de Q
  - Etape 6: Ajouter les extensions des chemins de N à Q

Pourquoi pas une liste visité?

- Pour les algorithmes de 'n'importe quel chemin', la liste 'Visité' ne sera jamais la cause d'un échec si jamais un chemin vers l'Objectif n'existe pas.
- Toutefois, et pour les algorithmes de 'chemin optimal', avec la liste 'Visité' on peut manquer le chemin optimal



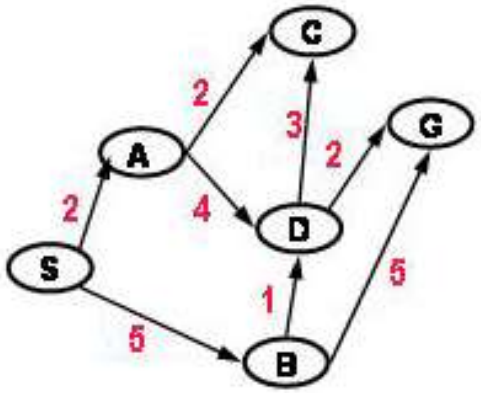
- Le chemin le plus court de S à G est (S A D G)
- Mais, en développant (S), A et D seront ajouter à la liste 'Visité' et (S A) ne serait pas développé (S A D)

Implémentation des stratégies optimale de recherche

- Coût Uniforme
  - Choisir le meilleur (mesuré par la longueur du chemin) élément de Q
  - Ajouter les extensions n'importe ou dans la liste Q

Coût Uniforme

- Comme 'Best-First', seulement on utilise 'la longueur totale' (coût) d'un chemin au lieu d'une heuristique de l'état.
- Chaque lien a une 'longueur' ou 'coût' (toujours >0)
- Objectif : le chemin le moins coûteux.

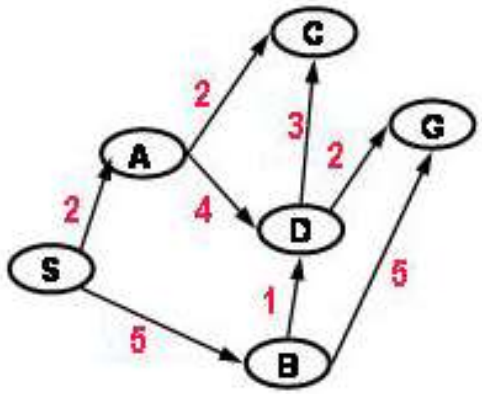


Coût total des chemins:

(S A C)	4
(S B D G)	8
(S A D C)	9

Coût Uniforme

	Q
1	(0 S)
2	(2 A S) (5 B S)
3	(4 C A S) (6 D A S) (5 B S)
4	(6 D A S) (5 B S)
5	(6 D B S) (10 G B S)* (6 D A S)
6	(8 G D B S)(9 C D B S)(10 G B S)* (6 D A S)
7	(8 G D A S) (9 C D A S) (8 G D B S)(9 C D B S)(10 G B S)*



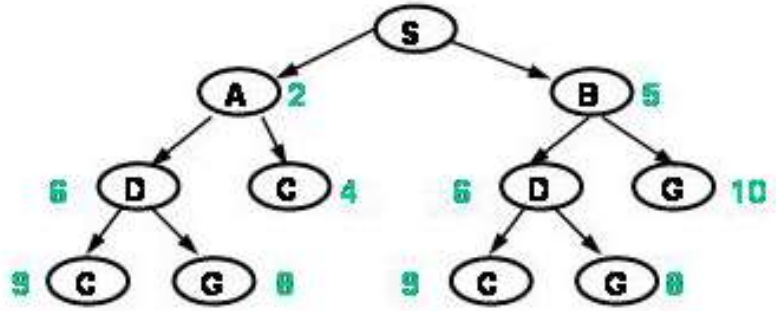
- Choisir le meilleur (Coût) élément de Q, ajouter les extensions à Q (Dans n'importe quelle position)
- Chemin en ordre inversé, l'état du nœud est la première entrée.

Pourquoi ne pas s'arrêter quand l'objectif est visité?

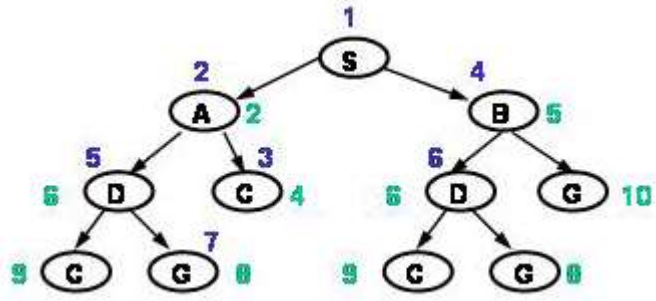
- Dans le cas de coût uniforme, il n'est pas correcte d'arrêter la recherche quand le premier chemin vers l'objectif a été généré; càd quand un nœud ayant comme état l'objectif a été ajouté à Q.
- On doit attendre jusqu'à ce que ce chemin soit retirée de Q pour être tester (dans l'étape 3 de l'algorithme). C'est à ce moment là seulement que l'on peut dire que notre chemin est le plus court puisqu'aucun autre chemin plus court à l'objectif n'a été développé.
- Cela est différent des recherches non-optimale, puisque le test de l'objectif était une affaire de convenance et d'efficacité et non pas d'exactitude de l'algorithme.
- Dans l'exemple précédent, un chemin a été généré à l'étape 5, mais c'est un autre chemin qui a été accepté à l'étape 7.

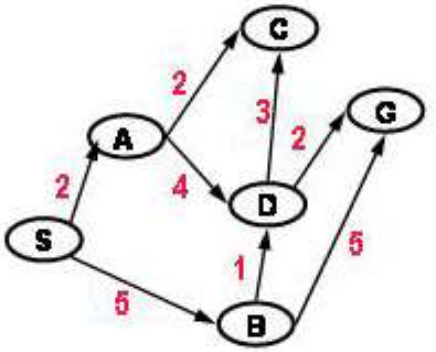
Coût Uniforme

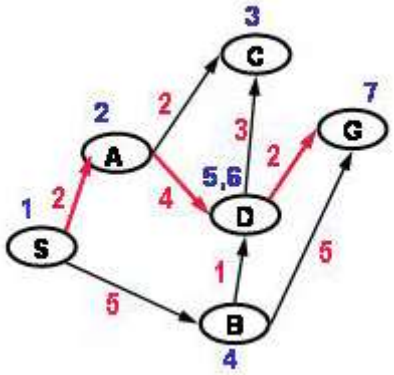
- Une autre façon de voir le coût uniforme
- Le coût uniforme énumère le chemin dans l'ordre des coûts.



Total path cost







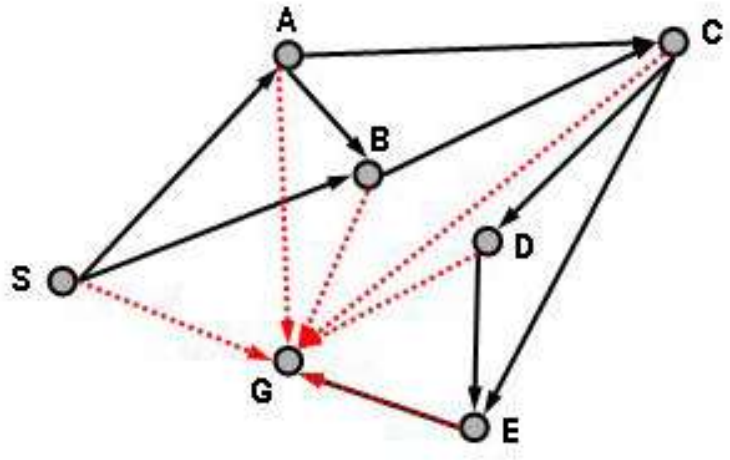
Classes de recherche

Classe	Nom	Opérations
N'importe quel chemin Non informé (aveugle)	Recherche en profondeur (Depth First Search) Recherche en largeur (Breadth First Search)	Exploration systématiques de tout l'arbre de recherche jusqu'à trouver un nœud objectif (Goal)
N'importe quel chemin Informé	Recherche meilleur en premier Best- First	Utilisation de mesures heuristiques pour estimer un état; ex: estimation de la distance à l'objectif
Optimal Non informé	Coût uniforme	Utilisation de la mesure longueur du chemin Trouver le plus court chemin
Optimal informé	**A***	**Utilisation de la longueur du chemin et d'une heuristique Trouve le chemin le plus court**

Direction vers l'objectif

- L'algorithme du coût uniforme cherche à identifier le chemin le plus court pour chaque état dans le graphe et en ordre. Le CU, n'a aucun indice pour chercher un chemin vers l'objectif un peu plus tôt.
  - C'est vraiment un algorithme pour tous les états et non seulement pour la recherche de l'objectif
- On peut introduire un indice avec une fonction heuristique h(N) qui est une estimation de la distance d'un état à l'objectif
  - Combien il nous reste pour parvenir à l'objectif
  - Ce n'est pas la vraie distance, mais une estimation, sinon pourquoi cherchons nous?
- Au lieu d'énumérer la longueur des chemins dans l'ordre longueur(g), énumérer les chemins en termes de :
  - f = estimation total de la longueur du chemin = g + h
- Une estimation qui toujours sous-estime la longueur réelle du chemin à l'objectif est appelé : **estimation admissible**.
  - Par exemple : une estimation de 0 est admissible (mais ne sert à rien)
  - Une distance de ligne droite est une estimation admissible pour la longueur d'un chemin dans un espace euclidien
- L'utilisation d'une estimation admissible garanti que le CU trouve toujours le chemin le plus court. (A* est CU avec une estimation admissible)

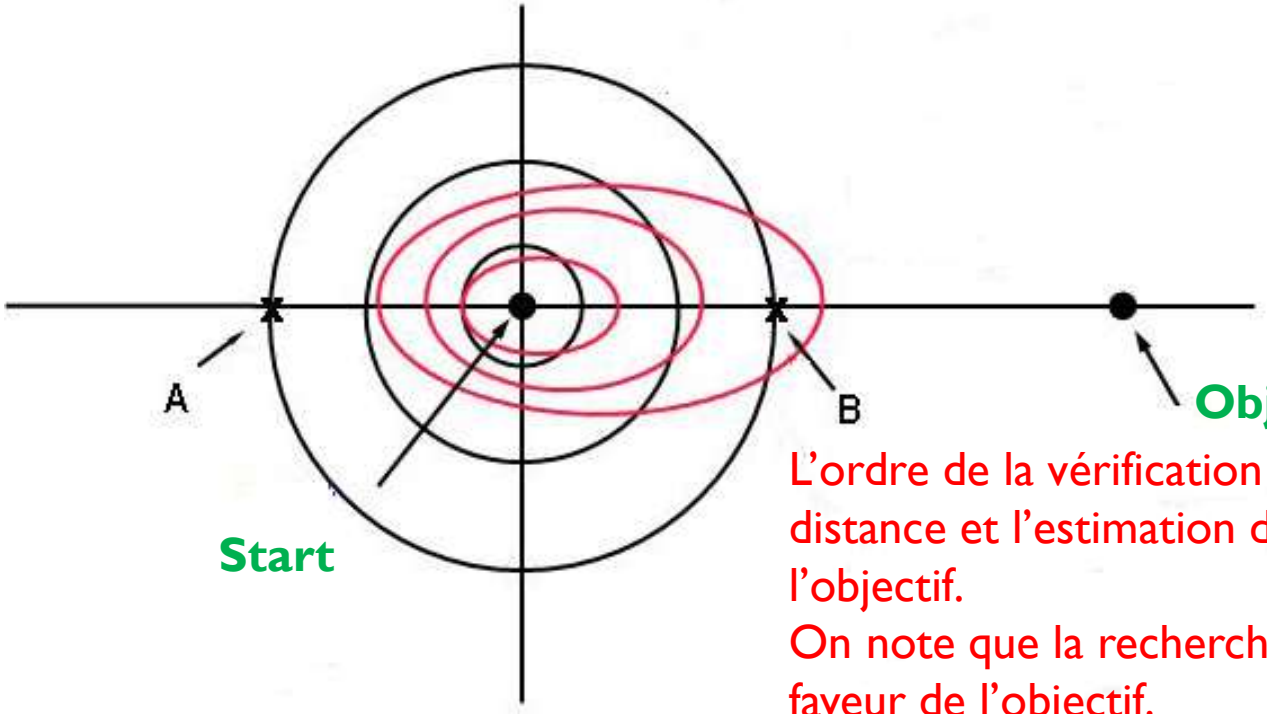
Exemple d'estimation d'une ligne droite



Sous estimation vers G
Ex: B vers G est très sous estimé

Pourquoi utiliser une distance d'estimation vers l'objectif ?

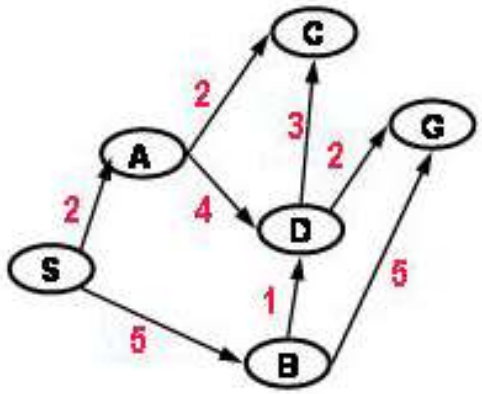
L'ordre dans lequel CU cherche vérifie les états, si A et B sont de la même distance de 'Start', ils seront vérifiés avant tout autre chemin plus long, l'algorithme n'est pas biaisé Et n'a aucun indice où se trouve G.



L'ordre de la vérification utilisant la distance et l'estimation de la distance à l'objectif.
On note que la recherche est biaisée en faveur de l'objectif.

A*

	Q
1	(0 S)
2	(4 A S) (8 B S)
3	(5 C A S) (7 D A S) (8 B S)
4	(7 D A S) (8 B S)
5	(8 G D A S) (10 C D A S) (8 B S)

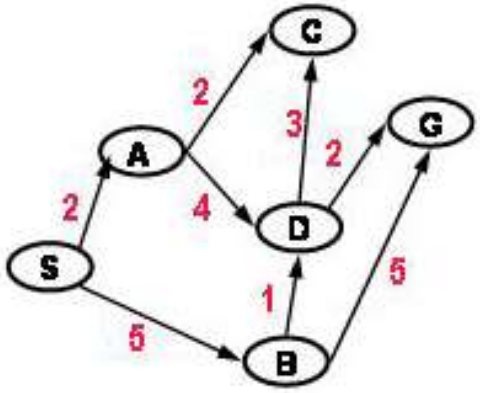


Valeurs heuristiques
A=2, C=1, S=0
B=3, D=1, G=0

- Choisir le meilleur (valeur heuristique) élément de Q, ajouter les extensions à Q (Dans n'importe quelle position)
- Chemin en ordre inversé, l'état du nœud est la première entrée.

A*

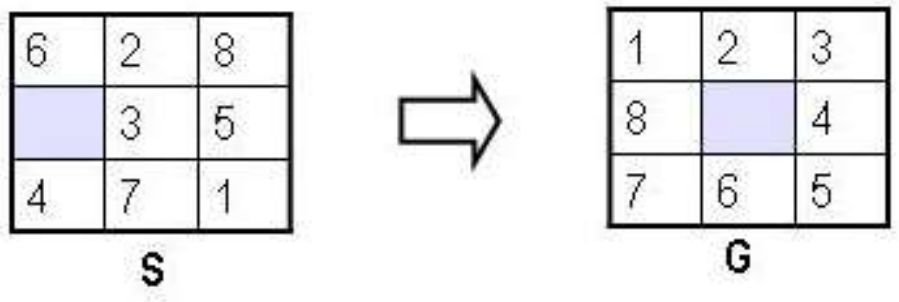
- Soient les liens dans la figure, est ce que la table des valeurs heuristiques qu'on a utilisé sur l'algorithme 'meilleur d'abord' représentent des valeurs admissibles?
  - A ok
  - B ok
  - C ok
  - D doit être <= 2
  - S est très grand, peut toujours être 0 ?



Valeurs heuristiques
A=2, C=1, S=10
B=3, D=4, G=0

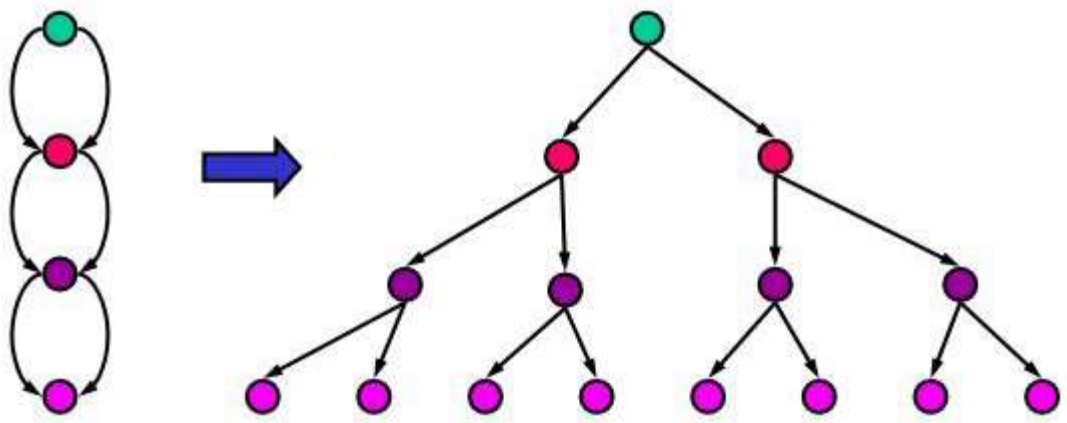
A*

- 8 puzzle : Bouger les carreaux pour arriver à l'objectif. Imaginez que vous bougez le carreau vide



- Alternatives pour l'estimation de la distance (nombre de mouvement) à l'objectif
  - Nombre de carreaux qui ne sont pas dans la bonne position
  - Somme de la distance de Manhattan, distance du carreau à son objectif (17 dans l'exemple ci-dessous):
    - Distance de Manhattan : $M((x_1,y_1), (x_2,y_2) = |x_1-x_2| + |y_1-y_2|$, chaque mouvement diminue la distance d'exactement un carreau.
- La deuxième estimation est bien meilleur pour prévoir le nombre de mouvements

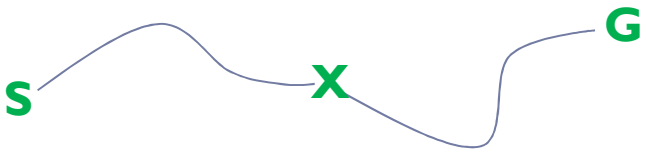
Etats vs. Chemins (reconsidération de A*)



- On ne peut pas utiliser une liste 'Visité' et considérer l'optimalité
- Peut on utiliser autre chose pour que le pire cas soit proportionnel aux nombres d'états et non pas au nombre de chemins (sans loop).

Programmation dynamique : Principe d'optimalité

- Le plus court chemin de S à G via un état X est construit à partir du chemin le plus court de S à X et le chemin le plus court de X à G



- On garde seulement le plus court chemin de S vers n'importe quel état X; si on trouve un autre chemin dans Q, on abandonne le chemin le plus long.
- Il est à noter que dans l'algorithme de coût uniforme, la première fois qu'un nœud avec un état X est développé (ou retiré de Q), ce chemin est le plus court de S à X
  - Cela est du au fait que CU développe les nœuds dans l'ordre des longueurs des chemins
- Donc une fois qu'un chemin à X, a été développé, on a pas besoin de développer un autre chemin vers X.
  - On peut sauvegarder une liste de ces états 'développés'.
  - On abandonne le nœud, si l'état X résultant est dans la liste des 'Développés'
  - On appelle la liste développé une **liste développé stricte**.
- Avec cette optimisation, le CU est plus efficace.

Algorithme de Recherche optimal simple coût uniforme

- Un nœud de recherche est un chemin d'un état X jusqu'à la racine ex: (X, B, A, S)
- L'état du nœud de recherche est l'état le plus récent ex: X
- Soit Q une liste de nœuds de recherche ex: ((X, B, A, S) (C, B, A, S) …)
- Soit S l'état initial

1. Initialiser Q avec le nœud de recherche (S) seulement;
2. Si Q est vide, Echec; Sinon, choisir un nœud de recherche N ayant le coût minimal de Q
3. Si Etat(N) est l'objectif (Goal), return N (On a réussi à trouver l'objectif)
4. Sinon Enlever N de Q
5. ..
6. Trouver tous les fils de Etat(N) et créer les extensions de N pour chaque descendant.
7. Ajouter les extensions des chemins de N à Q;
8. Go to Etape 2

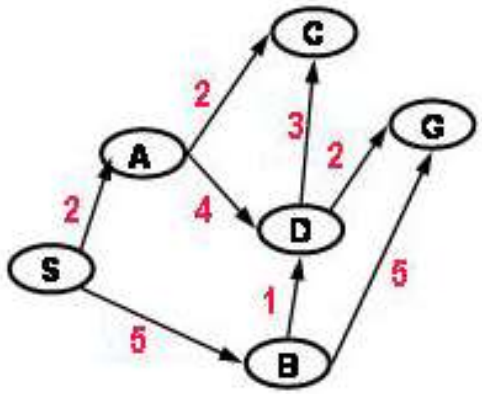
Algorithme de Recherche optimal simple coût uniforme + Liste développé stricte

- Un nœud de recherche est un chemin d'un état X jusqu'à la racine ex: (X, B, A, S)
- L'état du nœud de recherche est l'état le plus récent ex: X
- Soit Q une liste de nœuds de recherche ex: ((X, B, A, S) (C, B, A, S) …)
- Soit S l'état initial

1. Initialiser Q avec le nœud de recherche (S) seulement; **Mettre 'Développé' = { }**
2. Si Q est vide, Echec; Sinon, choisir un nœud de recherche N ayant le coût minimal de Q
3. Si Etat(N) est l'objectif (Goal), return N (On a réussi à trouver l'objectif)
4. Sinon Enlever N de Q
5. **Si Etat(N) est Développé, aller à l'étape 2. Sinon ajouter Etat(N) à Développé.**
6. Trouver tous les fils de Etat(N) **(Qui ne se trouvent pas dans 'Développé')** et créer les extensions de N pour chaque descendant.
7. Ajouter les extensions des chemins de N à Q; **Si un descendant est déjà dans Q, laisser seulement l'état avec le chemin le plus court dans Q.**
8. Go to Etape 2

Coût Uniforme avec une liste développé stricte

	Q	Développé
1	(0 S)	
2	(2 A S) (5 B S)	S
3	(4 C A S) (6 D A S) (5 B S)	S, A
4	(6 D A S) (5 B S)	S, A, C
5	~~(6 D B S)~~ (10 G B S) (6 D A S)	S, A, C, B
6	(8 G D A S) ~~(9 C D A S)~~ ~~(10 G B S)~~	S, A, C, B, D



- Choisir le meilleur élément de Q, ajouter les extensions à Q (Dans n'importe quelle position)
- Chemin en ordre inversé, l'état du nœud est la première entrée.

A* sans une liste développé

- Soit g(N) le coût du chemin de n, avec n le nœud de l'arbre de recherche; càd un chemin partiel
- Soit h(N) est h(Etat(N)), l'estimation heuristique de la longueur du chemin restant pour atteindre l'objectif à partir de l'Etat(N),
- Soit f(N) = g(N) + h(Etat(N)) l'estimation globale du coût d'un chemin d'un nœud; càd l'estimation d'un chemin à un objectif qui commence par un chemin donné par N.
- A* choisit le nœud avec la valeur de f minimal pour développement.
- A* (sans une liste développé) et avec une heuristique admissible est assurer de trouver des chemins optimaux – (ceux avec les plus faibles coût)

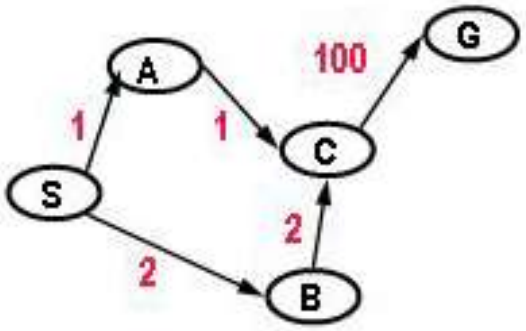
A* avec une liste développé

- Une implémentation de A* avec une liste développé requiert d'autres conditions plus fortes pour les heuristiques (que les sous estimations)
  - Et cela pour garantir de trouver tous les chemins optimaux

- Un contre-exemple : les valeurs heuristiques listées ci-dessous sont tous admissibles (sous estimées) mais A* utilisant une liste développées ne trouvera pas le chemin optimal.

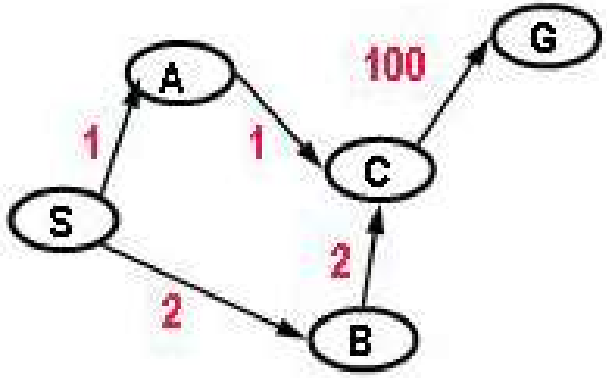
  - L'estimation trompeuse de B fausse l'algorithme, C est développé avant que le chemin optimal est trouvé

Valeurs heuristiques
A=100, C=90, S=0
B=1, G=0



A* avec une liste développé

	Q	Développé
1	(0 S)	
2	(3 B S) (101 A S)	S
3	(94 C B S) (101 A S)	B, S
4	(101 A S) (104 G C B S)	C, B, S
5	(104 G C B S)	A, C, B, S



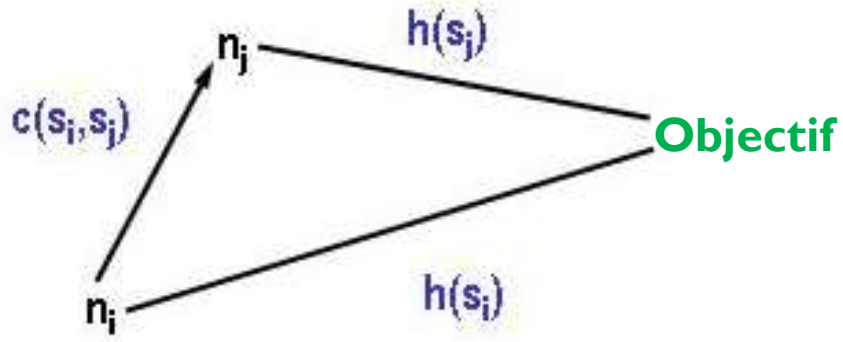
Valeurs heuristiques

A=100, C=90, S=0

B=1, G=0

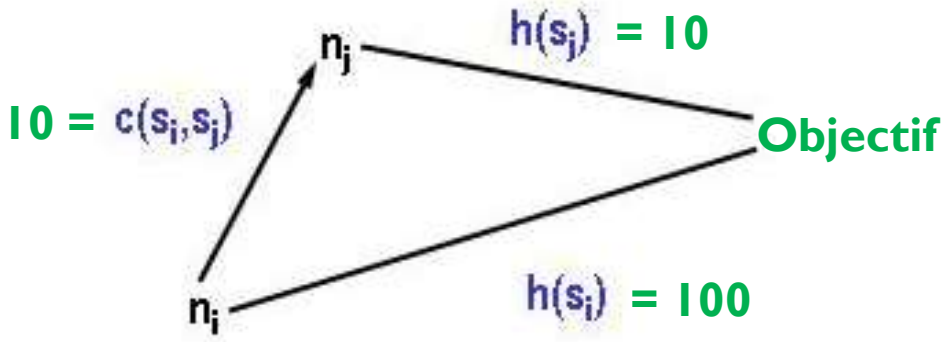
A* avec une liste développé (heuristique consistante)

- Pour implémenter A* avec une liste développé stricte, H doit satisfaire les conditions monotones (heuristiques consistantes)
  - $h(s_i) = 0$, n_i est un Objectif
  - $h(s_i)-h(s_j) <= c(s_i,s_j)$, pour n_j un fils de n_i
- L'heuristique ne doit pas décroitre plus que le coût entre les deux états.



Violation de la condition de consistance

- Exemple de la violation de la condition de consistance
  - $h(s_i)$-$h(s_j)$ <= $c(s_i,s_j)$, pour n_j un fils de n_i
  - Dans l'exemple 100 – 10 > 10
- Si vous croyez que de n_i que l'objectif est à 100 unités, alors, en se rapprochant de 10 unités à n_j ne doit pas vous rapprocher de 10 unités à l'objectif.



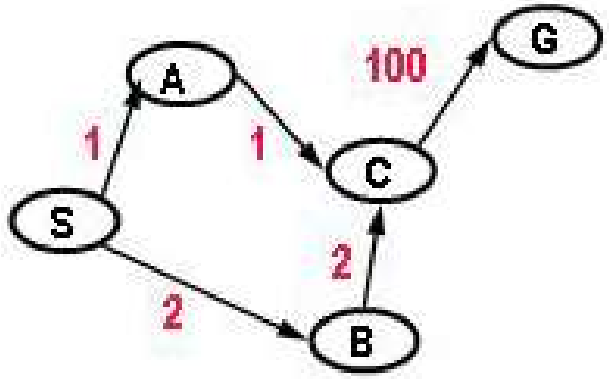
A* sans une liste développé (Rappel)

- Soit g(N) le coût du chemin de n, avec n le nœud de l'arbre de recherche; càd un chemin partiel
- Soit h(N) est h(Etat(N)), l'estimation heuristique de la longueur du chemin restant pour atteindre l'objectif à partir de l'Etat(N),
- Soit f(N) = g(N) + h(Etat(N)) l'estimation globale du coût d'un chemin d'un nœud; càd l'estimation d'un chemin à un objectif qui commence par un chemin donné par N.
- A* choisit le nœud avec la valeur de f minimal pour développement.
- A* (sans une liste développé) et avec une heuristique admissible est assurer de trouver des chemins optimaux – (ceux avec les plus faibles coût)

- L'heuristique consistance est seulement nécessaire pour l'optimalité quand on veut abandonner des chemins, sinon A* sans une liste développé est suffisant

A* sans une liste développé

	Q
1	(90 S)
2	(3 B S) (101 A S)
3	(94 C B S) (101 A S)
4	(101 A S) (104 G C B S)
5	(92 C A S) (104 G C B S)
6	**(102 G C A S)** (104 G C B S)



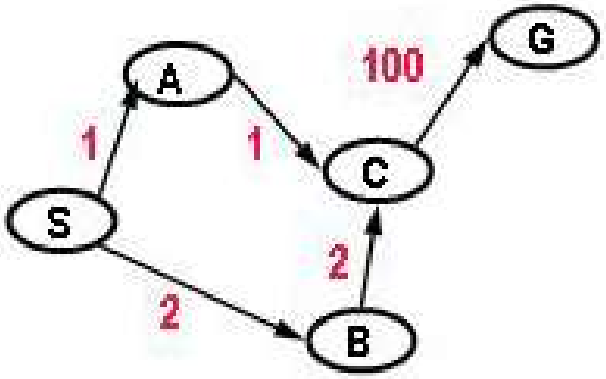
Valeurs heuristiques
A=100, C=90, S=90
B=1, G=0

A* avec une liste développé stricte

- Comme la recherche 'Coût Uniforme'
- Quand un nœud N est développé, si Etat(N) est dans la liste 'Développé', abandonner N, sinon ajouter Etat(N) à la liste 'Développé '
- Si un nœud dans Q, a le même état qu'un descendant de N, on garde seulement celui avec la plus petite f (f= g + h), ce qui correspond aussi au plus petit g.
- Pour que A* (avec une liste développé stricte) garantisse de trouver un chemin optimal, l'heuristique doit être consistante.

A* avec une liste développé

	Q	Développé
1	(90 S)	
2	(90 B S) (101 A S)	S
3	(101 A S) (104 C B S)	B, S
4	(102 C A S) (104 C B S)	A, B, S
5	**(102 G C A S)**	C, B, A, S



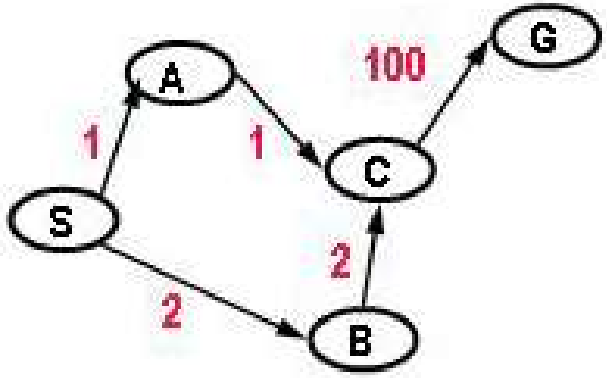
Valeurs heuristiques
Admissibles et consistantes
A=100, C=100, S=90
B=88, G=0

A* avec une heuristique inconsistante

- Que peut on faire si on a une heuristique inconsistante et cherchons des chemins optimaux?
- Modifier A* pour détecter et corriger les inconsistances déjà procédées
- Si on assume que l'on ai entrain d'ajouter nœud$_1$ dans Q, et que nœud$_2$ est présent dans la liste 'développé' sachant que Noeud$_1$Etat = Noeud$_2$Etat.
  - Dans le cas strict : Ne pas ajouter Noeud$_1$ dans Q
  - Dans le cas d'une liste Développé non stricte: (On doit être sûr que Nœud$_1$ n'a pas trouver un meilleur chemin que Noeud$_2$)
    - Si Noeud$_1$.Longueur_Chemin < Nœud$_2$.Longueur_Chemin
      - Supprimer Noeud$_2$ de la liste 'Développé'
      - Ajouter Nœud$_1$ dans Q

A* avec une liste développé et heuristique inconsistante

	Q	Développé
1	(0 S)	
2	(3 B S) (101 A S)	S
3	(94 C B S) (101 A S)	B, S
4	(101 A S) (104 G C B S)	~~C~~, B, S
5	(92 C A S) (104 G C B S)	A, B, S
6	**(102 G C A S)** (104 G C B S)	C, A, B, S



Valeurs heuristiques
A=100, C=90, S=0
B=1, G=0

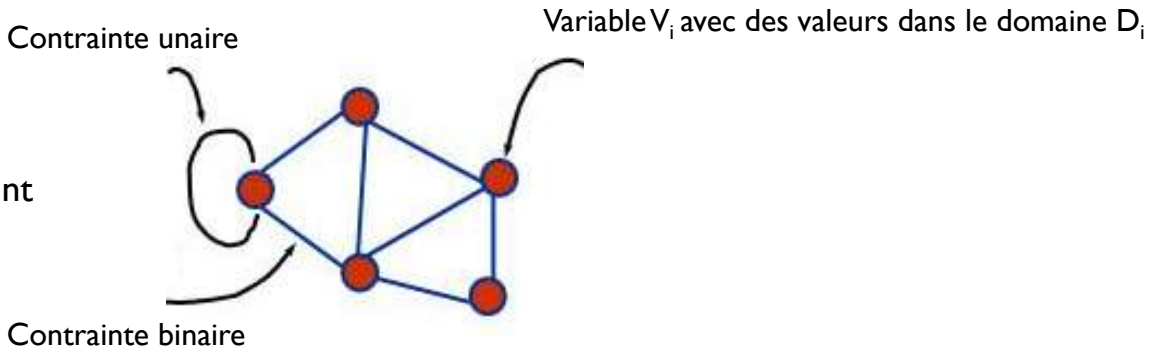
Intelligence Artificielle

- Introduction à l'intelligence artificielle
- Historique de l'IA
- Problèmes de recherche
  - Recherche dans les graphes
  - **CSP (Problèmes de Satisfaction de Contraintes)**
- Représentation de la connaissance (Knowledge) et Inférence
  - Logique propositionnelle et 1ière degré
  - Système basé sur les règles

Problèmes de satisfaction de contrainte

- Classe générale de problèmes
- CSP Binaire

Une variable à contrainte unaire est une variable que l'on a restreint son domaine



Graphe à contrainte

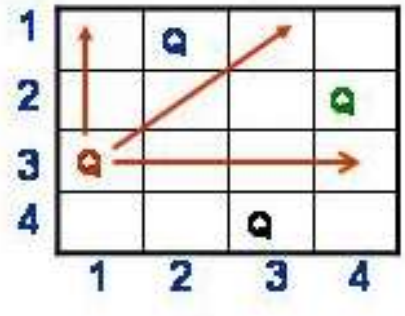
Problème :

Trouver une valeur d_i de D_i pour chaque V_i tel que toutes les contraintes sont satisfaites
(Trouver une notation valable pour les variables)

Problème des N-Reines

- Placer N reines dans un N*N échiquier tel que nulle d'entres elles ne peut attaquer les autres.

- Variables : Positions N*N de l'échiquier



- Domaines : Reine ou Blanc

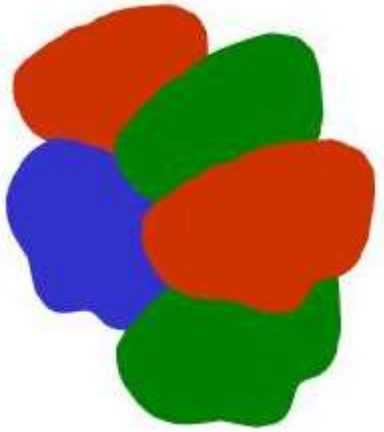
- Contraintes : 2 positions (Verticales; horizontales; et diagonales) ne peuvent être des reines.

Problèmes de planification

- Choisir le temps pour des activités .
- Variables : Activités
- Domaines : Ensemble de début des activités (ou ensemble de portions de temps)
- Contraintes : Activités qui utilisent la même ressource ne peuvent pas être en même temps et toutes les pré-conditions doivent être satisfaites.

Coloriage de graphes

- Choisir les couleurs pour les régions d'une carte, en évitant de colorier les régions adjacentes avec la même couleur.



- Variables : Régions
- Domaines : Couleurs permises
- Contraintes : Régions adjacentes doivent avoir des couleurs différentes.

Problèmes de satisfaisabilité (3-SAT)

- Problème NP-complet original
- (A OR B OR C) AND (!A OR C OR !B) ….
- Trouver les valeurs pour les variables booléennes A, B, C, … pour satisfaire la formule

- Variables : Clause sous la forme conjonctive (CNF)

- Domaines : Affectation des variables booléennes pour rendre la formule vrai

- Contraintes : Clauses partageant les variables booléennes doivent aussi partager leurs valeurs

Bonne nouvelle / Mauvaise nouvelle

- Bonne nouvelle : Classes de problèmes très générales et intéressantes
- Mauvaise nouvelle : Inclus les problèmes NP-Difficile.
- La performance dépendra du domaine et non des algorithmes.

Exemple de CSP

- Soit 40 cours ($cours_1$, $cours_2$, ..) et 10 Semestres (S_1, S_2, S_3,..S_{10}), trouver une planification légale
- Contraintes :
  - Pré-requis
  - Cours offerts dans des semestres défini
  - Cours limités par semestre (par exemple 6 par semestre)
  - Eviter les conflits d'avoir des cours en même temps.

- CSP: n'est pas pour exprimer des préférences (Soft constraints)

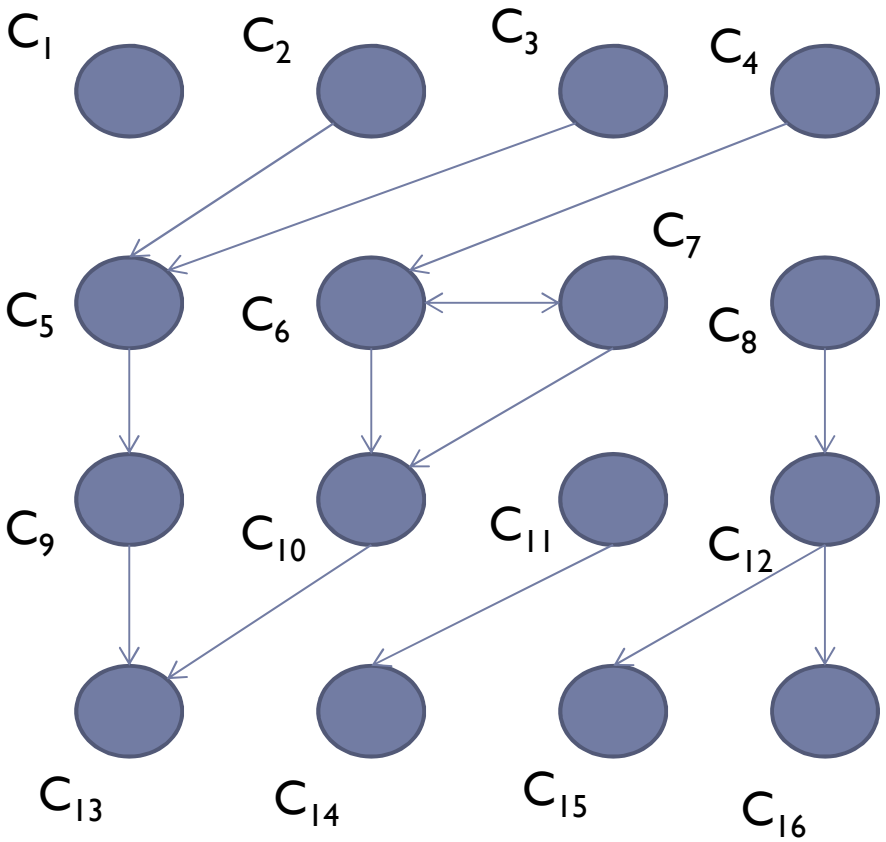
Exemple CSP (choix des variables et des valeurs)

- Variables
  - 1 – Semestres ?
  - 2- Créneaux de semestres ?
    - $(S_1, 1), (S_1,2), (S_1, 3), (S_1,4), (S_1, 5), (S_1,6), (S_2, 1), (S_2,2), ..$
  - 3- Cours?
- Domaines
  - Des combinaisons légales de par exemple 6 cours (très grand ensemble de valeurs)
  - Cours offerts durant ce semestre
  - Semestre ou Créneaux de semestre (Créneaux de semestre permet d'exprimer les contraintes sur un nombre limité de cours/semestre)

Les contraintes

- Les cours comme variables et les créneaux de semestre comme valeurs, on aura:
  - Les pré-requis ??
  - Les cours offerts seulement dans quelques semestres ??
  - Limiter le nombre de cours dans un semestre ??
  - Conflit de temps??

Exemple (16 cours 4 semestres de 4 créneaux)



Problèmes de Satisfaction de contraintes

- Les problèmes CSP nécessitent:

- 1- Propagation des contraintes pour éliminer les valeurs qui ne font pas partie de la solution

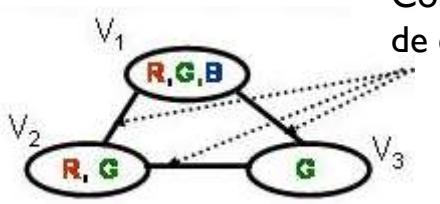
- 2- Une recherche, pour explorer les affectations valides

Propagation des contraintes (Cohérence des arcs)

- La cohérence des arcs élimine des valeurs, du domaine des variables, qui ne peuvent pas faire partie d'une solution cohérente.
- $V_i \rightarrow V_j$
  - L'arc dirigé (V_i, V_j) est un arc cohérent si
  - Quelque soit x dans D_i, il existe y dans D_i, tel que (x,y) est valable par la contrainte de l'arc
  - Autrement dit, pour chaque valeur dans le domaine D_i, il existe une valeur dans le domaine V_j qui satisfait la contrainte de l'arc.
- On peut arriver à la cohérence de l'arc en supprimant des valeurs de D_i (ceux qui ne satisfassent pas la condition)

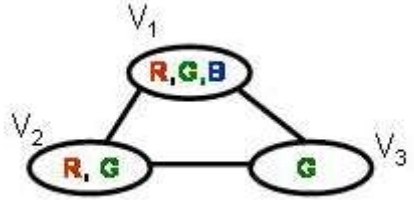
Exemple de Propagation des contraintes

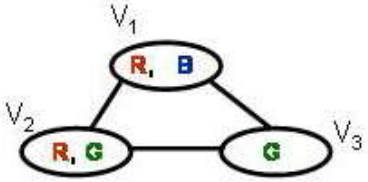
- Coloriage des graphes
  - Valeurs initiales sont indiqués

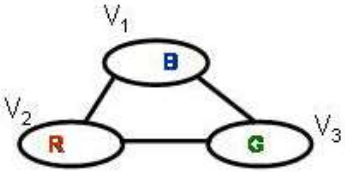


Contraintes d'utilisation de couleurs différentes

Arc Examiné	Valeur supprimé
V1 - V2	Aucune
V1 - V3	V1(G)
V2 - V3	V2(G)
V1 - V2	V1(R)
V1 - V3	Aucune
V2 - V3	Aucune







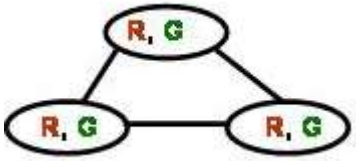
- Chaque contrainte d'un arc non-dirigé est réellement deux contraintes d'arcs dirigés, l'effet montré ci-haut et après examen des deux arcs

Exemple de Propagation des contraintes

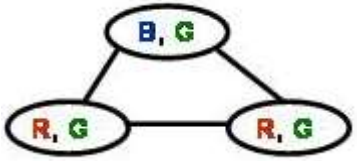
- En générale, on a besoin de faire une autre passe sur n'importe quelle variable (entête d'un arc), dont on vient de supprimer une valeur.

- Si l'on assume 'd' le nombre maximale de valeurs dans D
- Et 'e' le nombre d'arcs.
  - Il faut $O(d^2)$ pour tester la cohérence de chaque arc
  - Tester la cohérence de tous les arcs une fois est de l'ordre de $O(ed^2)$.
  - On peut tester un arc plus qu'une fois après propagation d'une suppression. Mais si on test un arc après que des variables de son domaine ont changé, alors le nombre maximum que l'on peut vérifier un arc est 'd'. D'où la pire situation est de l'ordre de $O(ed^3)$

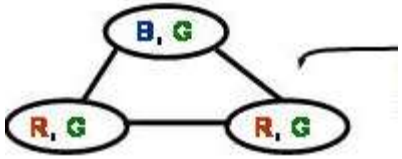
La cohérence n'est pas assez



Arc cohérent mais
pas de solutions



Arc cohérent mais
1solution



On assume B,R n'est pas permis

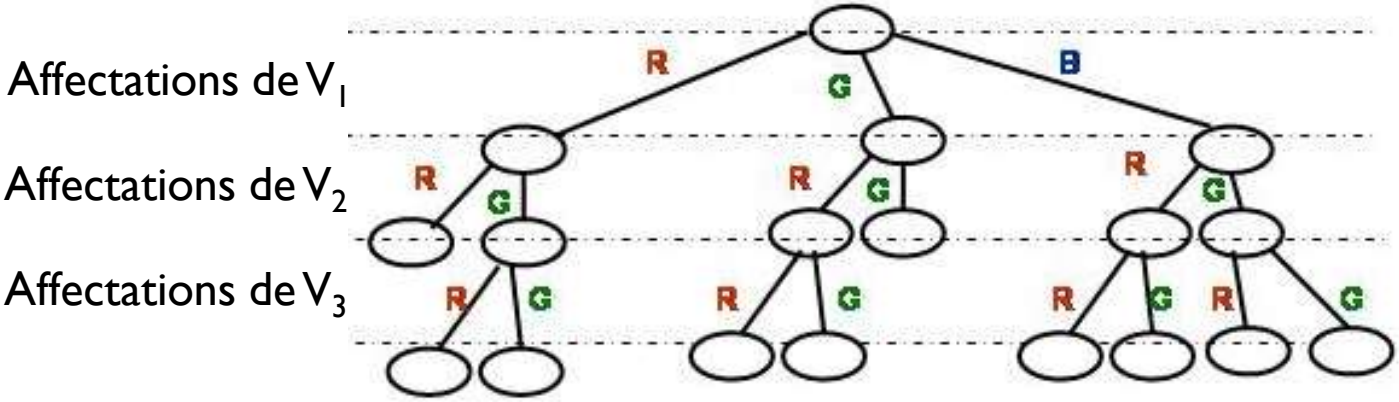
Arc cohérent mais
2 solutions;
(B,R,G) et
(B,G,R)

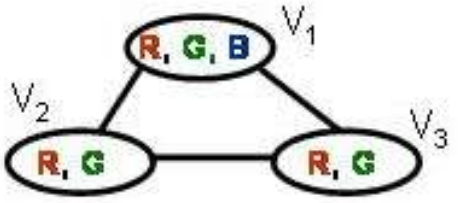
Quand il y a plusieurs valeurs pour une variable, on ne sait donc pas s'il y a zéro, une ou plusieurs réponses cohérentes.

On a donc besoin de chercher une solution

Chercher une solution (Backtracking)

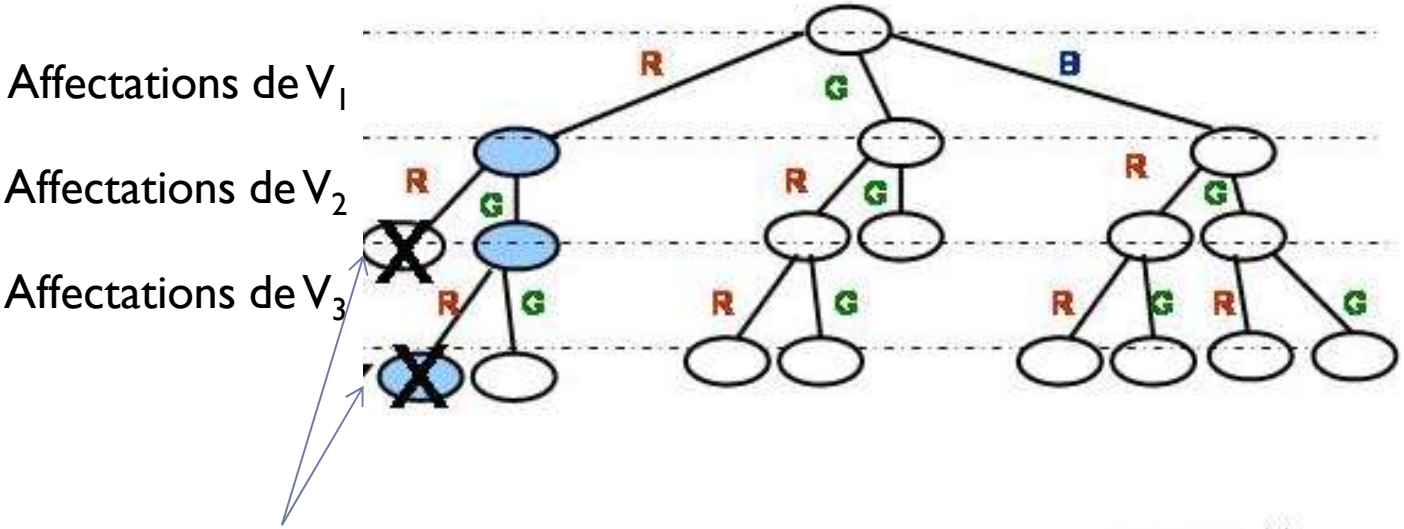
- Quand on a beaucoup de valeurs dans un domaine (ou/et la cohérence des arcs est faible), la cohérence des arcs n'aide pas beaucoup, on a besoin de faire une recherche.
- L'approche la plus simple est le 'Backtracking' (DFS)



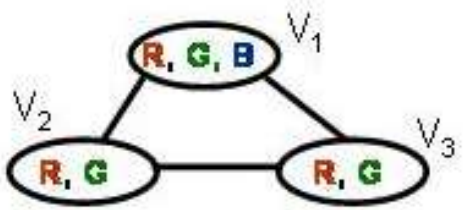


Chercher une solution (Backtracking)

- Quand on a beaucoup de valeurs dans un domaine (ou/et la cohérence des arcs est faible), la cohérence des arcs n'aide pas beaucoup, on a besoin de faire une recherche.
- L'approche la plus simple est le 'Backtracking' (DFS)

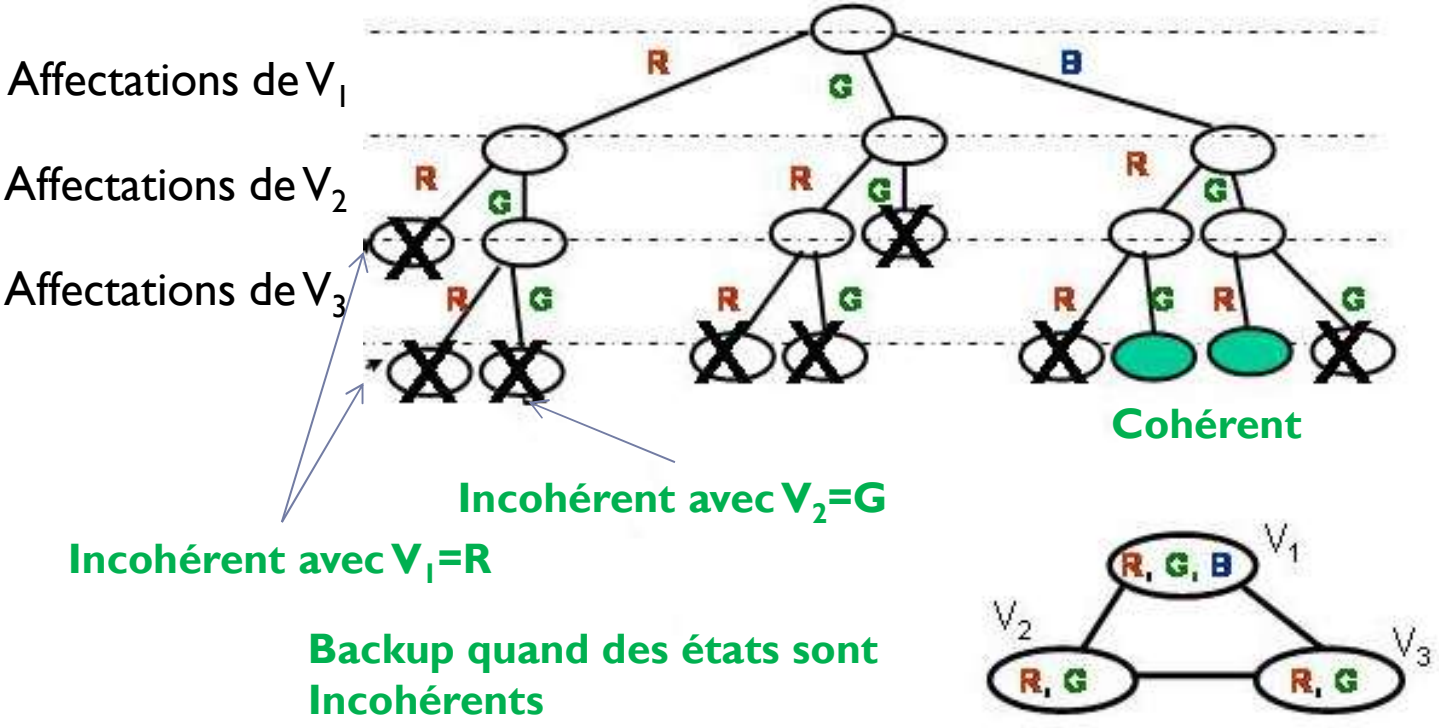


Backup quand des états sont Incohérents



Chercher une solution (Backtracking:BT)

- Quand on a beaucoup de valeurs dans un domaine (ou/et la cohérence des arcs est faible), la cohérence des arcs n'aide pas beaucoup, on a besoin de faire une recherche.
- L'approche la plus simple est le 'Backtracking' (DFS)



Combiner le backtracking avec la propagation des contraintes

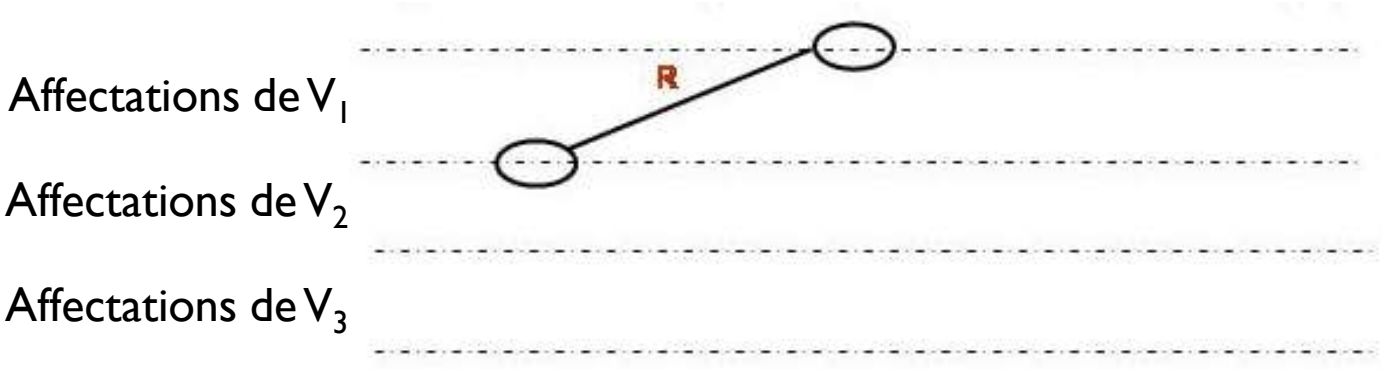
- Un nœud dans l'arbre BT est une affectation partielle dans lequel le domaine de chaque variable a été choisi (une tentative de mettre la variable à une seule valeur)

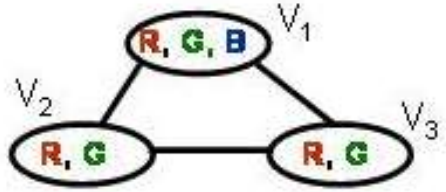
- Utiliser la propagation des contraintes (cohérence des arcs) pour propager les effets de l'affectation de cette tentative; càd éliminer les valeurs incohérentes avec les valeurs courantes.
  - Cela réduit le facteur de branchement

- Question : Combien de propagation devrions nous faire?
  - Réponse: Propagation locale des domaines avec des affectations uniques → Vérification à l'avance (Forward Checking)

Le Backtracking avec le Forward Checking

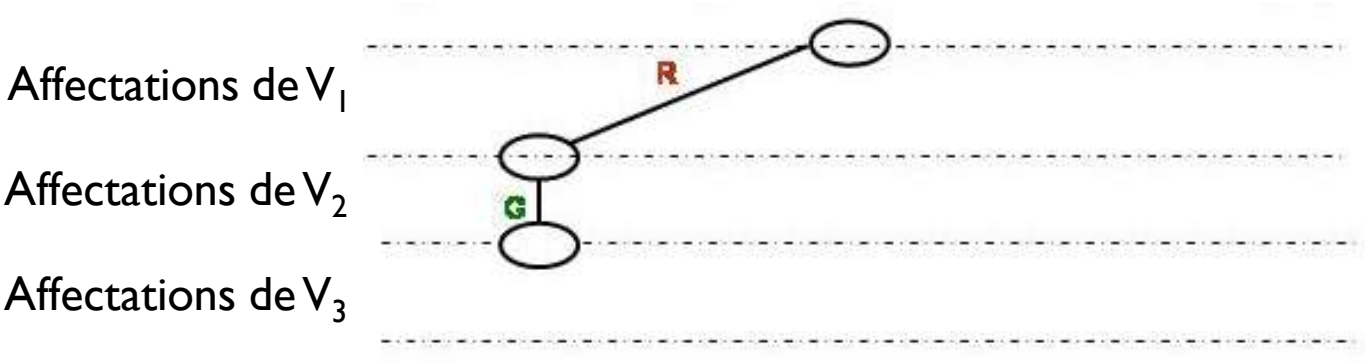
- Quand examiner une affectation $V_i = d_k$, supprimer toute valeur incohérente avec cette affectation des domaines voisins dans le graph de contraintes.

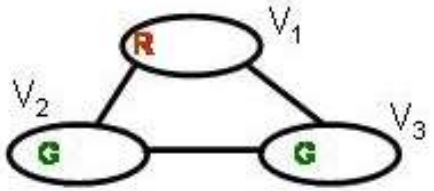




Le Backtracking avec le Forward Checking

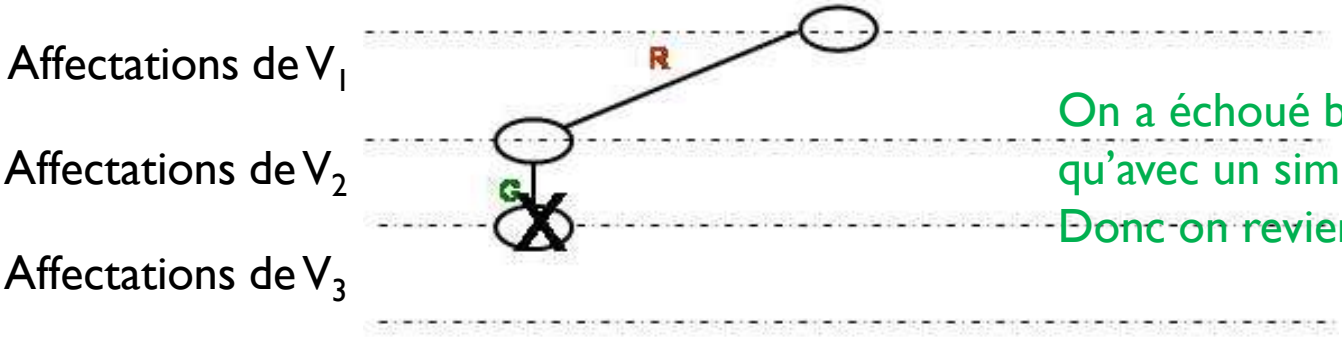
- Quand examiner une affectation $V_i = d_k$, supprimer toute valeur incohérente avec cette affectation des domaines voisins dans le graph de contraintes.



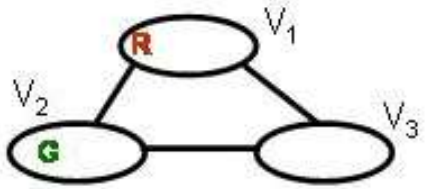


Le Backtracking avec le Forward Checking

- Quand examiner une affectation $V_i = d_k$, supprimer toute valeur incohérente avec cette affectation des domaines voisins dans le graph de contraintes.



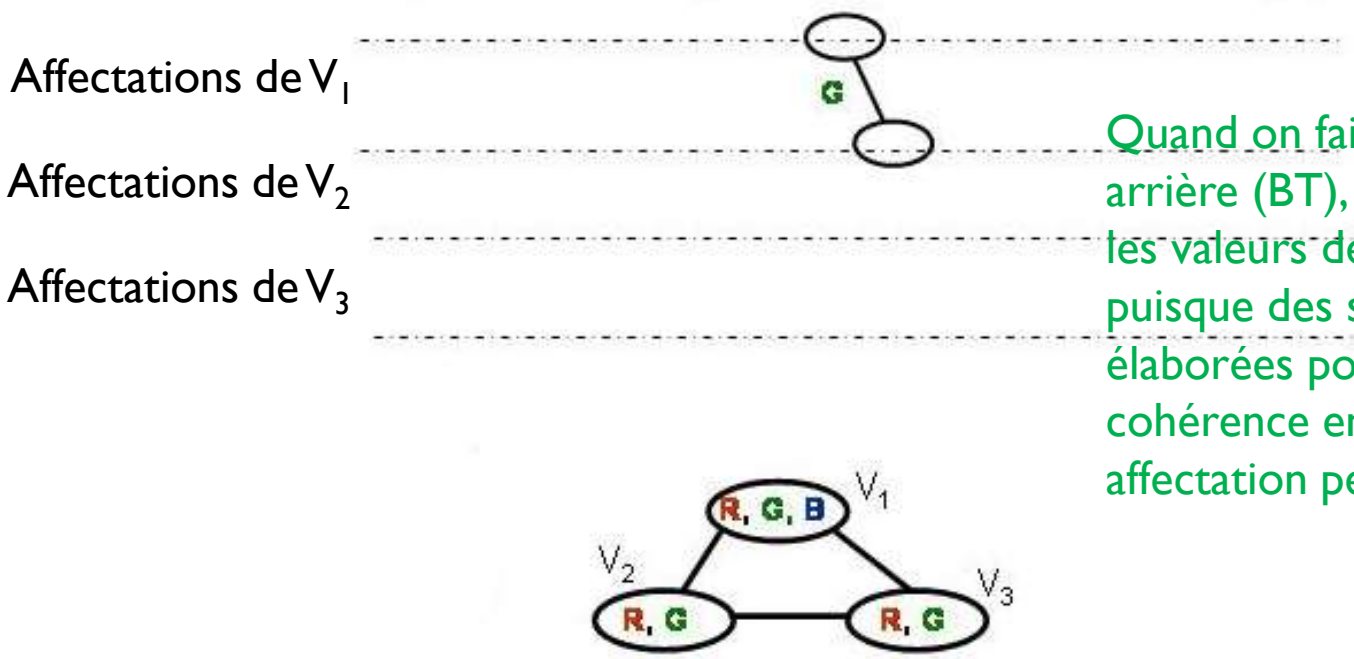
On a échoué bien plus tôt que qu'avec un simple BT
Donc on revient en arrière (BT)



On a un conflit quand domaine devient vide

Le Backtracking avec le Forward Checking

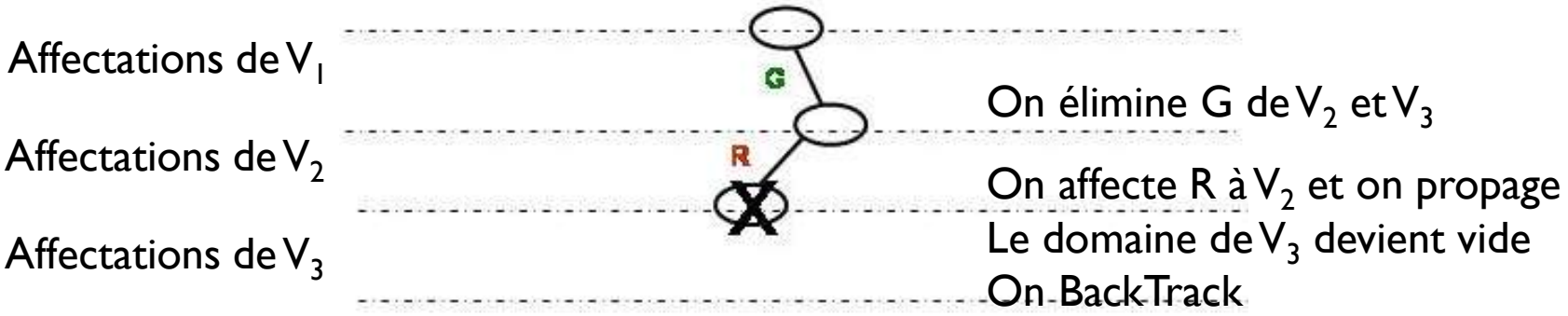
- Quand examiner une affectation $V_i = d_k$, supprimer toute valeur incohérente avec cette affectation des domaines voisins dans le graph de contraintes.

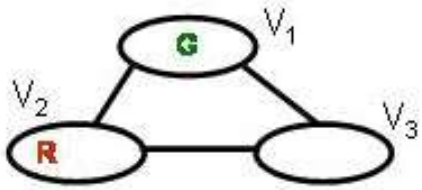


Quand on fait du chainage en arrière (BT), on doit restaurer les valeurs des domaines, puisque des suppression ont été élaborées pour arriver à une cohérence en faisant des affectation pendant la recherche.

Le Backtracking avec le Forward Checking

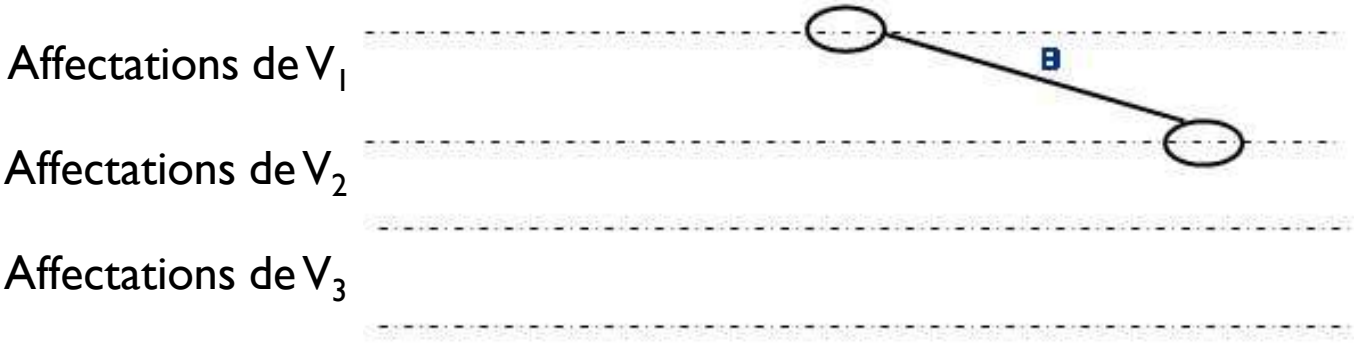
- Quand examiner une affectation $V_i = d_k$, supprimer toute valeur incohérente avec cette affectation des domaines voisins dans le graph de contraintes.

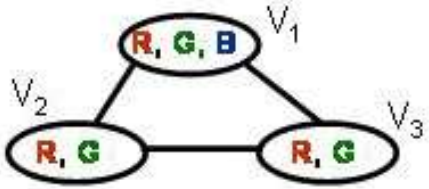




Le Backtracking avec le Forward Checking

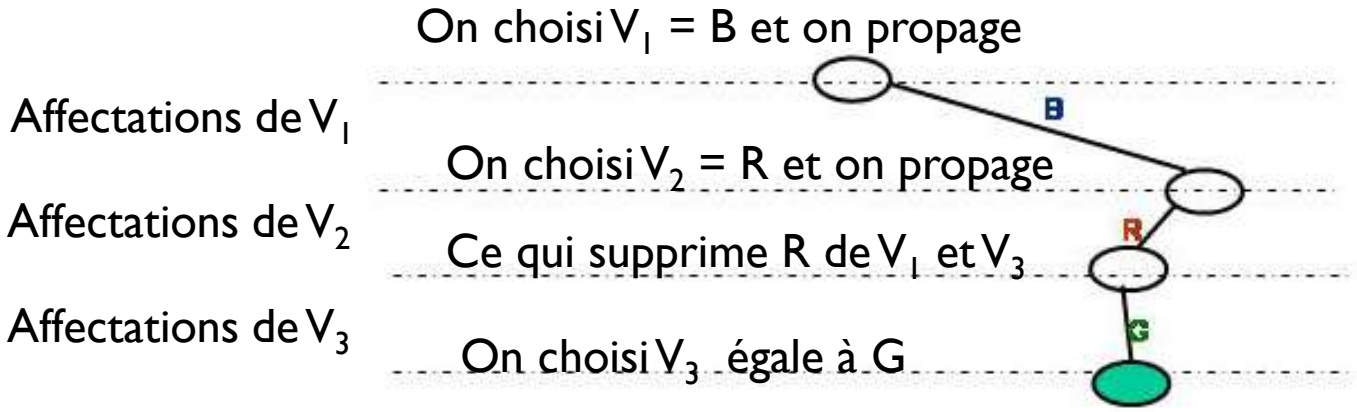
- Quand examiner une affectation $V_i = d_k$, supprimer toute valeur incohérente avec cette affectation des domaines voisins dans le graph de contraintes.



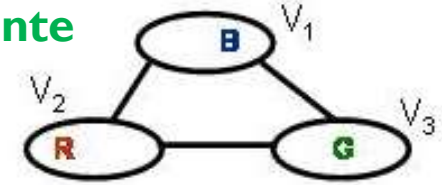


Le Backtracking avec le Forward Checking

- Quand examiner une affectation $V_i = d_k$, supprimer toute valeur incohérente avec cette affectation des domaines voisins dans le graph de contraintes.

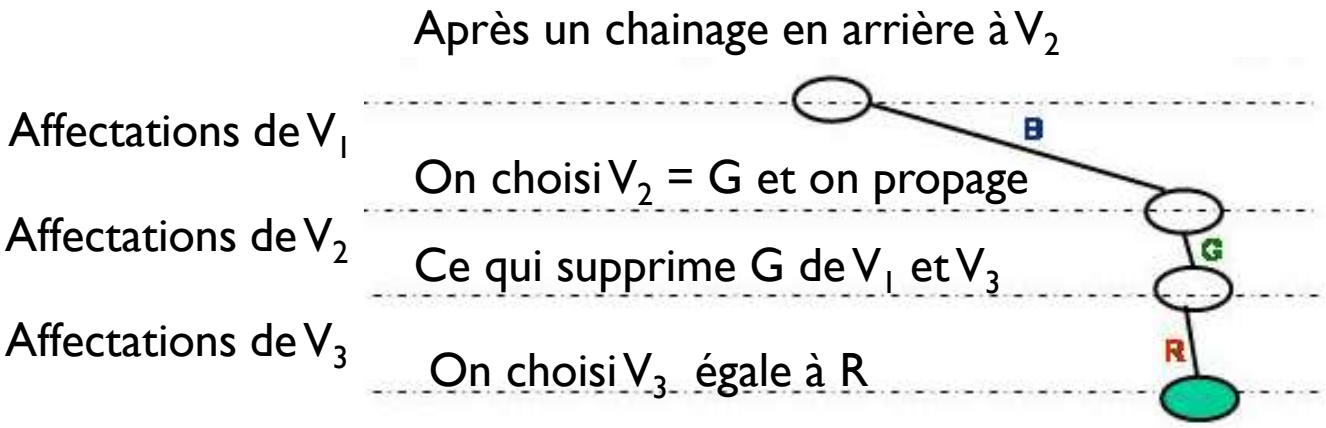


Une affectation cohérente

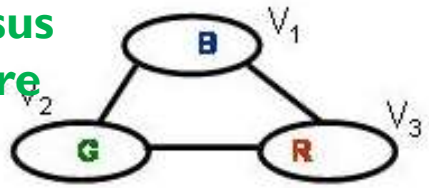


Le Backtracking avec le Forward Checking

- Quand examiner une affectation $V_i = d_k$, supprimer toute valeur incohérente avec cette affectation des domaines voisins dans le graph de contraintes.



On continue le processus Pour chercher une autre solution



BT-FC avec ordre dynamique

- Le 'Backtracking' traditionnel utilise un ordre des variables et des valeurs fixés, ex; l'ordre est aléatoire ou placer les variables avec plusieurs contraintes en premier.

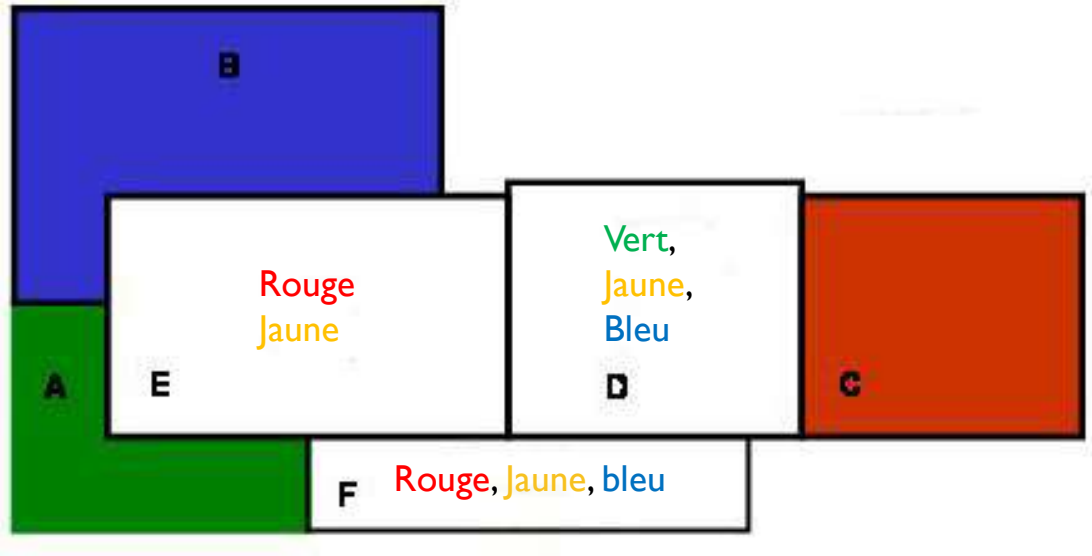
- On peut faire mieux en choisissant un ordre dynamique pendant la recherche.

- **Les variables avec le plus de contraintes**
  - Pendant la vérification à l'avance ('Forward Checking'), choisir la variable avec le moins de valeur à assigner (minimiser le facteur de branchement)
  - Si Echec, arrive plus vite

- **Les valeurs avec le moins d'impacts**
  - Choisir la valeur qui a le moins d'impact sur les domaines des voisins (une valeur qui élimine le moins de valeurs des voisins)
  - Minimiser la probabilité de l'échec

Exemple

- Couleurs: R,V, B, J; (A:V); (B:B); (C:R)



- Quel pays colorier après? → ? représente la variable avec le plus de contraintes (plus petit domaine)
- Quelle couleur choisir ? → ? est la valeur avec le moins d'impacts pour les voisins (élimine moins de valeurs pour ?et ?)

GSAT avec recherche heuristique

- Espace d'état : Espace des affectations complètes des variables
- Etat initial : Une affectation totale aléatoire
- Etat Objectif : Une affectation qui satisfait les contraintes
- Actions : Changer une valeur d'une variable dans l'affectation courante
- Heuristique : Le nombre de clauses satisfaites (contraintes); (que l'on veut maximiser), ou minimiser le nombre des clauses insatisfaites (contraintes).

GSAT

- For i=1 to **MaxEssais**
  - Sélectionner une affectation aléatoire A
  - Score = Nombre de clauses satisfaites
  - For j=1 to **MaxChangements**
    - If (A satisfait toutes les clauses dans F) return A
    - Else changer une variable qui maximise le score
    - Changer une variable choisi aléatoirement si aucun changement de variables n'augmente le score

Recherche sur un arbre de jeu

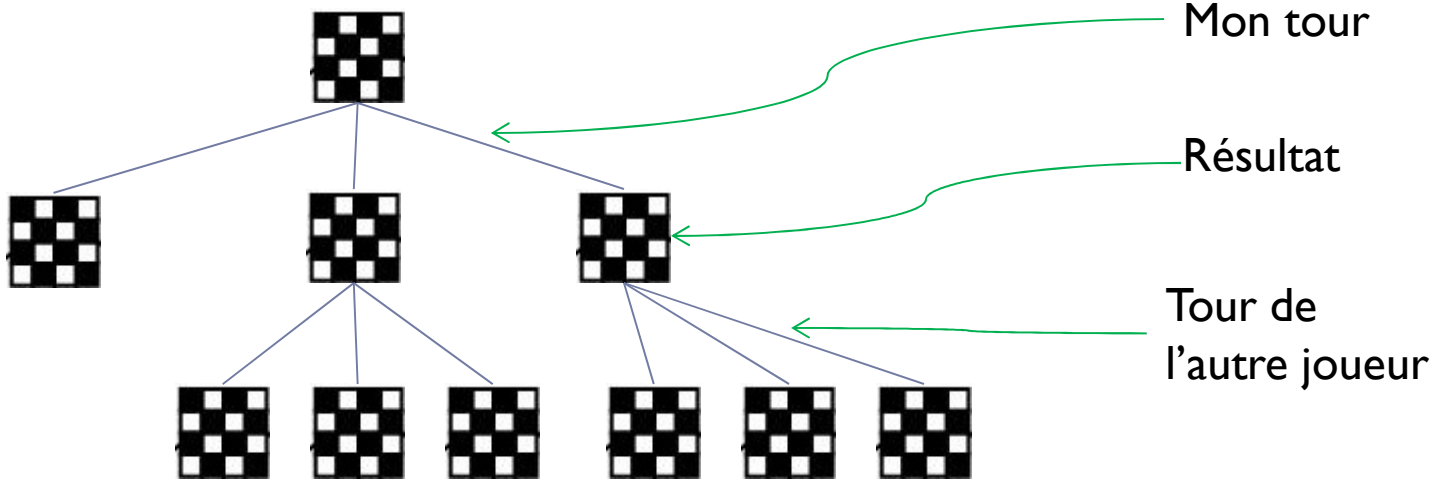
- Etat initial : Position initiale du jeu et du joueur
- Opérateurs : un opérateur pour chaque mouvement légal
- Etats Objectifs : Des positions de jeu gagnante
- Fonction de 'Scoring': affectation de valeurs numériques aux états
- Arbre de jeu: coder tous les jeux possible
- Notre objectif n'est pas la recherche d'un chemin, mais seulement le prochain tour à faire (qui peut amener à une position gagnante)
- Notre meilleur tour dépends de ce que l'autre joueur va faire

Génération de tours

b= Facteur de branchement

d= Profondeur

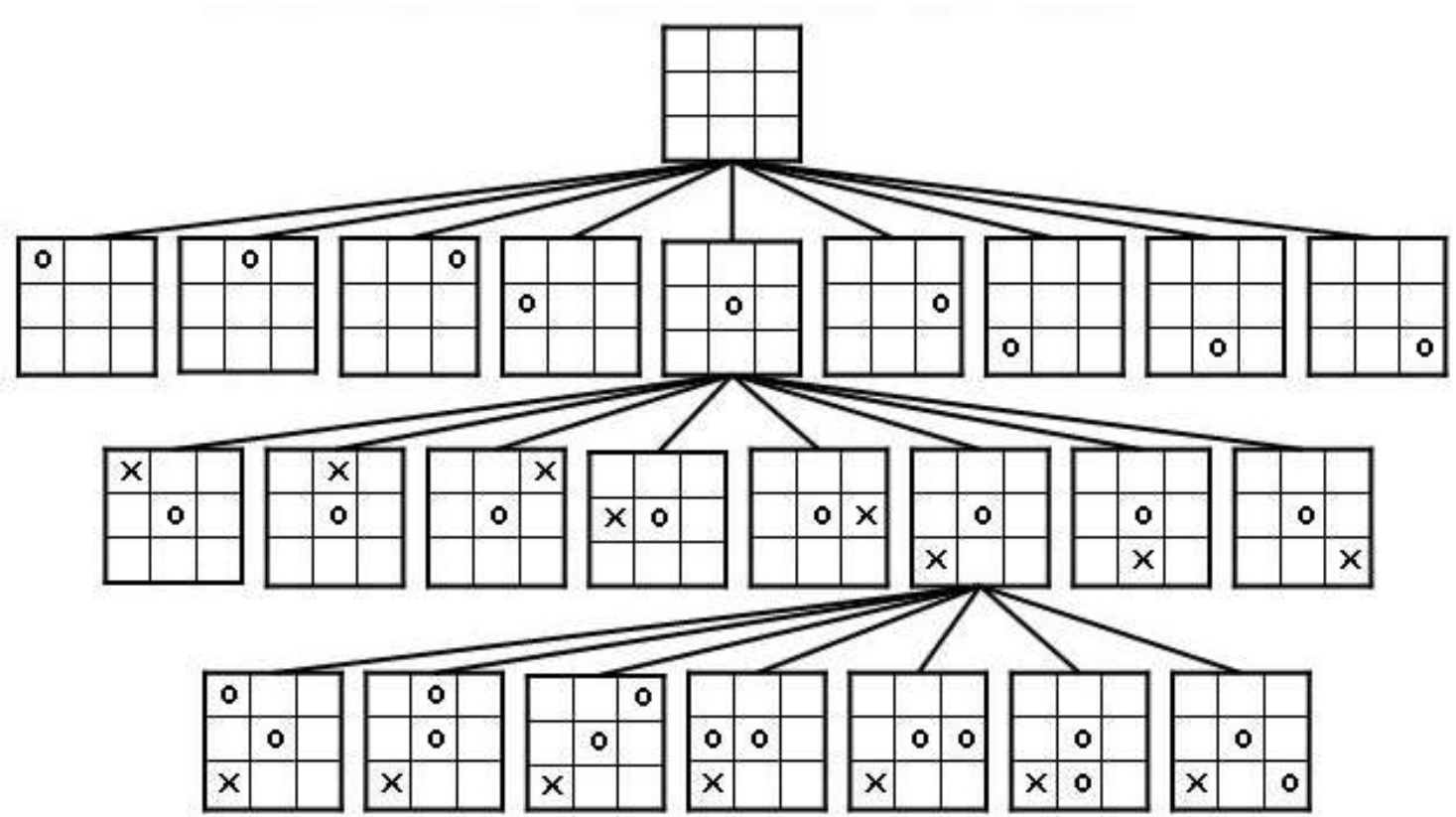
Arbre de jeu



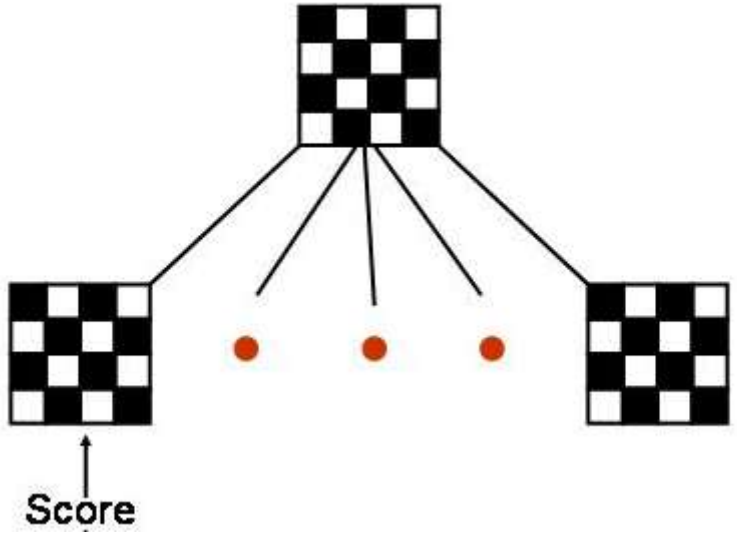
Echecs : b= 36, d> 40

36^{40} !

Arbre partiel de Tic-Tac-Toe



Fonction de 'scoring' et Evaluation statique



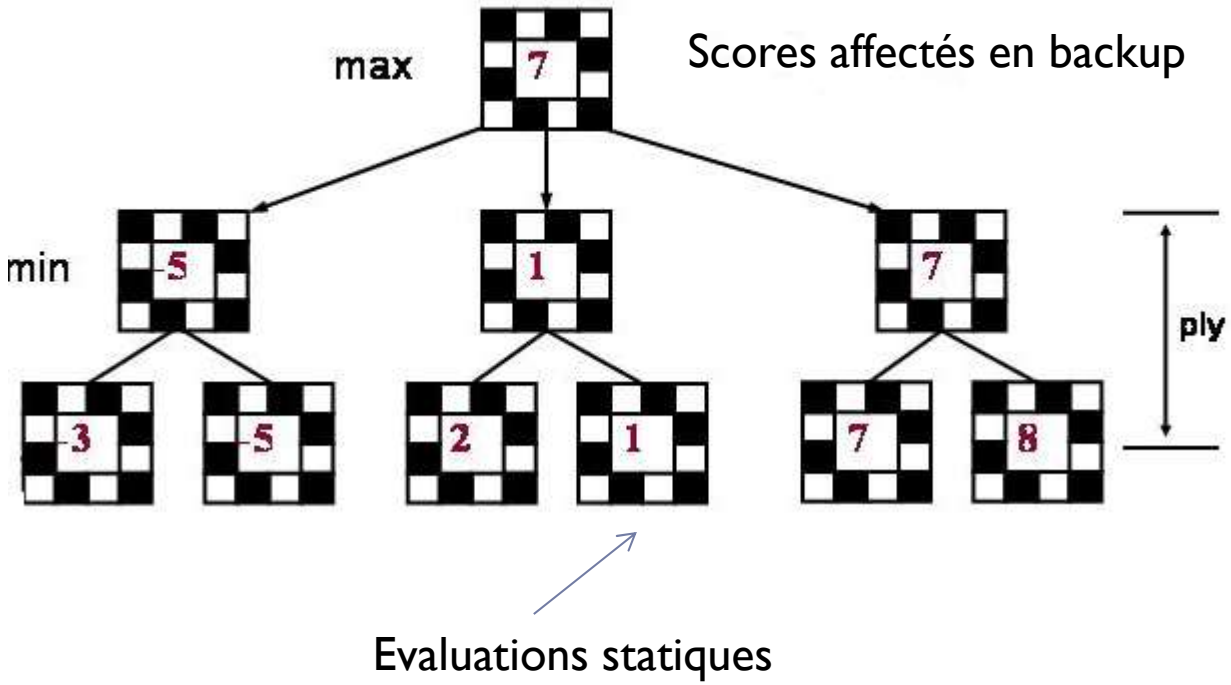
- Exemple: Echecs

Un score pour gagner à partir d'ici

- $S = C_1 * \text{Matériel} + C_2 * \text{structure des pions} + C_3 * \text{Mobilité} + C_4 * \text{Sureté du roi} + C_5 * \text{contrôle du centre}$
- Matériel (Pion = 1; Chevalier = 3; Fou =3,5; Tour = 5; Reine = 9)

Recherche à l'avance limité + Scoring

- Look ahead + scoring



Algorithme Min-Max

- # Appel initial est Max-valeur(état, profondeur)

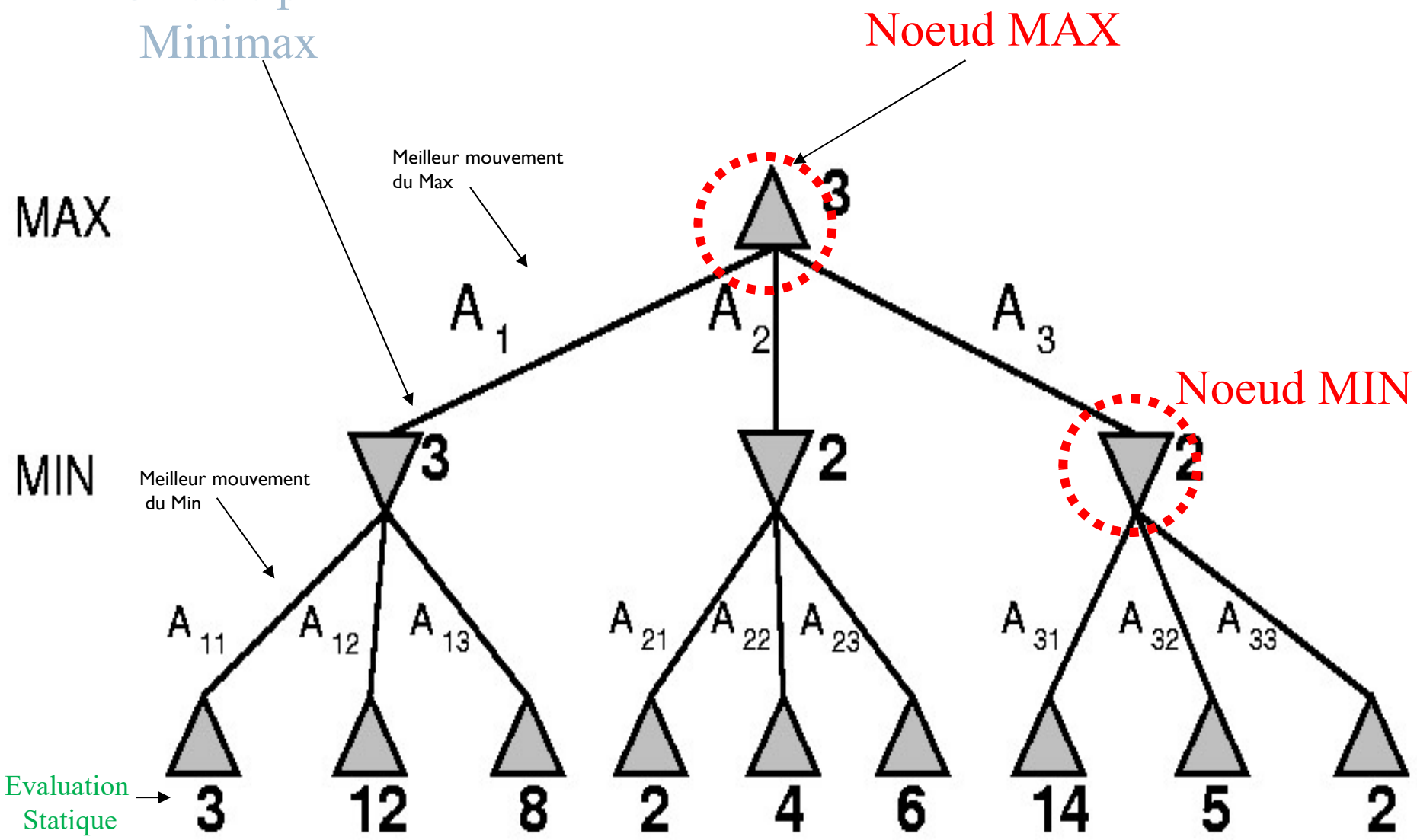
- Fonction Max-Valeur(état, profondeur)
  If (profondeur ==0) alors return EVAL(état)
  v= -∞
  For chaque s SUCCESSEURS(état) DO
    v_Succ = Min-Valeur(s, profondeur -1)
    v = Max(v; v_Succ)
  End
  return v

- Fonction Min-Valeur(etat, profondeur)
  If (profondeur ==0) alors return EVAL(état)
  v= ∞
  For chaque s SUCCESSEURS(état) DO
    v_Succ= Max-Valeur(s, profondeur -1)
    v = Min(v; v_Succ)
  End
  return v

- Affectation de valeurs heuristiques aux feuilles de l'arbre

- Propagation des valeurs

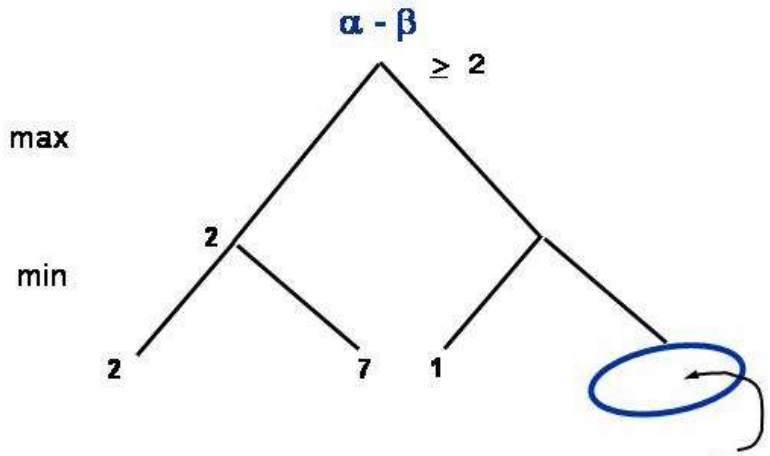
Algorithme Min-Max



Elagage Alpha-Beta (α-β pruning)

Principe :

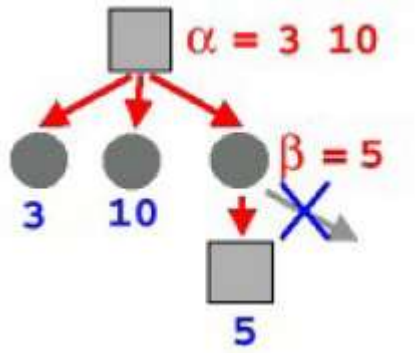
- Ne plus générer les successeurs d'un noeud dès qu'il est évident que ce noeud ne sera pas choisi, compte tenu des noeuds déjà examinés
- Chaque noeud MAX conserve la trace d'une α-valeur, qui est la valeur de son meilleur successeur trouvé jusqu'à présent
- Chaque noeud MIN conserve la trace d'une β-valeur, qui est la valeur de son pire successeur trouvé jusqu'à présent
- Valeurs initiales : $\alpha = -\infty$ et $\beta = +\infty$
- α = limite inférieure du score et β = limite supérieure du score



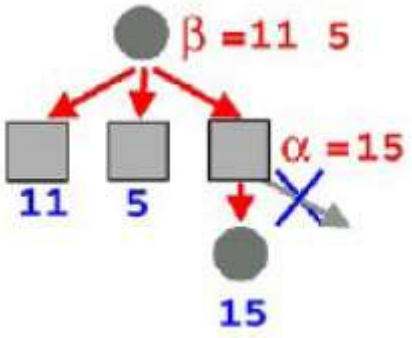
Elagage Alpha-Beta (α-β pruning)

Règles :

- Interrompre la recherche d'un noeud MAX si son α-valeur >= β-valeur de son nœud parent.



- Interrompre la recherche d'un noeud MIN si son β-valeur <= α-valeur de son nœud parent.



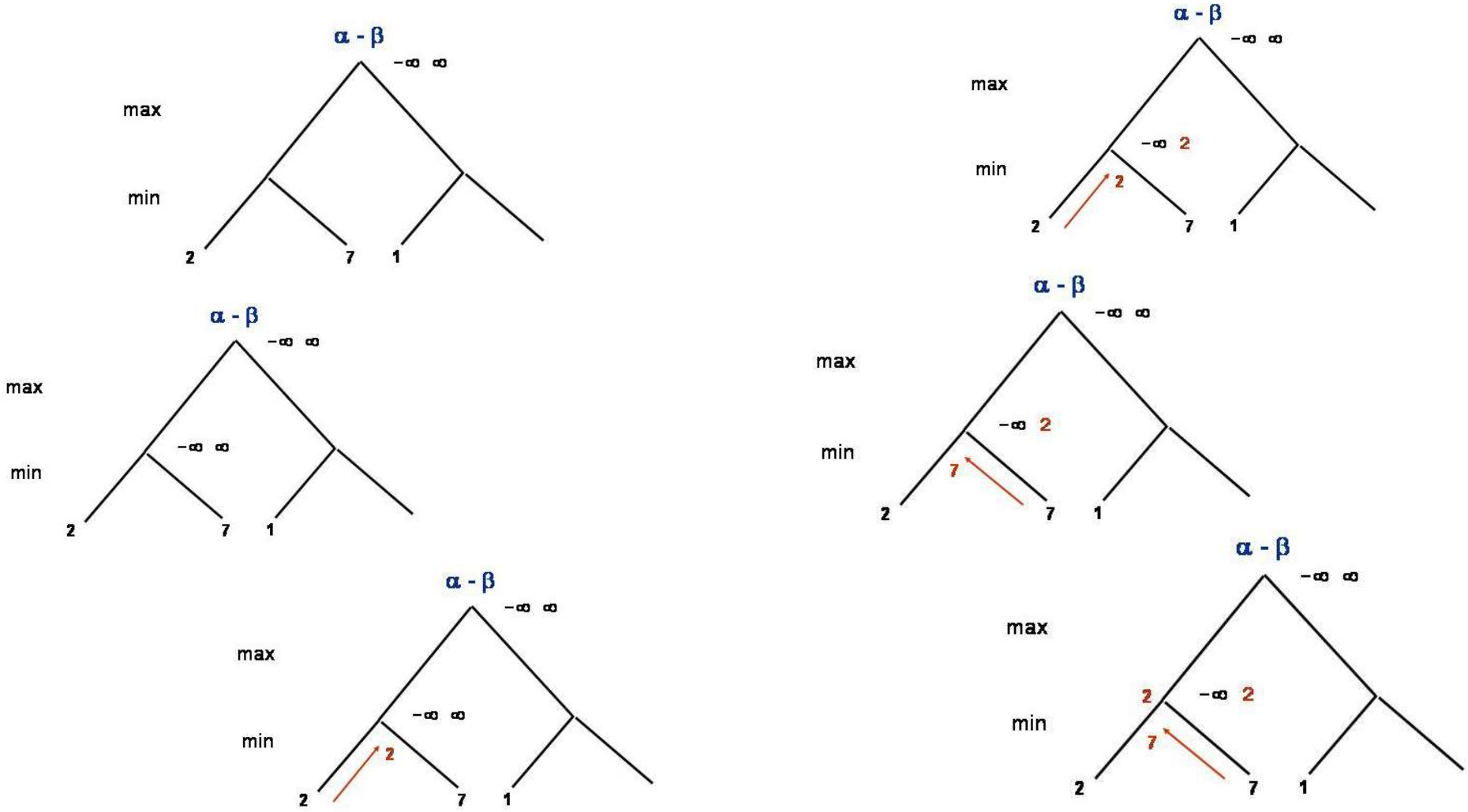
Algorithme Min-Max avec (α-β pruning)

- //α = meilleur score pour MAX, β meilleur score pour MIN
- // Appel initial est Max-valeur(état, - ∞, ∞, Max_profondeur)

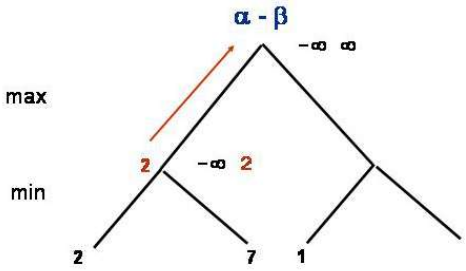
- Fonction Max-Valeur(état, α , β ,profondeur)
  If (profondeur ==0) then return EVAL(état)
  For chaque s SUCCESSEURS(état) DO
  α = MAX(α; Min-Valeur(s, α , β , profondeur -1)
  If α>= β then return α // Coupure
  End
  return α

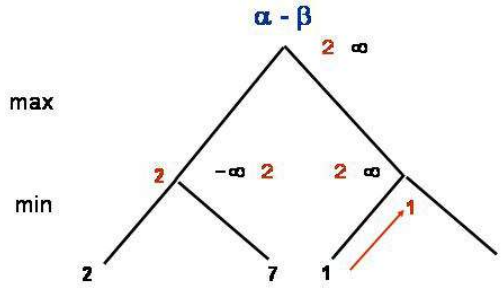
- Fonction Min-Valeur(état, α , β ,profondeur)
  If (profondeur ==0) alors return EVAL(état)
  For chaque s SUCCESSEURS(état) DO
  β = MIN(β; Max-Valeur(s, α , β , profondeur -1)
  If β <=α then return β // Coupure
  End
  return β

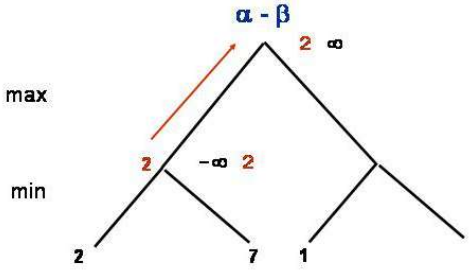
Algorithme Min-Max avec (α-β pruning)

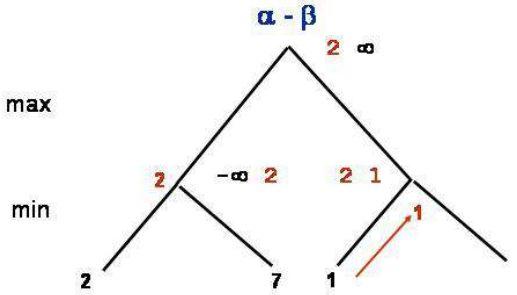


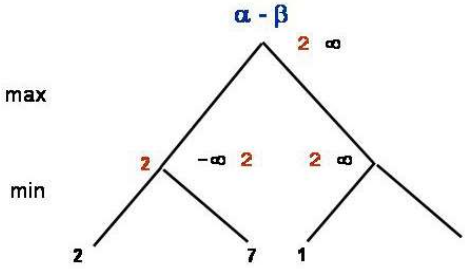
Algorithme Min-Max avec (α-β pruning)

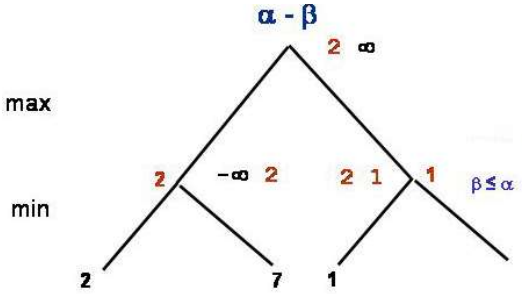












Intelligence Artificielle

- Introduction à l'intelligence artificielle
- Historique de l'IA
- Problèmes de recherche
  - Recherche dans les graphes
  - CSP (Problèmes de Satisfaction de Contraintes)
- Représentation de la connaissance (Knowledge) et Inférence
  - Logique propositionnelle et 1ière degré
  - Système basé sur les règles