

Coloration de graphe, backtracking, branch and bound

par Aymeric ‘Bébert’ Bernard

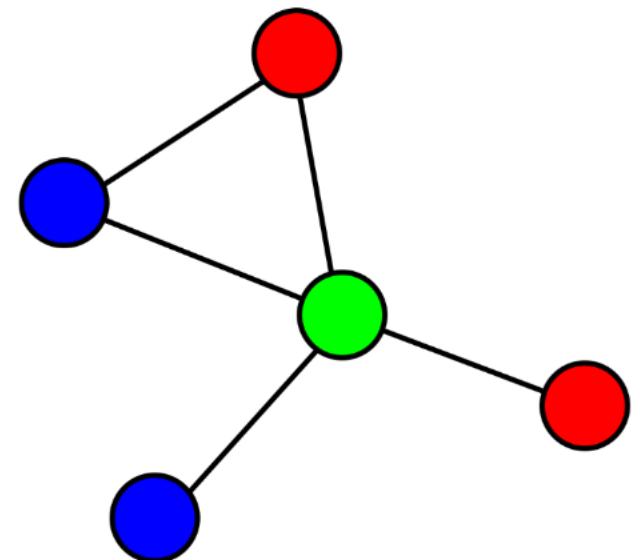
Coloration de graphe

Coloration de graphe : définitions

Comment colorer un graphe de telle sorte que deux nœuds adjacents soient de couleur différente ?

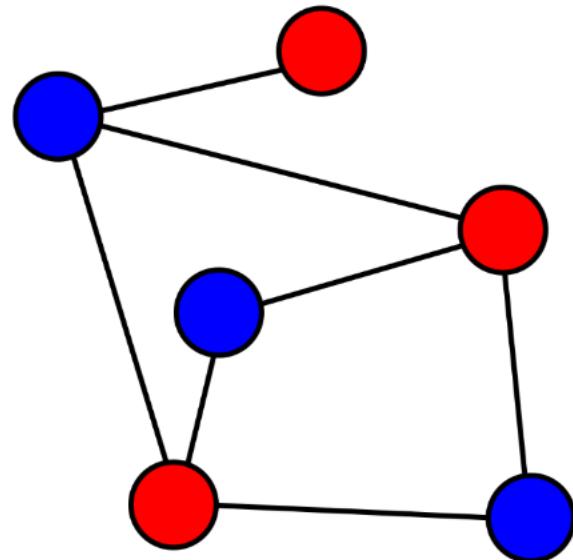
Couleur : nombre entier, généralement entre 0 et $n - 1$

$\chi(G)$: nombre minimal de couleurs pour colorer
 G



Coloration de graphe : graphe biparti

Il existe une partition des sommets du graphe en deux ensembles G₁, G₂ telle que toute arête relie un sommet de G₁ à un sommet de G₂

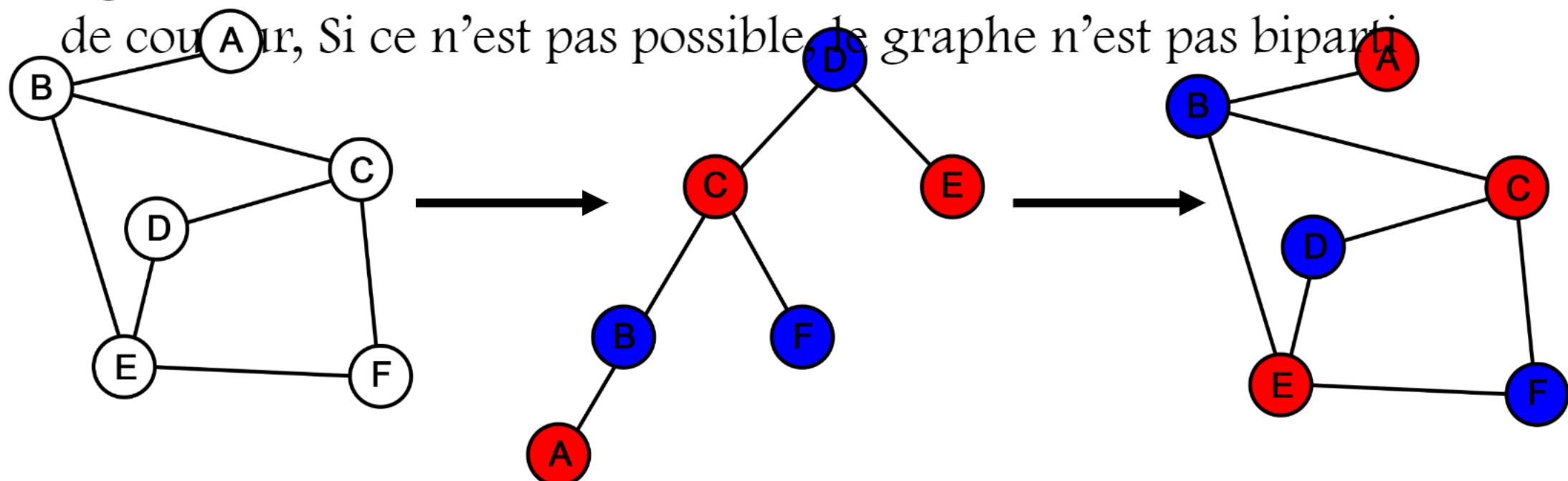


Graphe biparti : $\chi(G) = 2$ (graphe colorable avec 2 couleurs)

Coloration de graphe : graphe biparti

Comment savoir si un graphe est biparti ?

→ On parcourt le graphe, et à chaque changement de profondeur on change

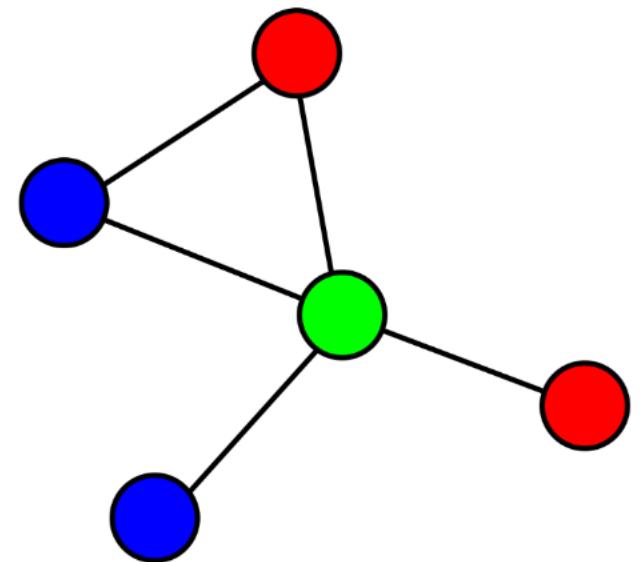


Coloration de graphe : comment faire ?

Quel algorithme permet de colorer un graphe ?
Quid de l'optimalité ?

Deux algorithmes principaux :

- Algorithme glouton, naïf, non optimal mais rapide
- Backtracking, complexité élevée mais permet d'avoir un résultat optimal



Coloration de graphe : algorithme glouton

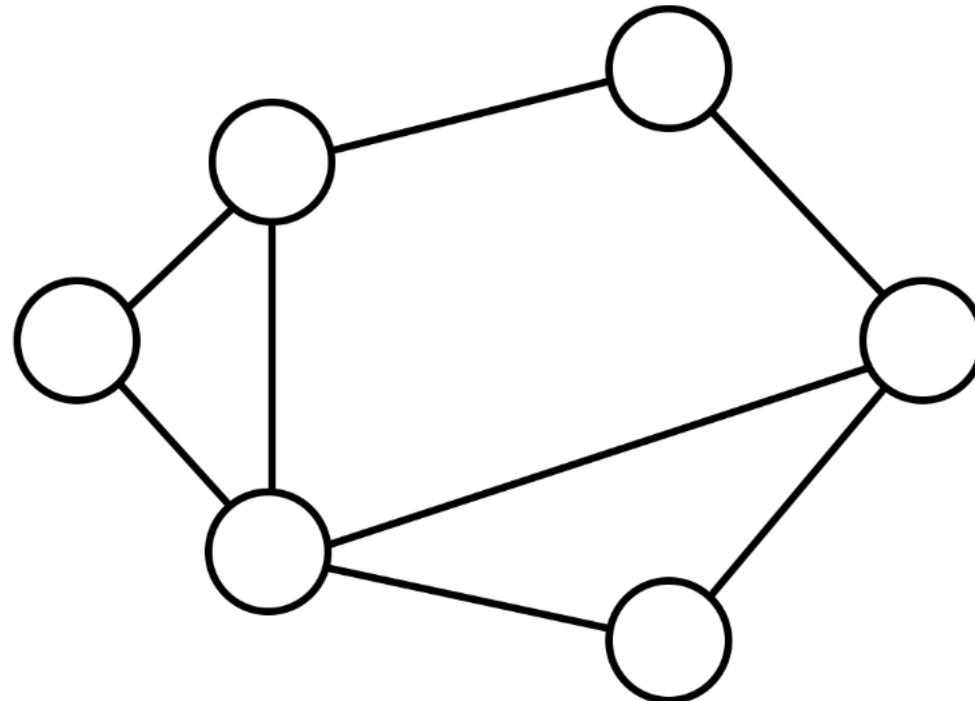
Algorithme glouton : avance étape par étape, choisit une solution optimale localement, sans souci d'optimalité globale.

Principe : on parcourt le graphe, et à chaque nœud on assigne la première couleur possible (en fonctions des voisins déjà colorés).

Problème : le nombre de couleurs dépend du nœud de départ, de la manière de parcourir le graphe...

Coloration de graphe : algorithme glouton

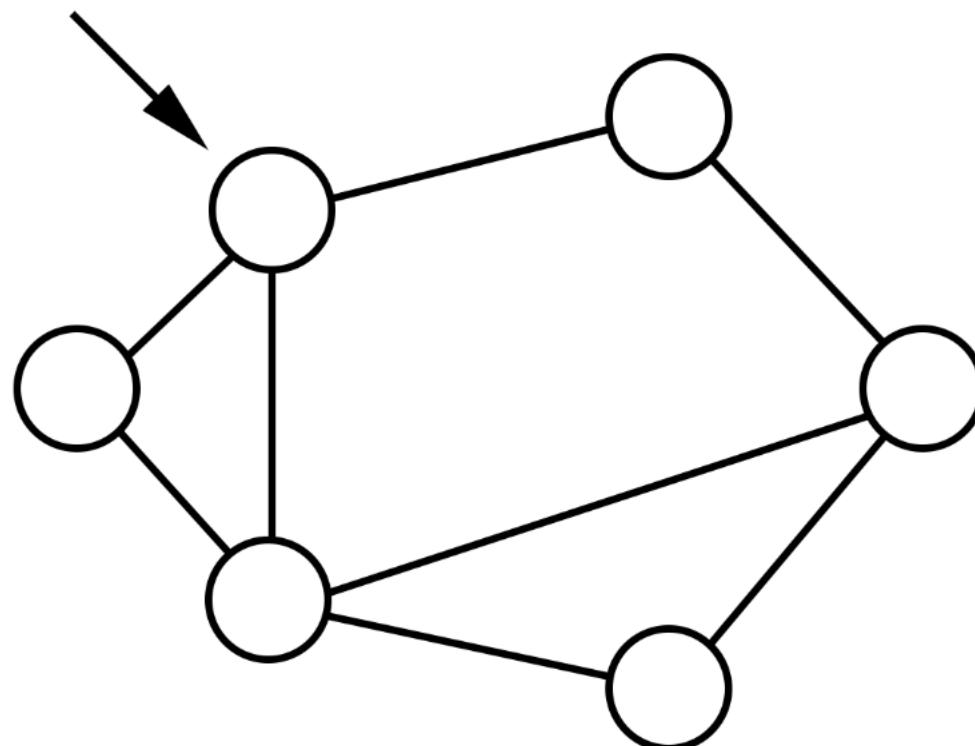
Exemple :



Couleurs utilisées :

Coloration de graphe : algorithme glouton

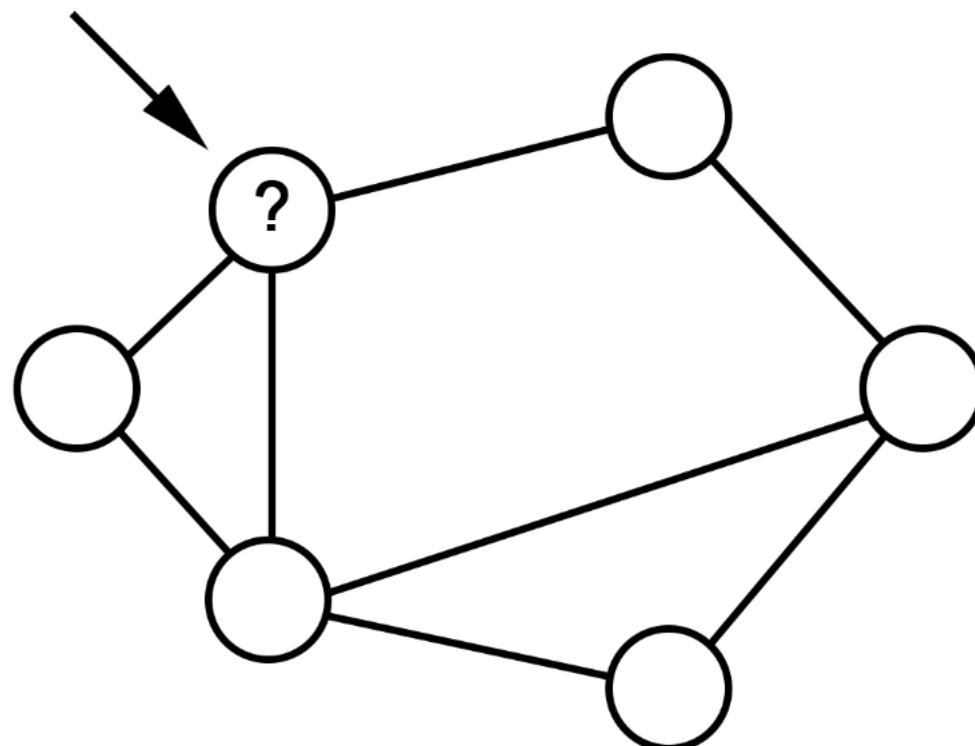
Exemple :



Couleurs utilisées :

Coloration de graphe : algorithme glouton

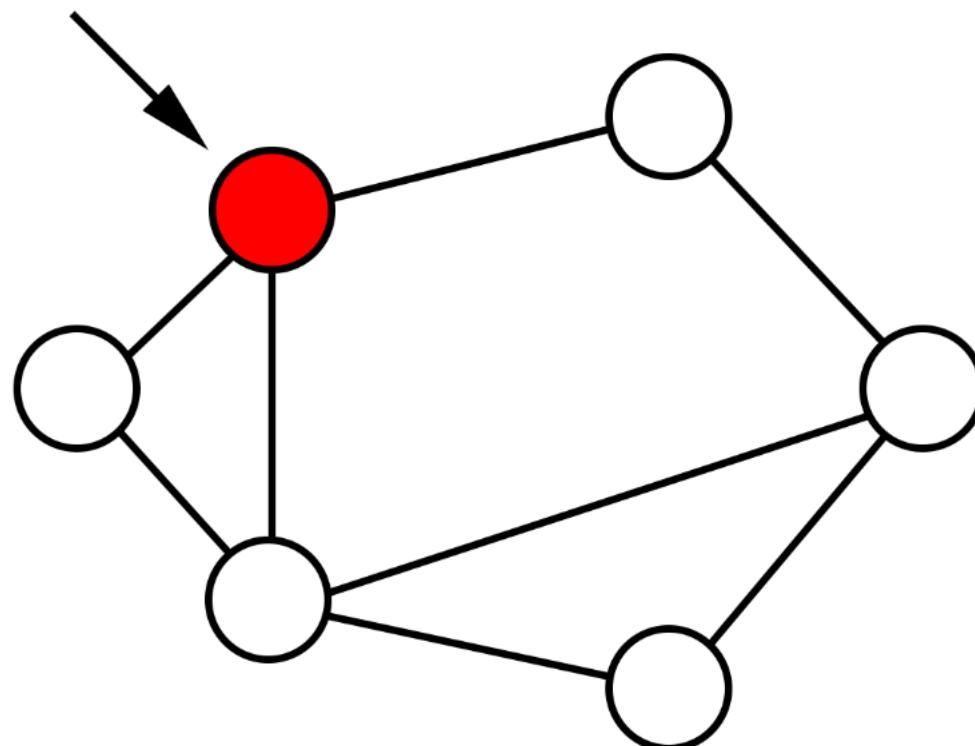
Exemple :



Couleurs utilisées :

Coloration de graphe : algorithme glouton

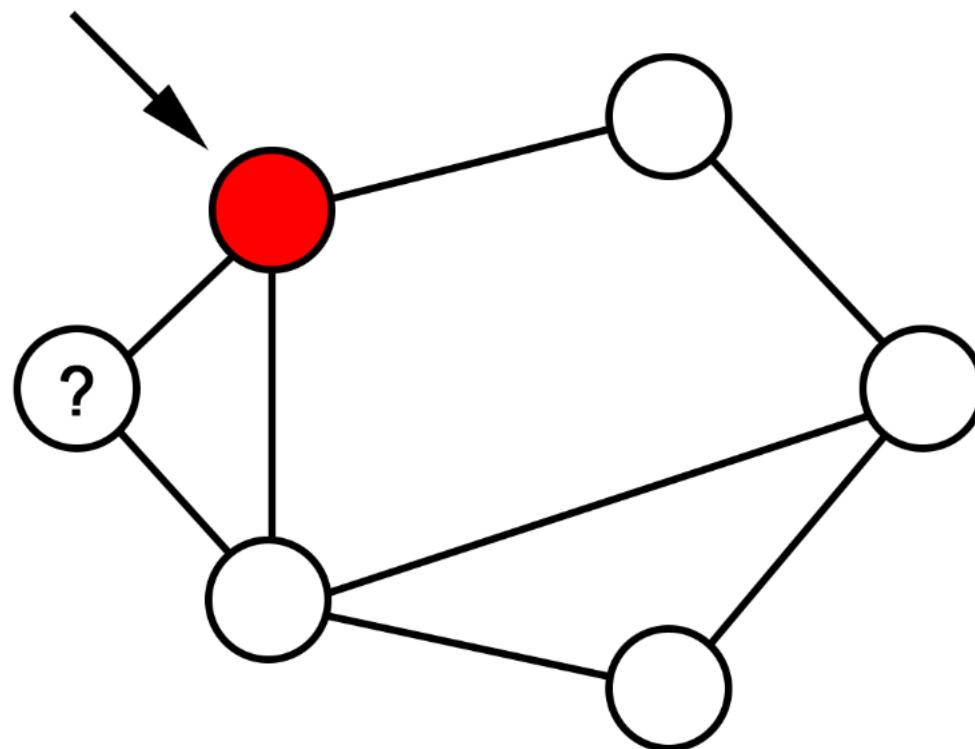
Exemple :



Couleurs utilisées : 0

Coloration de graphe : algorithme glouton

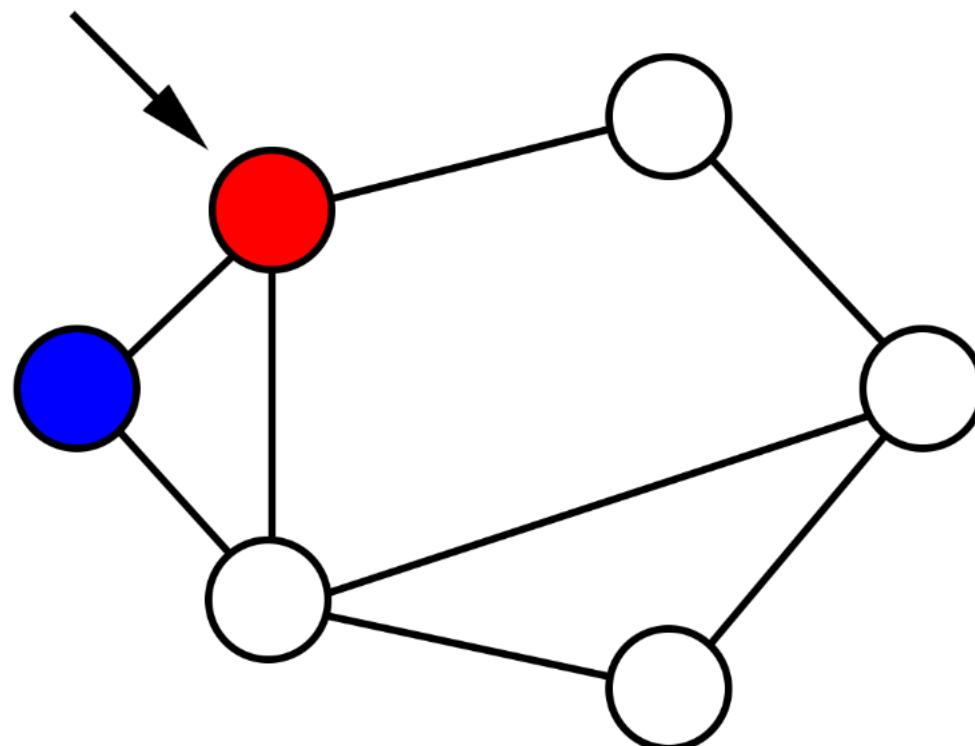
Exemple :



Couleurs utilisées : 0

Coloration de graphe : algorithme glouton

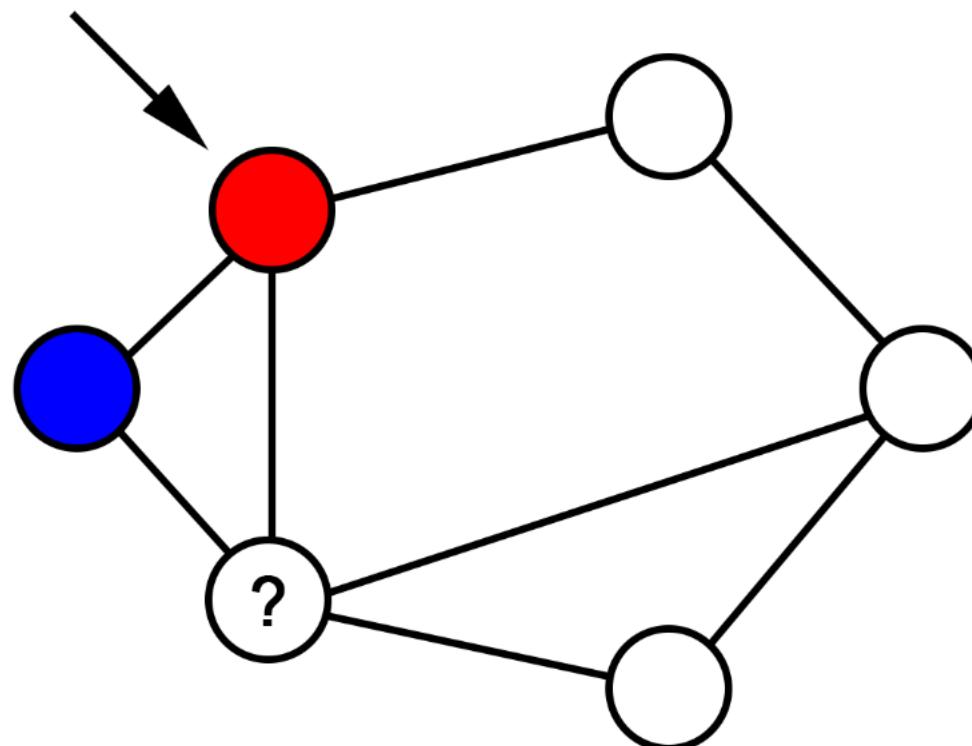
Exemple :



Couleurs utilisées : 0, 1

Coloration de graphe : algorithme glouton

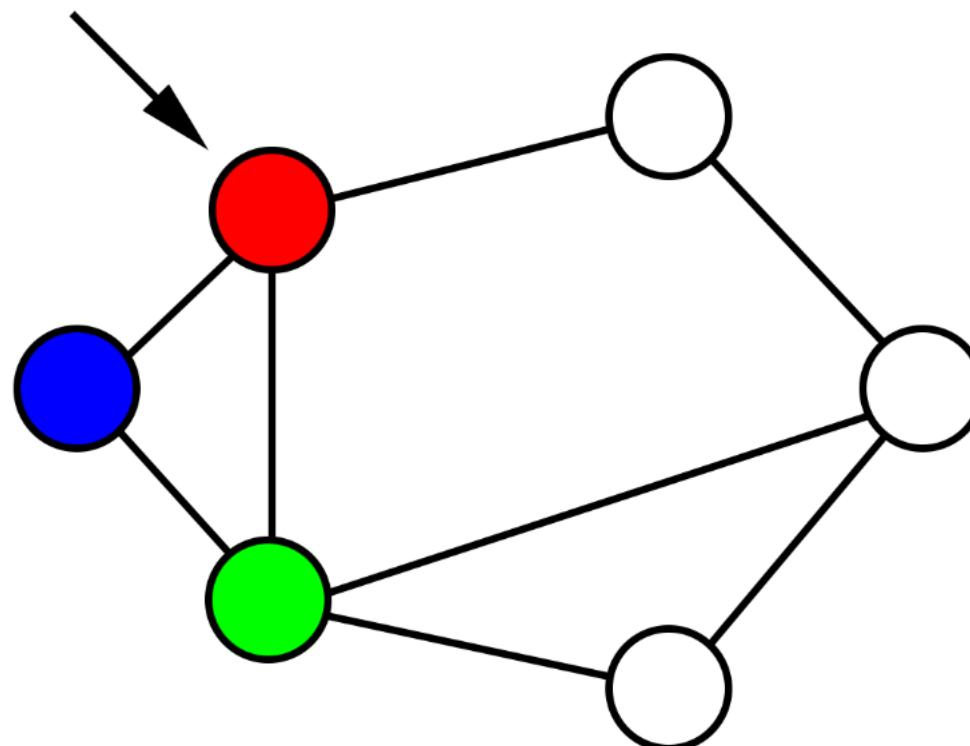
Exemple :



Couleurs utilisées : 0, 1

Coloration de graphe : algorithme glouton

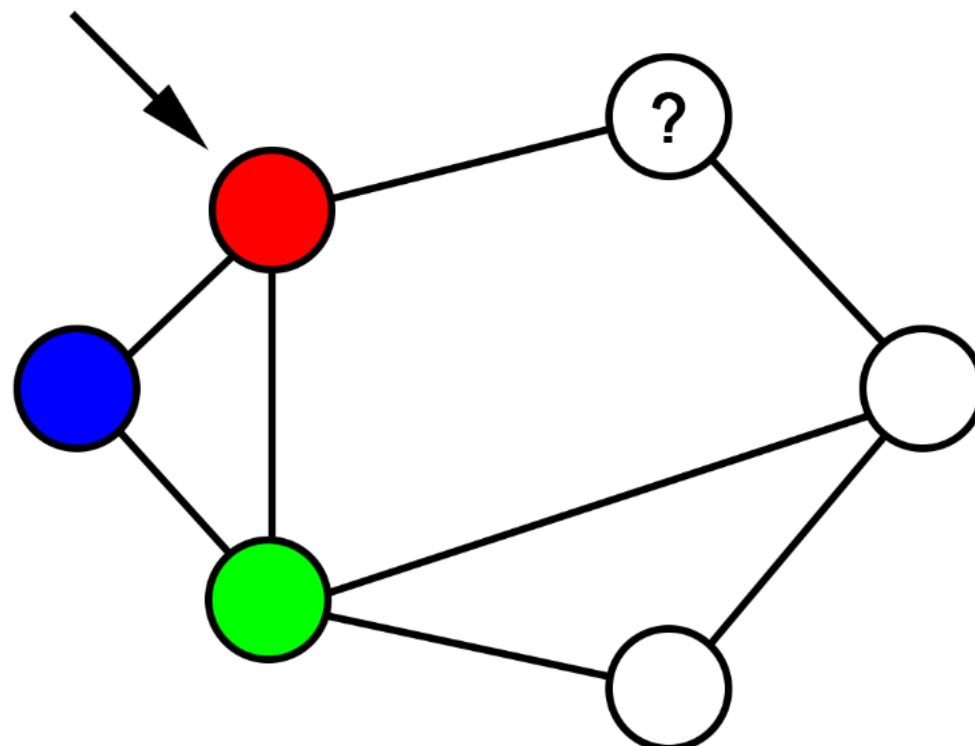
Exemple :



Couleurs utilisées : 0, 1, 2

Coloration de graphe : algorithme glouton

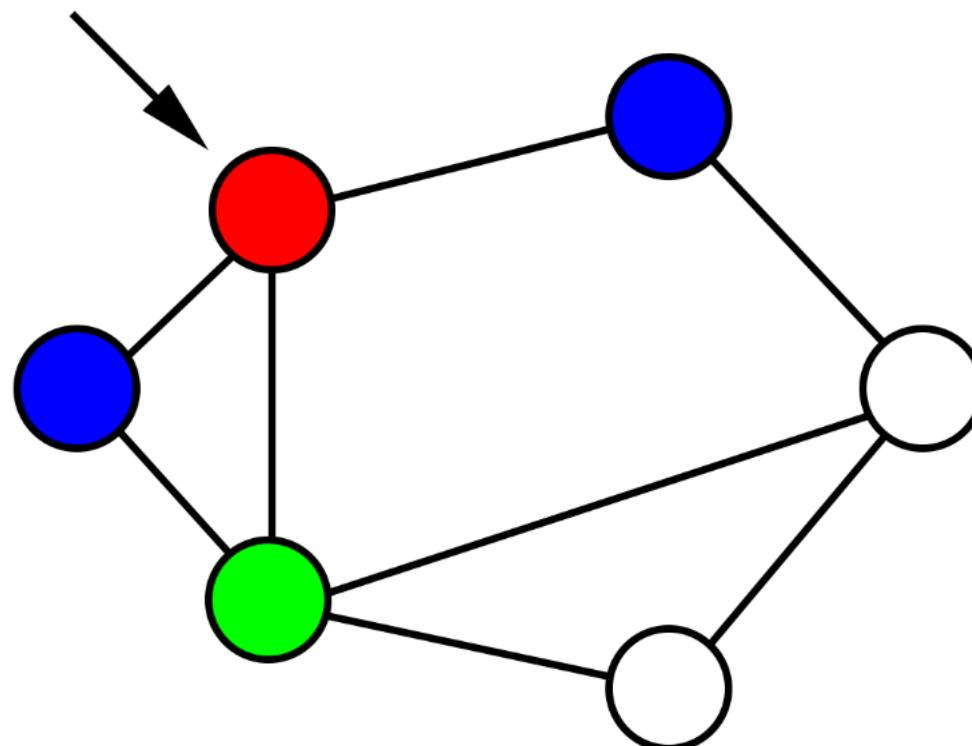
Exemple :



Couleurs utilisées : 0, 1, 2

Coloration de graphe : algorithme glouton

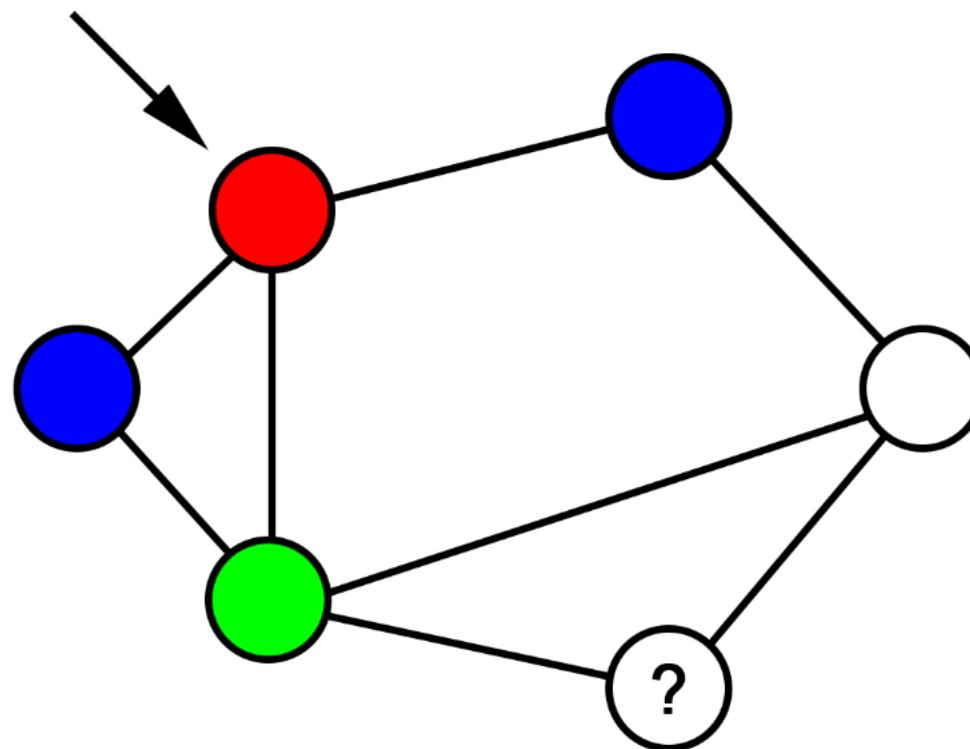
Exemple :



Couleurs utilisées : 0, 1, 2

Coloration de graphe : algorithme glouton

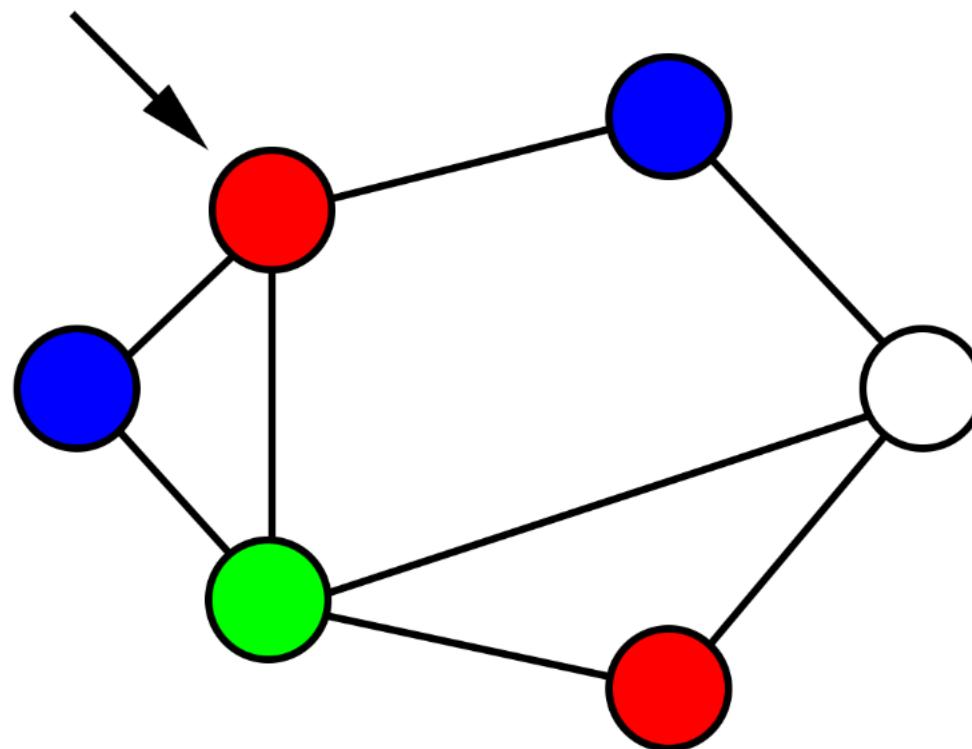
Exemple :



Couleurs utilisées : 0, 1, 2

Coloration de graphe : algorithme glouton

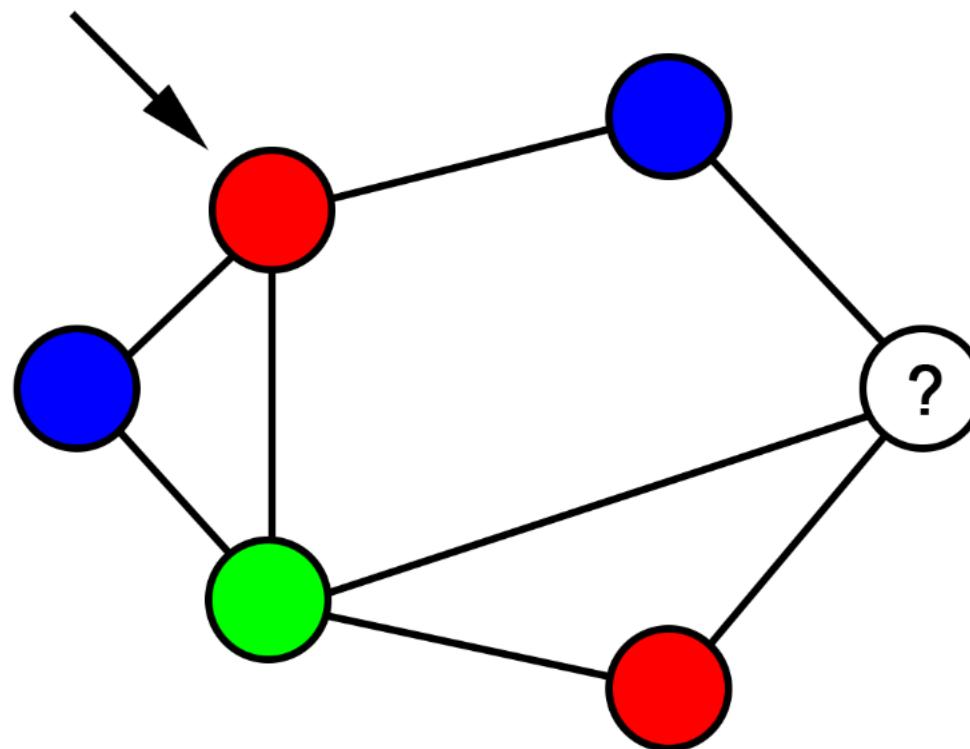
Exemple :



Couleurs utilisées : 0, 1, 2

Coloration de graphe : algorithme glouton

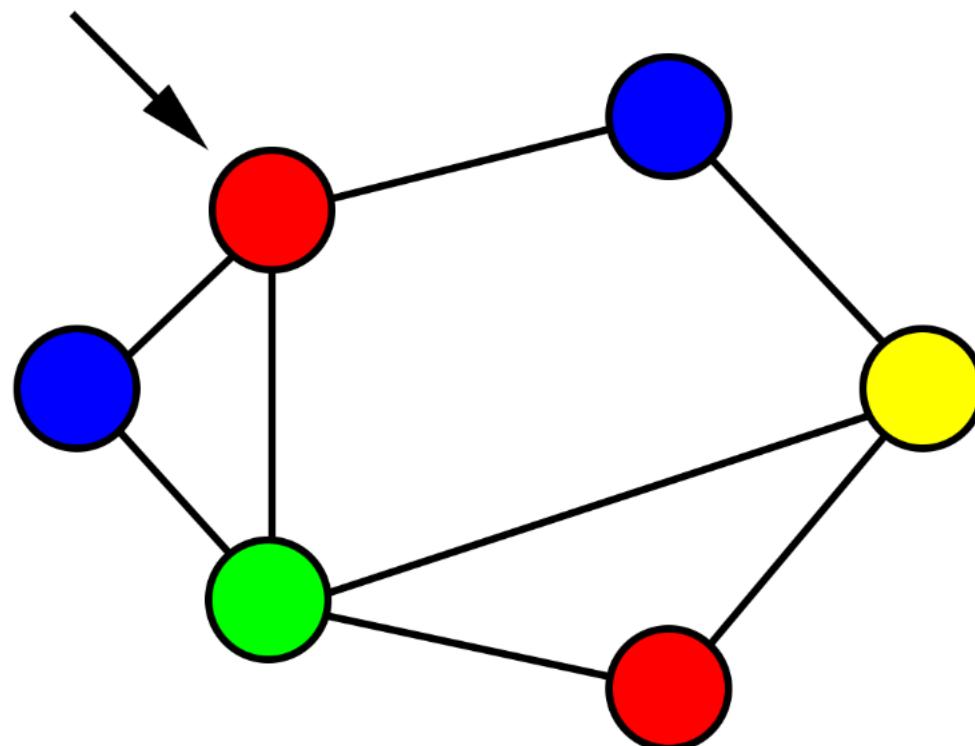
Exemple :



Couleurs utilisées : 0, 1, 2

Coloration de graphe : algorithme glouton

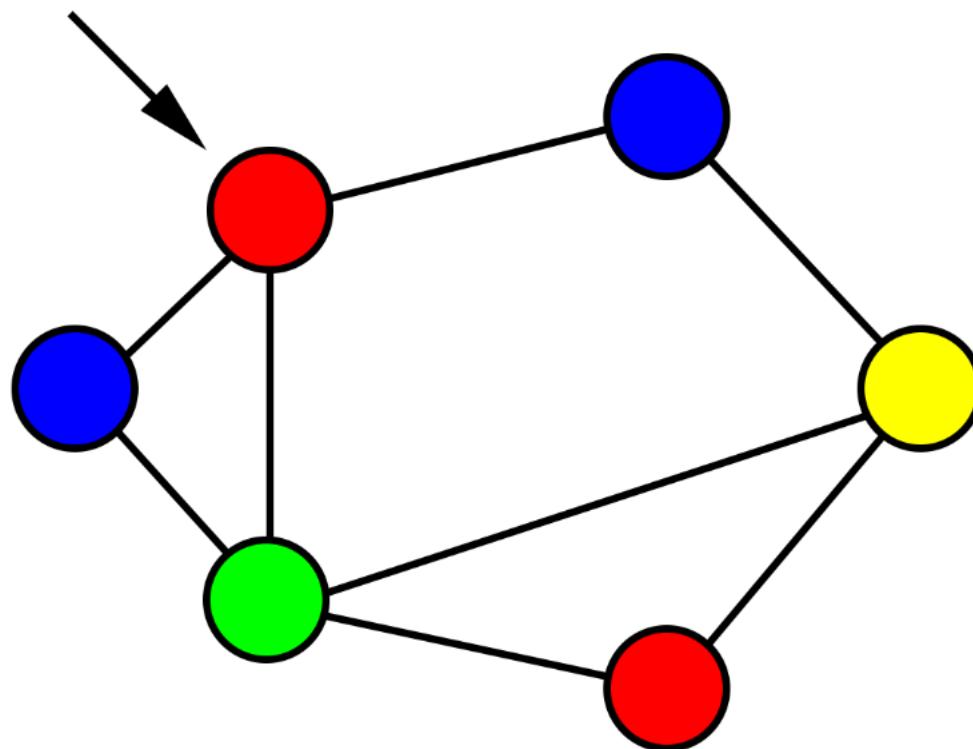
Exemple :



Couleurs utilisées : 0, 1, 2, 3

Coloration de graphe : algorithme glouton

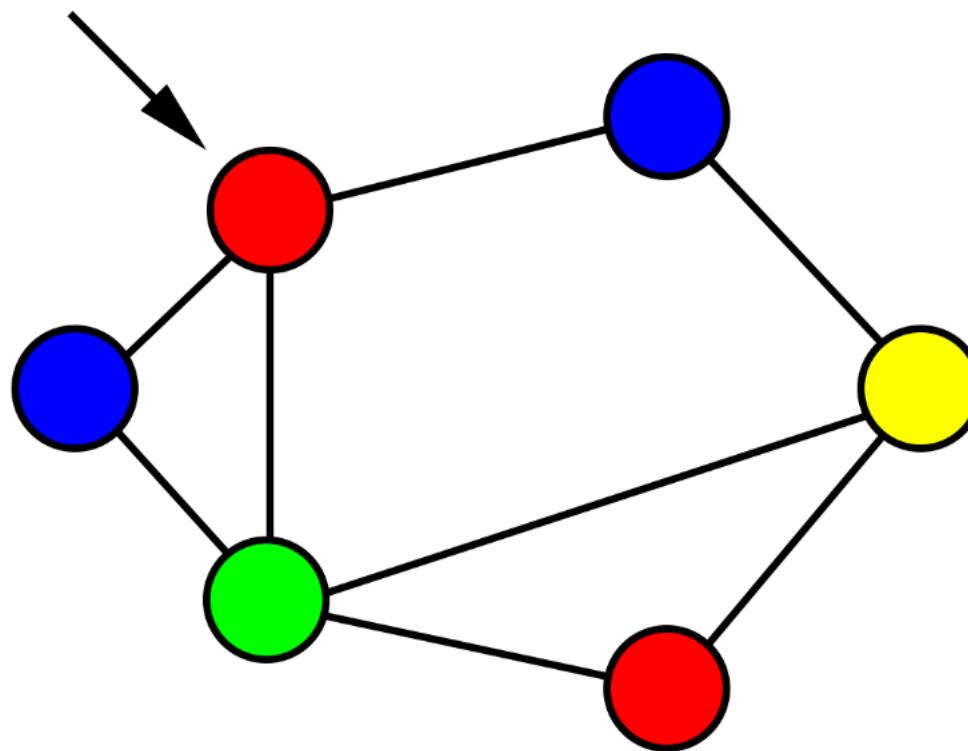
Exemple :



Couleurs utilisées : 0, 1, 2, 3 → 4 couleurs

Coloration de graphe : algorithme glouton

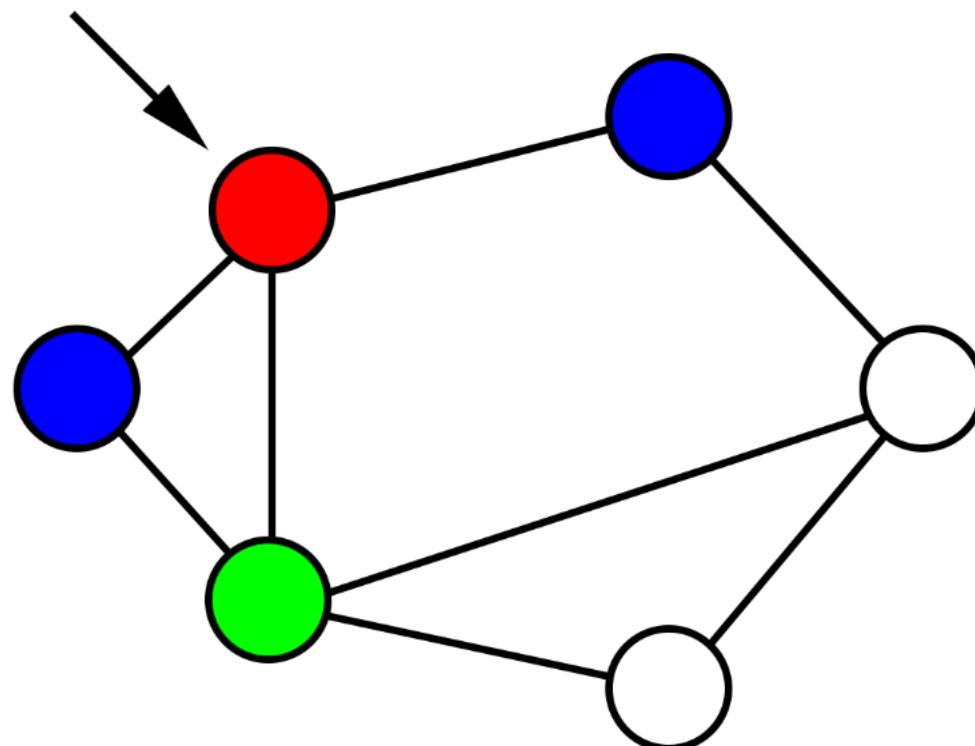
Exemple :



Et pourtant...

Coloration de graphe : algorithme glouton

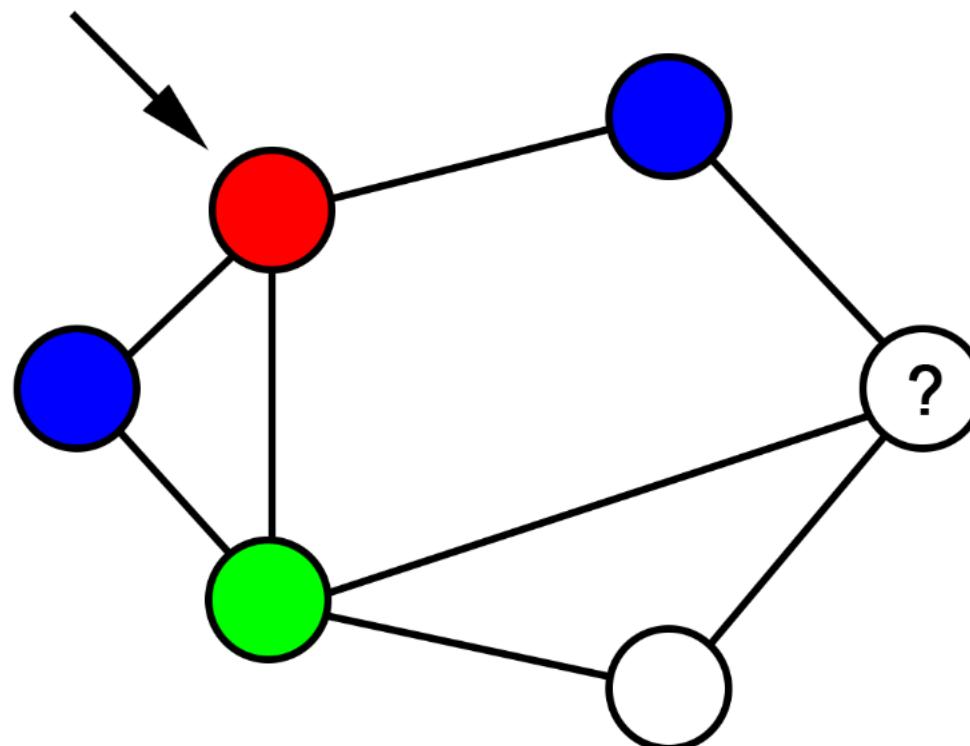
Exemple :



Couleurs utilisées : 0, 1, 2

Coloration de graphe : algorithme glouton

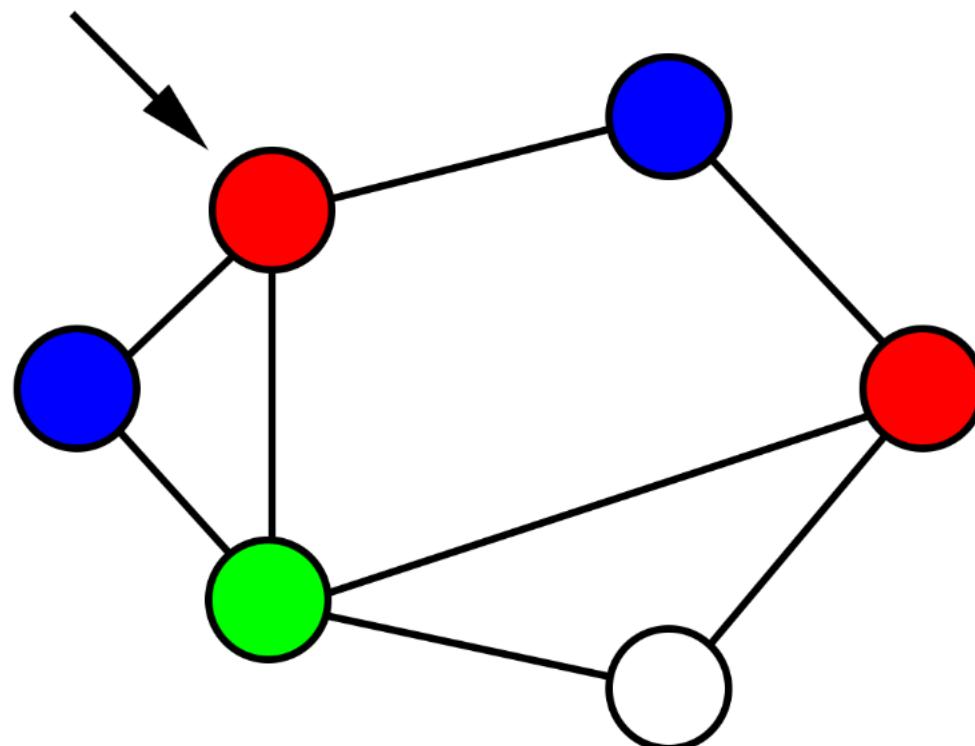
Exemple :



Couleurs utilisées : 0, 1, 2

Coloration de graphe : algorithme glouton

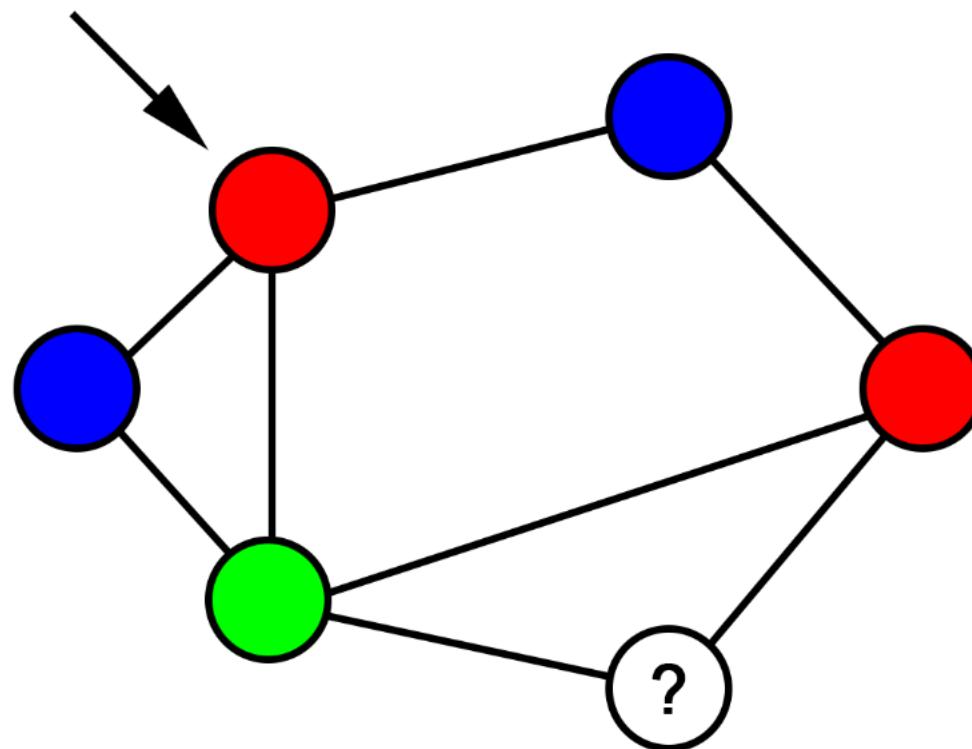
Exemple :



Couleurs utilisées : 0, 1, 2

Coloration de graphe : algorithme glouton

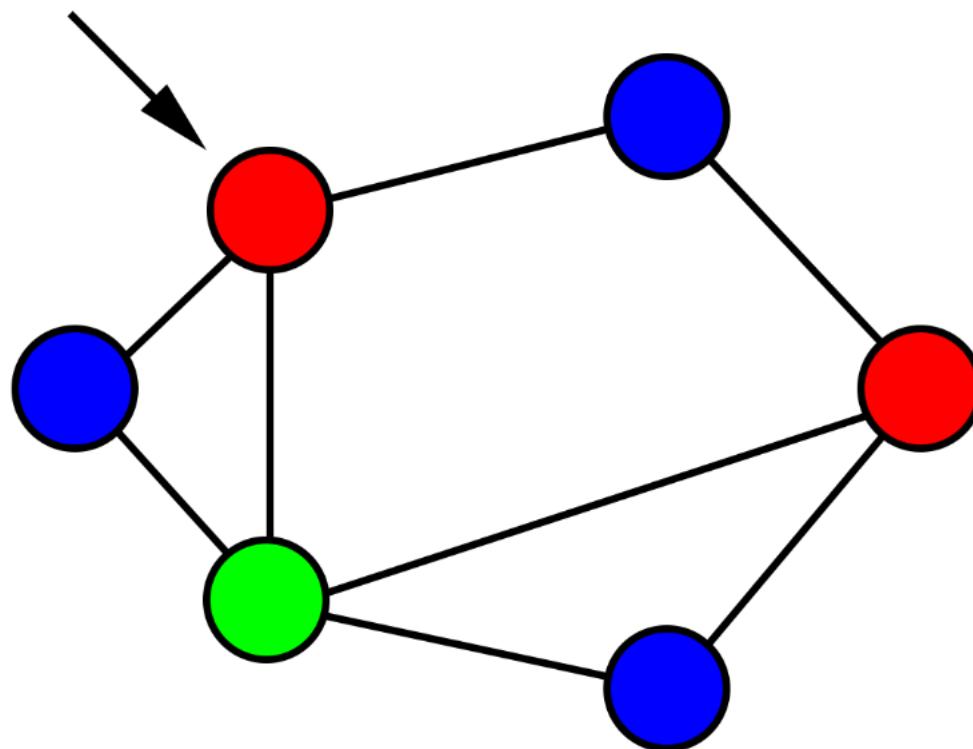
Exemple :



Couleurs utilisées : 0, 1, 2

Coloration de graphe : algorithme glouton

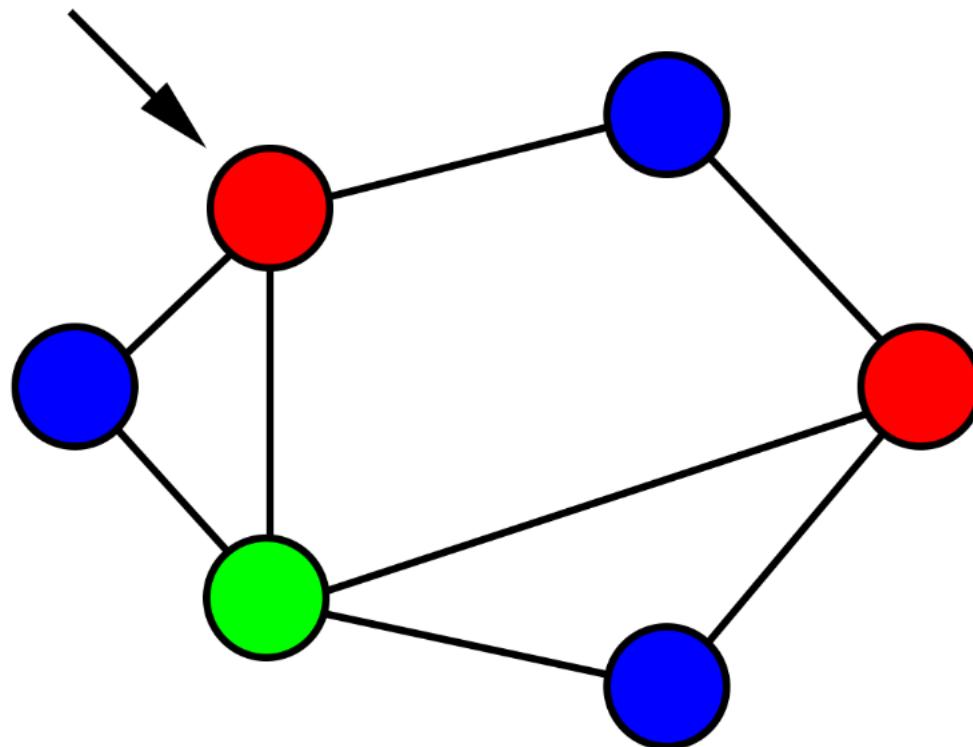
Exemple :



Couleurs utilisées : 0, 1, 2

Coloration de graphe : algorithme glouton

Exemple :



Couleurs utilisées : 0, 1, 2 → 3 couleurs

Coloration de graphe : algorithme glouton

```
function greedyColor(G):
```

```
    c ← 0
```

```
    ...
```

```
    ...
```

```
    ...
```

```
    ...
```

```
    ...
```

```
    ...
```

```
    ...
```

```
    ...
```

```
    ...
```

```
    ...
```

```
    ...
```

```
    ...
```

```
    ...
```

```
    return c
```

Coloration de graphe : algorithme glouton

```
function greedyColor(G):
```

```
    c ← 0
```

```
    for i ← 0 to n-1:
```

```
        ...
```

```
        ...
```

```
        ...
```

```
        ...
```

```
        ...
```

```
        ...
```

```
        ...
```

```
        ...
```

```
        ...
```

```
        ...
```

```
        ...
```

```
        ...
```

```
        ...
```

```
    return c
```

Coloration de graphe : algorithme glouton

Coloration de graphe : algorithme glouton

```
function greedyColor(G):
    c ← 0
    for i ← 0 to n-1:
        forbiddenColors ← tableau(c, False)
        for n in neighbors(G[i]):
            if n.color != None:
                forbiddenColors[n.color] ← True
        k ← c
        for j ← c-1 downto 0 if not(forbiddenColors[j]):
            k ← j
        ...
        ...
        ...
        ...
        ...
    return c
```

Coloration de graphe : algorithme glouton

```
function greedyColor(G):
    c ← 0
    for i ← 0 to n-1:
        forbiddenColors ← tableau(c, False)
        for n in neighbors(G[i]):
            if n.color != None:
                forbiddenColors[n.color] ← True
        k ← c
        for j ← c-1 downto 0 if not(forbiddenColors[j]):
            k ← j
        if k = c:
            c = c+1
            G[i].color ← c
        else:
            G[i].color ← k
    return c
```

Coloration de graphe : algorithme glouton

Le nombre de couleurs utilisé $\leq \Delta(G) + 1$

($\Delta(G)$ = degré de G = nombre maximal d'arêtes partant d'un nœud)

Complexité : $\Theta(|V| + |E|)$

Le backtracking

(et son application à la coloration de graphe)

Le backtracking : schéma général

Principe général : revenir légèrement en arrière sur les décisions prises en cas de blocage

Utilisé surtout dans la programmation sous contraintes :

- ~ Colorer un graphe
- ~ Problème des n dames
- ~ Résolution d'une grille de sudoku
- ~ Mise en place d'emplois du temps
- ~ ...

Le backtracking : schéma général

On veut créer un algorithme qui renvoie **True** si une solution à notre problème (avec ses contraintes) existe, et **False** sinon (variante : stocker une ou toutes les solutions trouvées et les renvoyer).

« revenir légèrement en arrière » : par récursivité, on ne crée pas de fonction que revient explicitement sur une étape précédente !

Le backtracking : schéma général

Schéma de l'algorithme :

```
function backtrack(s,d): # s situation courante, d profondeur courante  
| ...
```

Le backtracking : schéma général

Schéma de l'algorithme :

```
function backtrack(s,d):  
| -- si s est solution, on renvoie True --
```

Le backtracking : schéma général

Schéma de l'algorithme :

```
function backtrack(s,d):
| if isSolution(s):
|   | return True # ou stocker si on veut stocker toutes les solutions
| ...
| ...
```

Le backtracking : schéma général

Schéma de l'algorithme :

```
function backtrack(s,d):
| if isSolution(s):
| | return True
| -- sinon, on veut parcourir les situations de profondeur d+1 --
| ...
```

Le backtracking : schéma général

Schéma de l'algorithme :

```
function backtrack(s,d):
| if isSolution(s):
| | return True
| for n in nextSituations(s):
| | ...
| ...
...
```

Le backtracking : schéma général

Schéma de l'algorithme :

```
function backtrack(s,d):
| if isSolution(s):
| | return True
| for n in nextSituations(s):
| | -- appel récursif pour la situation n, à une profondeur d+1 --
| ...
| ...
```

Le backtracking : schéma général

Schéma de l'algorithme :

```
function backtrack(s,d):
| if isSolution(s):
| | return True
| for n in nextSituations(s):
| | if backtrack(n,d+1): # solution trouvée dans cette branche !
| | | ...
| ...
...
```

Le backtracking : schéma général

Schéma de l'algorithme :

```
function backtrack(s,d):
| if isSolution(s):
| | return True
| for n in nextSituations(s):
| | if backtrack(n,d+1):
| | | -- on fait remonter l'information dans la recursion --
| ...
| ...
```

Le backtracking : schéma général

Schéma de l'algorithme :

```
function backtrack(s,d):
| if isSolution(s):
| | return True
| for n in nextSituations(s):
| | if backtrack(n,d+1):
| | | return True
| ...
| ...
```

Le backtracking : schéma général

Schéma de l'algorithme :

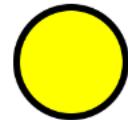
```
function backtrack(s,d):
| if isSolution(s):
| | return True
| for n in nextSituations(s):
| | if backtrack(n,d+1):
| | | return True
-- si aucune de ces situations n'est possible, renvoyer False --
```

Le backtracking : schéma général

Schéma de l'algorithme :

```
function backtrack(s,d):
| if isSolution(s):
| | return True
| for n in nextSituations(s):
| | if backtrack(n,d+1):
| | | return True
| return False
```

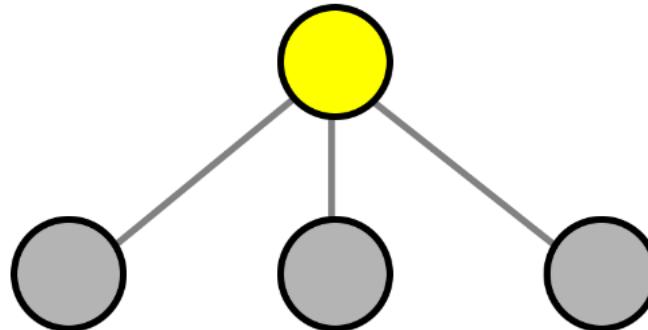
Le backtracking : illustration du principe



```
1 function backtrack(s,d):  
2 | if isSolution(s):  
3 | | return True  
4 | for n in nextSituations(s):  
5 | | if backtrack(n,d+1):  
6 | | | return True  
7 | return False
```

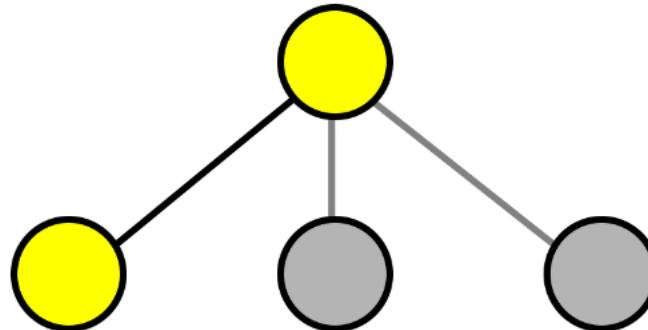
Le backtracking : illustration du principe

```
1 function backtrack(s,d):  
2 | if isSolution(s):  
3 | | return True  
4 | for n in nextSituations(s):  
5 | | if backtrack(n,d+1):  
6 | | | return True  
7 | return False
```



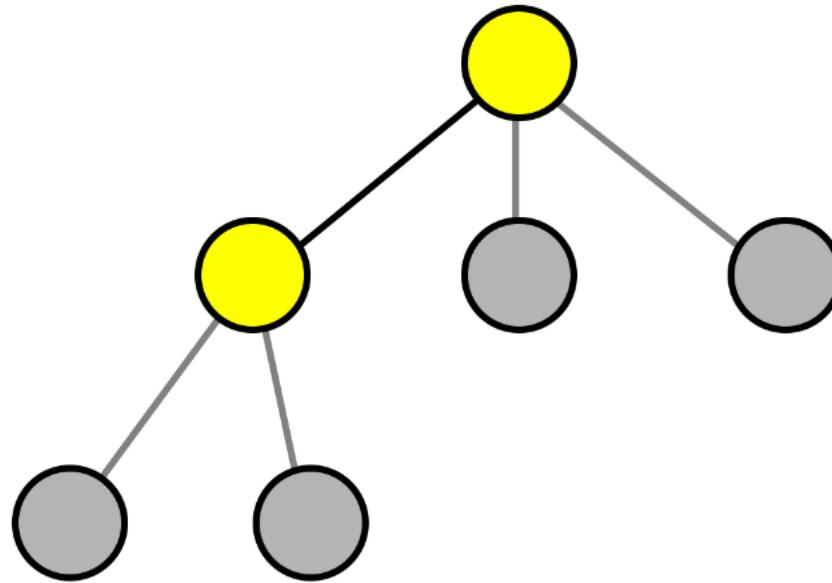
Le backtracking : illustration du principe

```
1 function backtrack(s,d):  
2 | if isSolution(s):  
3 | | return True  
4 | for n in nextSituations(s):  
5 | | if backtrack(n,d+1):  
6 | | | return True  
7 | return False
```



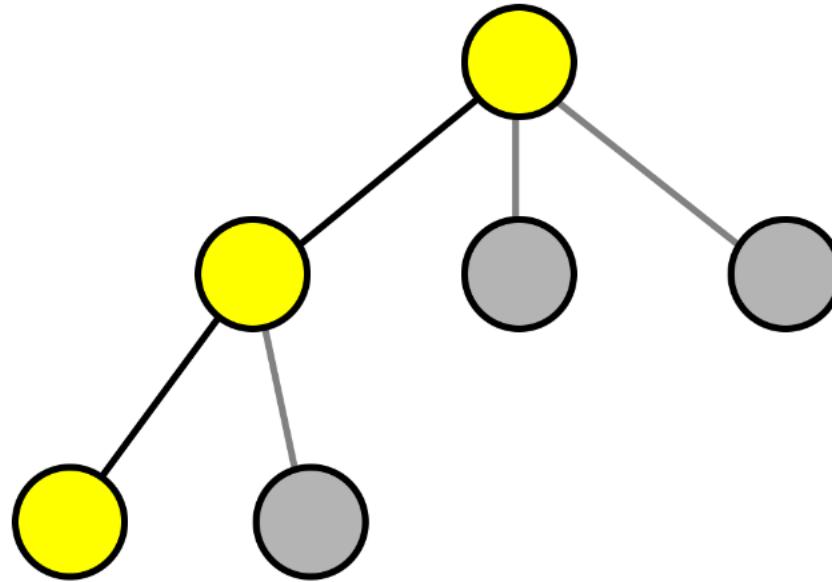
Le backtracking : illustration du principe

```
1 function backtrack(s,d):  
2 | if isSolution(s):  
3 | | return True  
4 | for n in nextSituations(s):  
5 | | if backtrack(n,d+1):  
6 | | | return True  
7 | return False
```



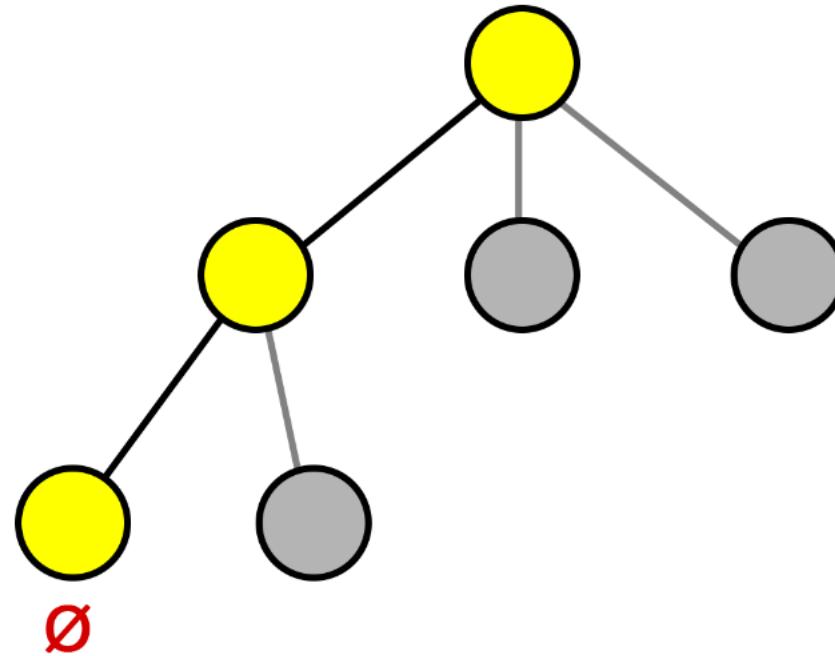
Le backtracking : illustration du principe

```
1 function backtrack(s,d):  
2 | if isSolution(s):  
3 | | return True  
4 | for n in nextSituations(s):  
5 | | if backtrack(n,d+1):  
6 | | | return True  
7 | return False
```



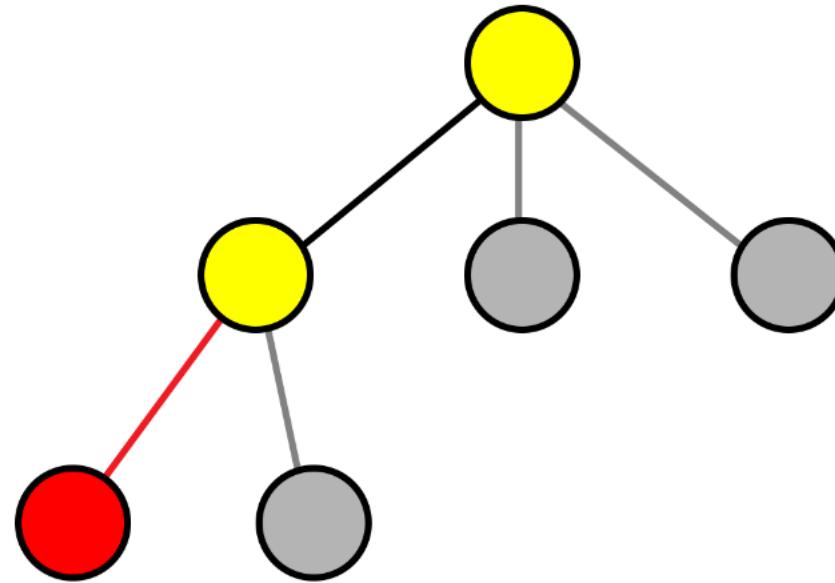
Le backtracking : illustration du principe

```
1 function backtrack(s,d):  
2 | if isSolution(s):  
3 | | return True  
4 | for n in nextSituations(s):  
5 | | if backtrack(n,d+1):  
6 | | | return True  
7 | return False
```



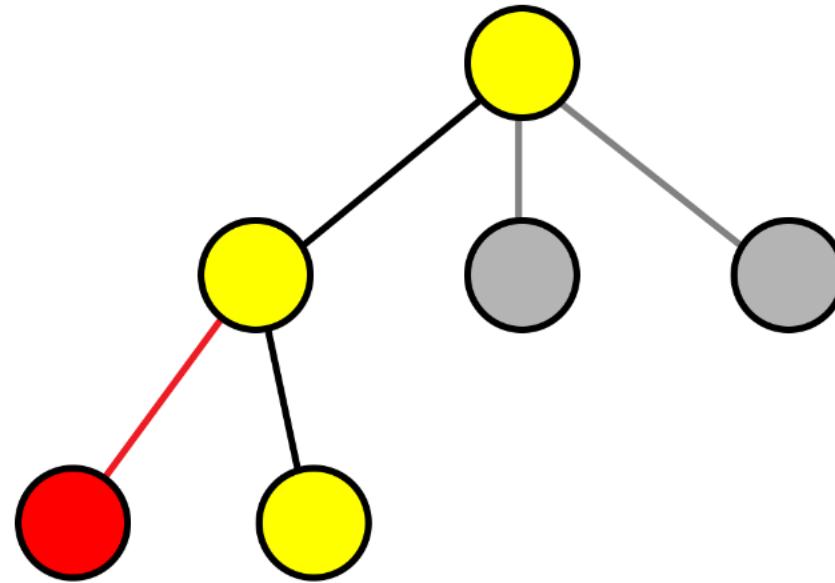
Le backtracking : illustration du principe

```
1 function backtrack(s,d):  
2 | if isSolution(s):  
3 | | return True  
4 | for n in nextSituations(s):  
5 | | if backtrack(n,d+1):  
6 | | | return True  
7 | return False
```



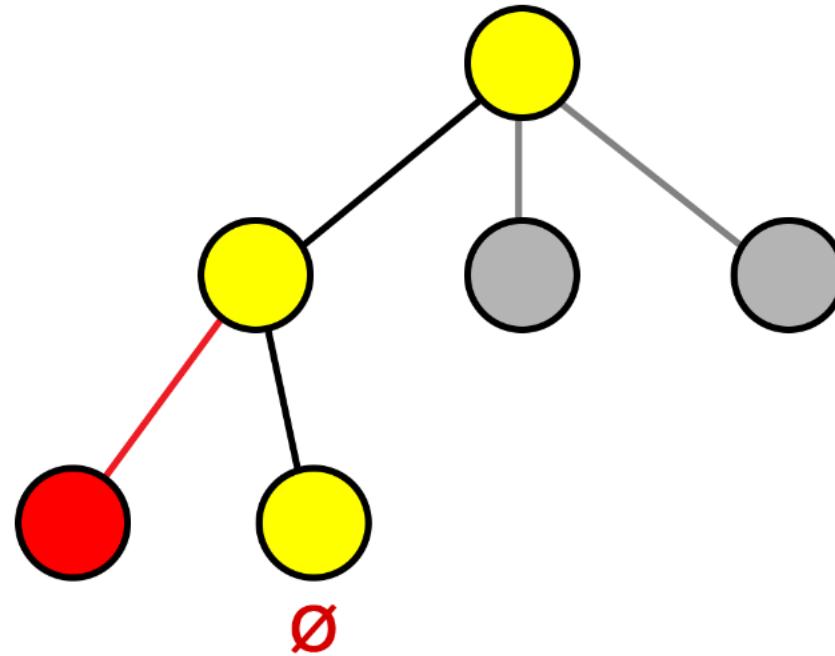
Le backtracking : illustration du principe

```
1 function backtrack(s,d):  
2 | if isSolution(s):  
3 | | return True  
4 | for n in nextSituations(s):  
5 | | if backtrack(n,d+1):  
6 | | | return True  
7 | return False
```



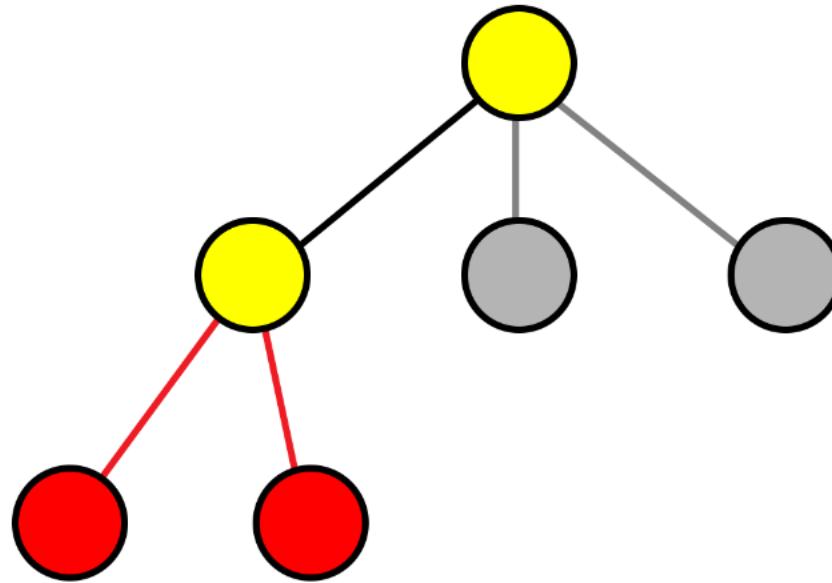
Le backtracking : illustration du principe

```
1 function backtrack(s,d):  
2 | if isSolution(s):  
3 | | return True  
4 | for n in nextSituations(s):  
5 | | if backtrack(n,d+1):  
6 | | | return True  
7 | return False
```



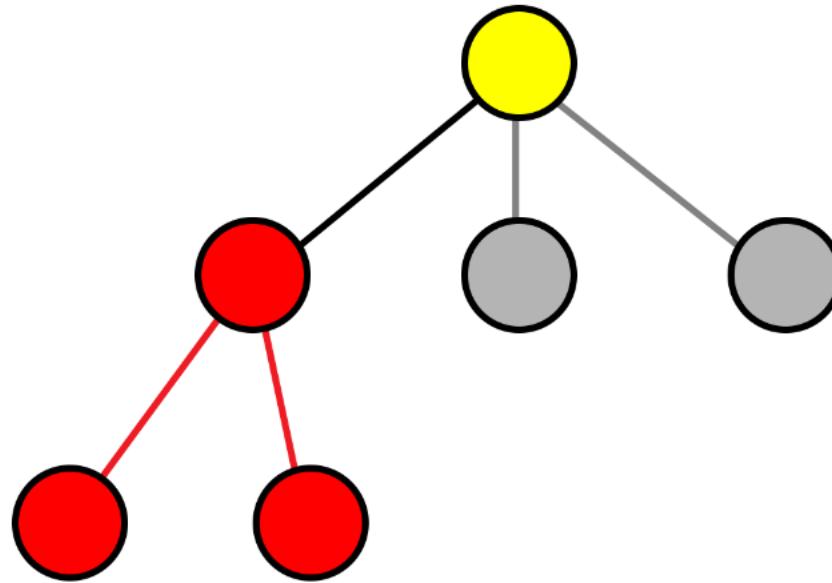
Le backtracking : illustration du principe

```
1 function backtrack(s,d):  
2 | if isSolution(s):  
3 | | return True  
4 | for n in nextSituations(s):  
5 | | if backtrack(n,d+1):  
6 | | | return True  
7 | return False
```



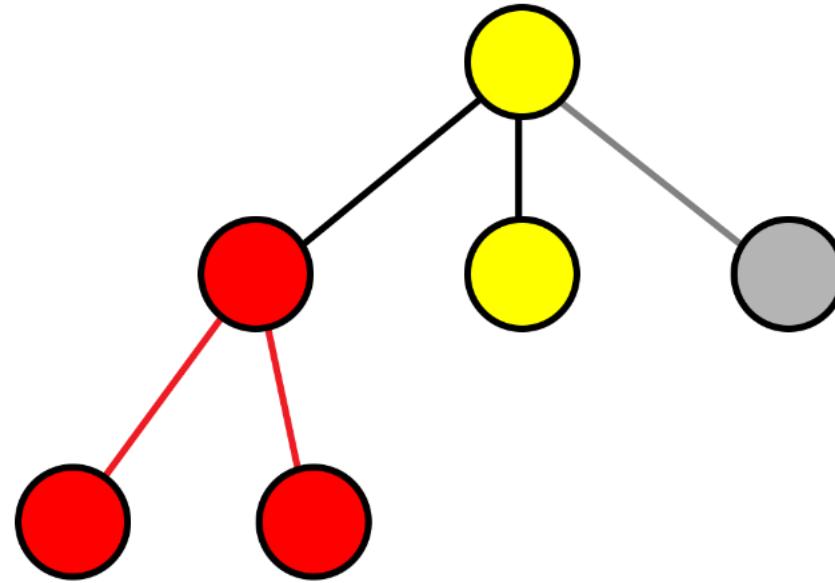
Le backtracking : illustration du principe

```
1 function backtrack(s,d):  
2 | if isSolution(s):  
3 | | return True  
4 | for n in nextSituations(s):  
5 | | if backtrack(n,d+1):  
6 | | | return True  
7 | return False
```



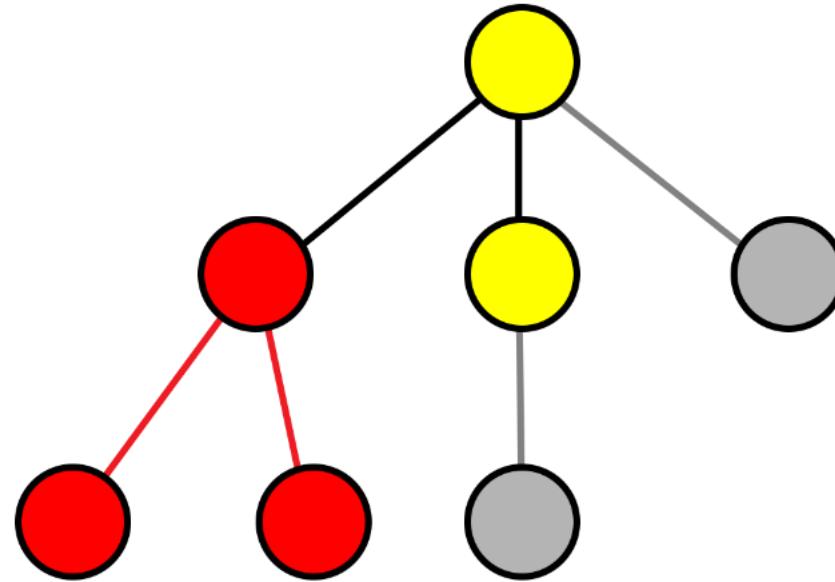
Le backtracking : illustration du principe

```
1 function backtrack(s,d):  
2 | if isSolution(s):  
3 | | return True  
4 | for n in nextSituations(s):  
5 | | if backtrack(n,d+1):  
6 | | | return True  
7 | return False
```



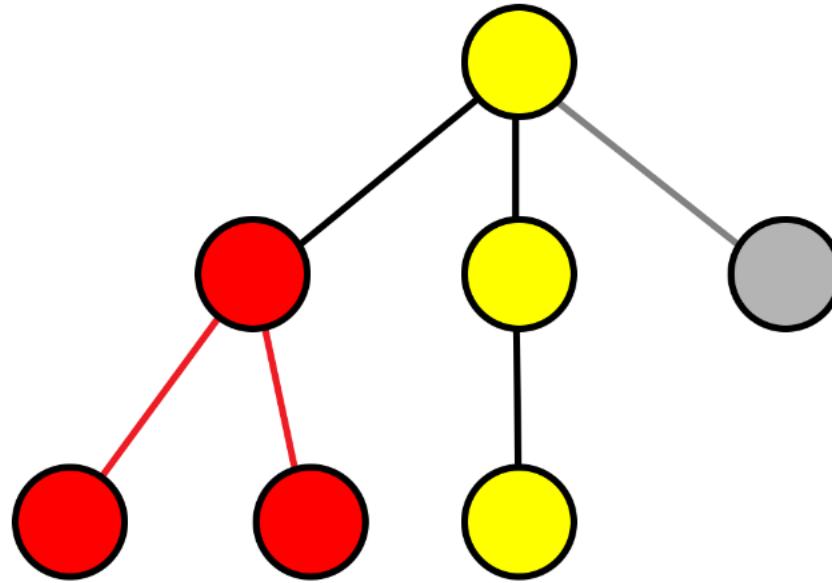
Le backtracking : illustration du principe

```
1 function backtrack(s,d):  
2 | if isSolution(s):  
3 | | return True  
4 | for n in nextSituations(s):  
5 | | if backtrack(n,d+1):  
6 | | | return True  
7 | return False
```



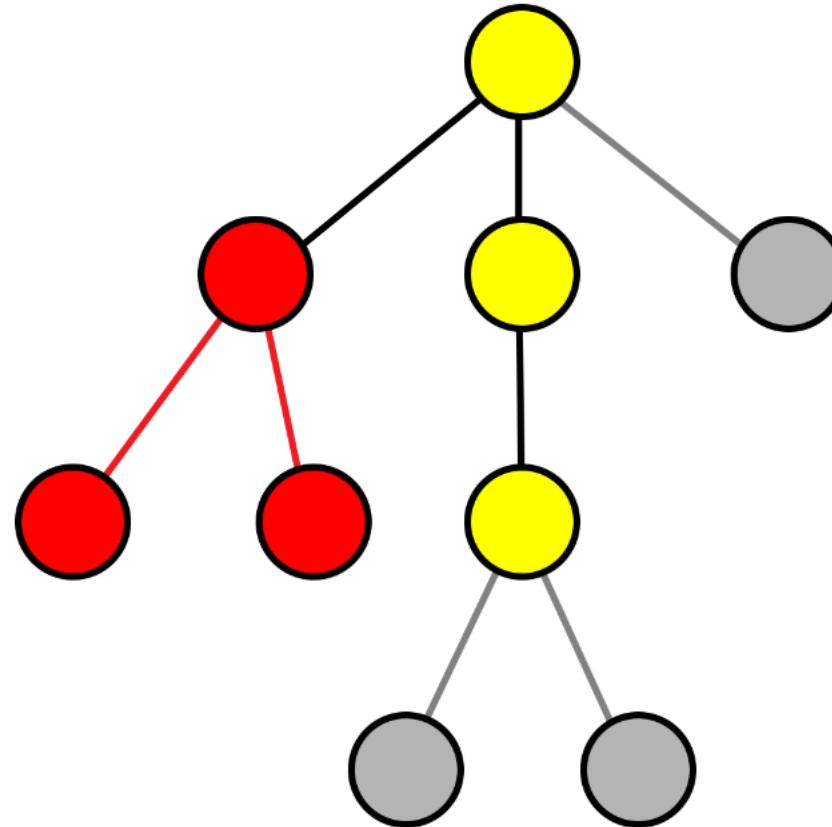
Le backtracking : illustration du principe

```
1 function backtrack(s,d):  
2 | if isSolution(s):  
3 | | return True  
4 | for n in nextSituations(s):  
5 | | if backtrack(n,d+1):  
6 | | | return True  
7 | return False
```



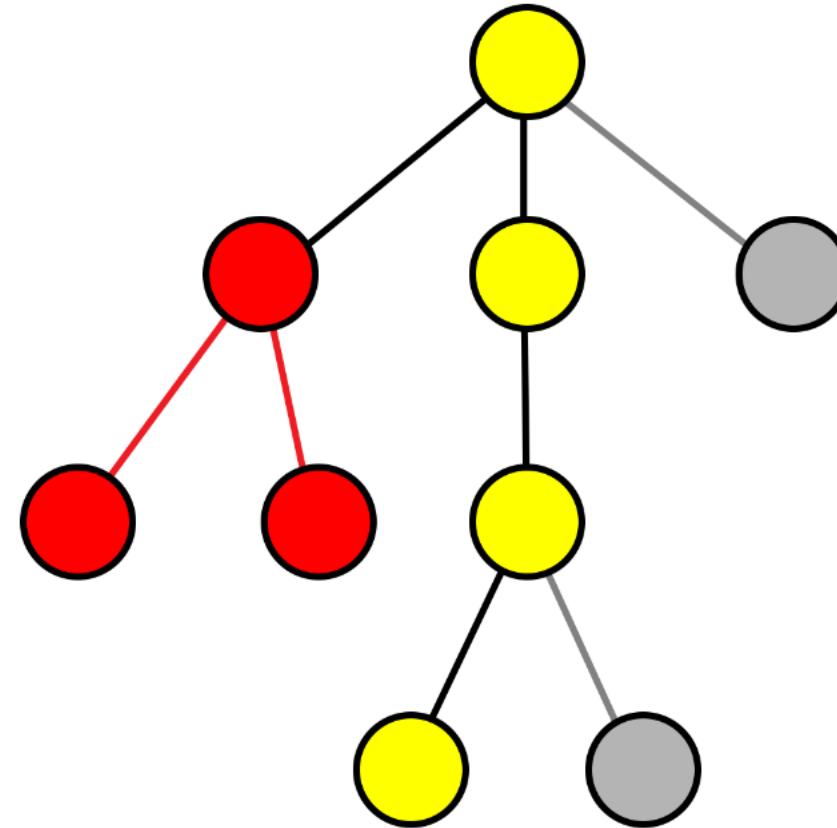
Le backtracking : illustration du principe

```
1 function backtrack(s,d):  
2 | if isSolution(s):  
3 | | return True  
4 | for n in nextSituations(s):  
5 | | if backtrack(n,d+1):  
6 | | | return True  
7 | return False
```



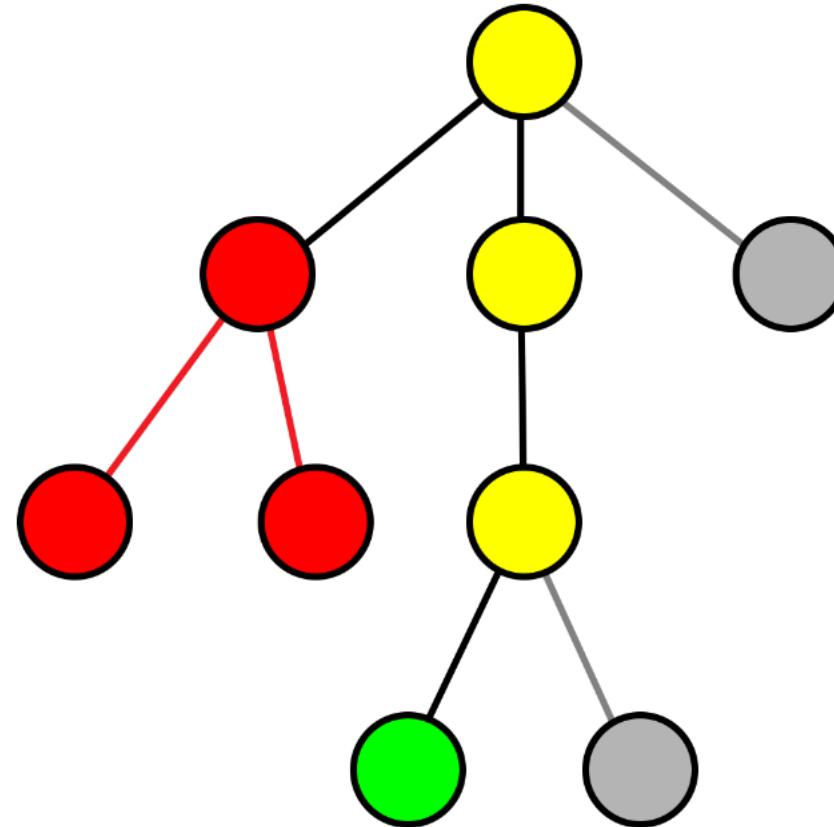
Le backtracking : illustration du principe

```
1 function backtrack(s,d):  
2 | if isSolution(s):  
3 | | return True  
4 | for n in nextSituations(s):  
5 | | if backtrack(n,d+1):  
6 | | | return True  
7 | return False
```



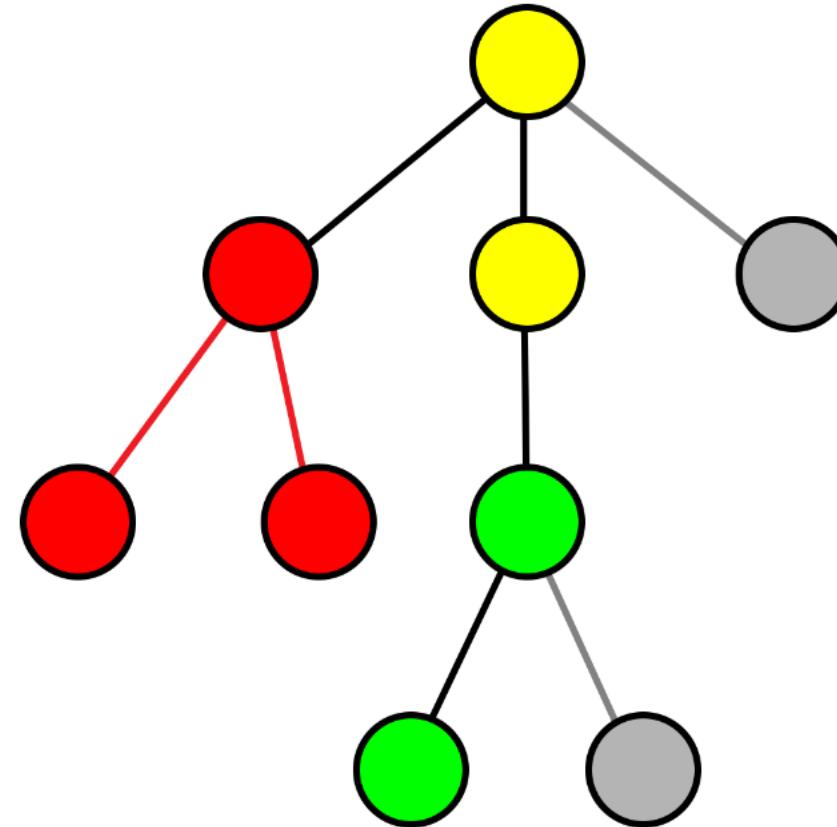
Le backtracking : illustration du principe

```
1 function backtrack(s,d):  
2 | if isSolution(s):  
3 | | return True  
4 | for n in nextSituations(s):  
5 | | if backtrack(n,d+1):  
6 | | | return True  
7 | return False
```



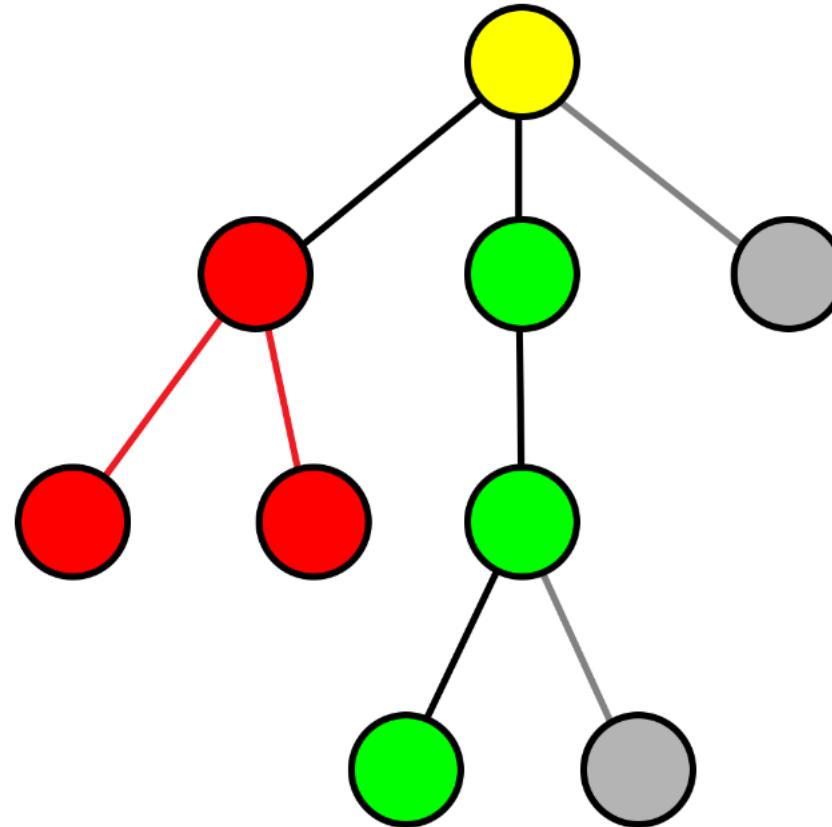
Le backtracking : illustration du principe

```
1 function backtrack(s,d):  
2 | if isSolution(s):  
3 | | return True  
4 | for n in nextSituations(s):  
5 | | if backtrack(n,d+1):  
6 | | | return True  
7 | return False
```



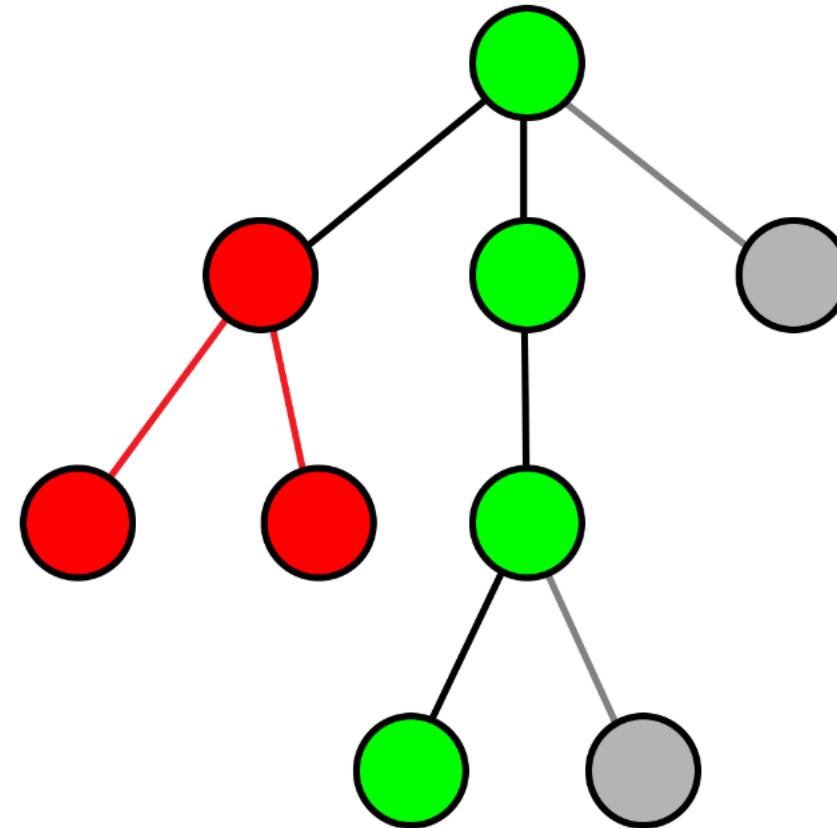
Le backtracking : illustration du principe

```
1 function backtrack(s,d):  
2 | if isSolution(s):  
3 | | return True  
4 | for n in nextSituations(s):  
5 | | if backtrack(n,d+1):  
6 | | | return True  
7 | return False
```



Le backtracking : illustration du principe

```
1 function backtrack(s,d):  
2 | if isSolution(s):  
3 | | return True  
4 | for n in nextSituations(s):  
5 | | if backtrack(n,d+1):  
6 | | | return True  
7 | return False
```



Le backtracking : attention

Lors d'une mise en place d'un algorithme de backtracking, faire attention aux points suivants :

- Soigner la situation d'arrêt (valable pour tout algorithme récursif)
- Si le problème peut présenter un cycle (on peut revenir à un état déjà traité), il faut trouver un moyen de repérer les états déjà parcourus (attribut, stockage dans un dictionnaire,...)
- La complexité du backtracking est exponentielle, en $\Theta(b^d)$ avec `b` le facteur de branchement et `d` la profondeur maximale de récursion.

Le backtracking : coloration de graphe

```
function backtrack(s,d):
| if isSolution(s):
| | return True
| for n in nextSituations(s): →
| | if backtrack(n,d+1):
| | | return True
| return False
```

maxColor = n

```
function backtrackColor(G,k): # k : noeud courant
| if k = |V(G)|:
| | return True
| for c ← 0 to maxColor:
| | V(G)[k].color ← c
| | if not c in [n.color for n in neighbors(V(G)[k])]:
| | | if backtrackColor(G,k+1):
| | | | return True
| | V(G)[k].color ← -1 # décolorer le sommet
| return False
```

L'algorithme renvoie **True** si le graphe est colorable avec `maxColor` couleurs.

Le backtracking : coloration de graphe optimale

A partir de cet algorithme, il est possible d'obtenir $\chi(G)$

```
maxColor = n
```

```
function backtrackColor(G,k):
| if k = |V(G)|:
| | return True
| for c < 0 to maxColor:
| | V(G)[k].color <- c
| | if not c in [n.color for n in neighbors(V(G)[k])]:
| | | if backtrackColor(G,k+1):
| | | | return True
| | V(G)[k].color <- -1
| return False
```

```
maxColor <- ∞
```



```
function backtrackColorOpt(G,k):
| if k = |V(G)|:
| | maxColor <- min(maxColor, countColors(G))
| for c < 0 to maxColor:
| | V(G)[k].color <- c
| | if not c in [n.color for n in neighbors(V(G)[k])]:
| | | backtrackColorOpt(G,k+1):
| | | | V(G)[k].color <- -1
```

Branch and bound

(séparation et évaluation)

Branch and bound : principe

Branch : séparer l'ensemble des solutions en ensembles plus petits

Bound : évaluer ces sous-ensembles et n'explorer que ceux pouvant contenir une solution meilleure que la meilleure solution courante (par majoration).

« solution courante » : on stocke la meilleure solution trouvée jusqu'à présent par l'algorithme.

Cela nous donne encore une fois une structure d'arbre, dans lequel seules les branches les plus prometteuses vont être parcourues (on parle d'élagage, d'arbre de décision,...)

Branch and bound : comment s'y prendre ?

Comment savoir si un ensemble de solutions ne contient pas de solution optimale ?

→ Déterminer une borne inférieure pour le coût des solutions de cet ensemble (s'il s'agit d'un problème de minimisation).

Si cette borne inférieure est de coût supérieur au coût de la meilleure solution courante, le sous-ensemble ne contient pas d'optimum.

Branch and bound : complexité

En théorie la même que celle du backtracking : exponentielle, en $\Theta(b^d)$

En pratique, on peut arriver bien plus vite à une solution optimale (ou acceptable, on peut s'arrêter quand on veut)

Utilisations :

- Problèmes d'optimisation
- IA (échecs,...)

Branch and bound : exemple

	item 1	item 2	item 3	item 4
coût	4	5	6	2
poids	33	49	60	32
Nombre	x_1	x_2	x_3	x_4

Poids maximal : 130

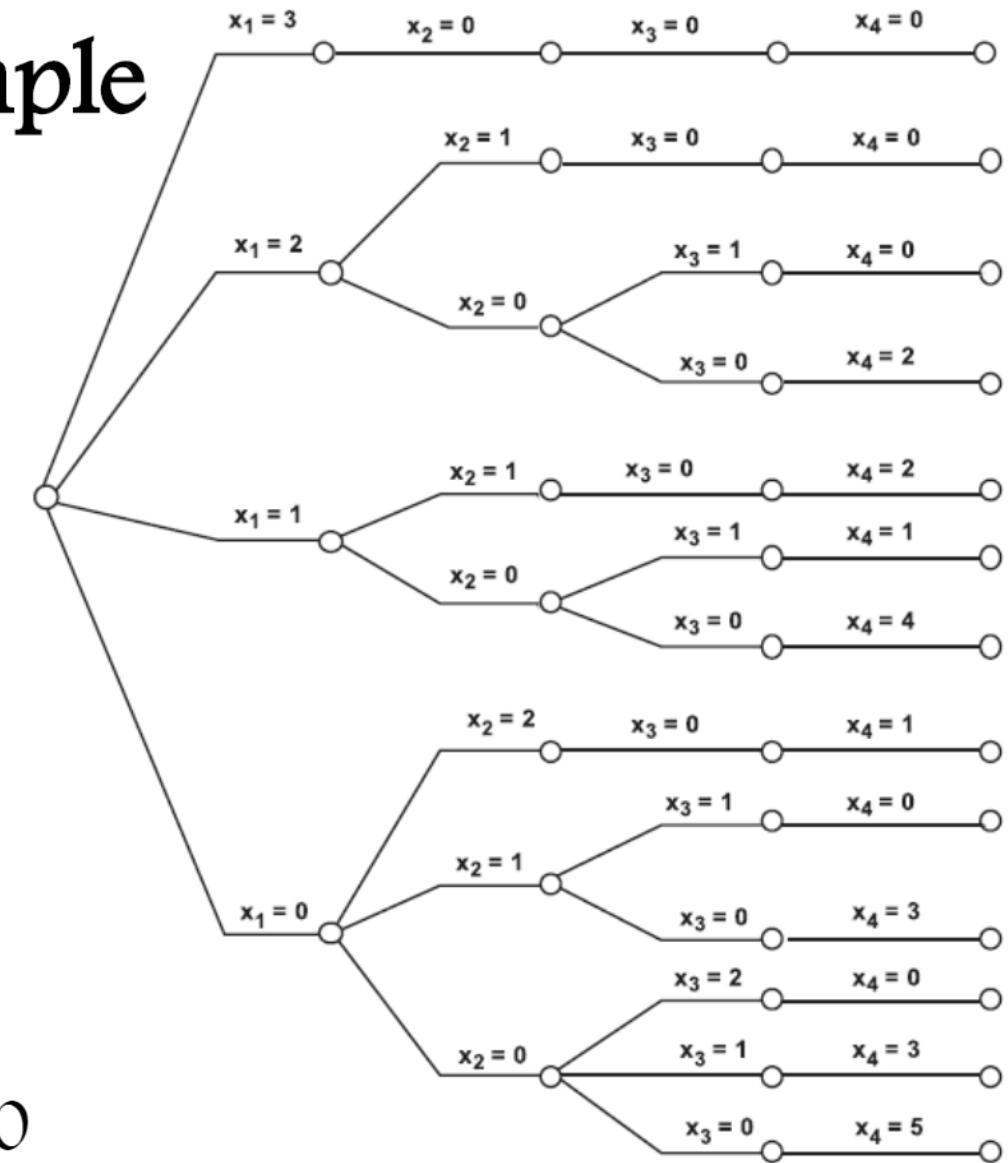
Nombre d'objet maximal : 4

On veut maximiser :

$$4x_1 + 5x_1 + 6x_1 + 2x_1$$

Sous la contrainte :

$$33x_1 + 49x_1 + 60x_1 + 32x_1 \leq 130$$

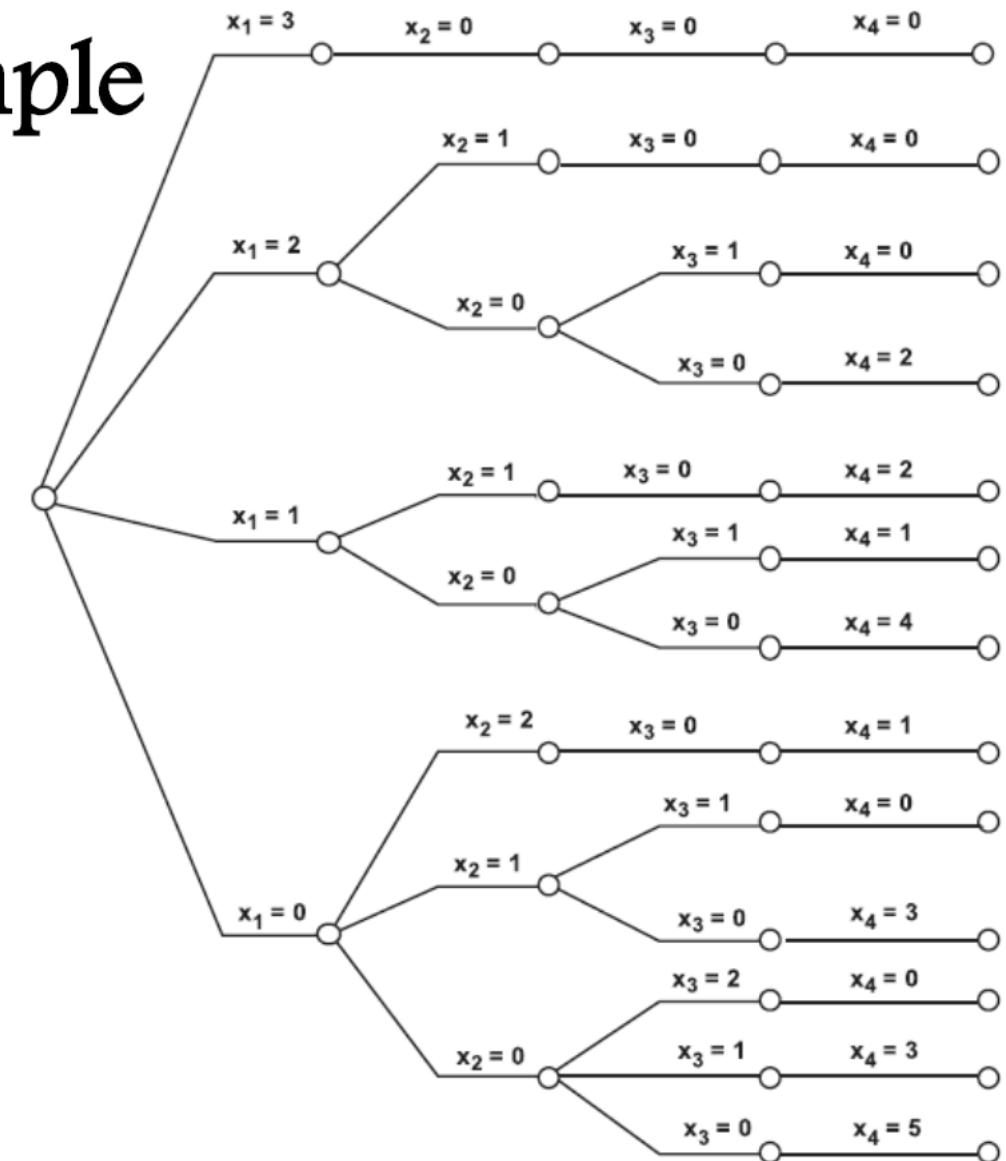


Branch and bound : exemple

	item 1	item 2	item 3	item 4
coût	4	5	6	2
poids	33	49	60	32
Nombre	x_1	x_2	x_3	x_4

Fonction d'évaluation pour ce problème
(critère de choix pour le prochain noeud) :
maximiser coût/poids

Ici item 1 a un coût/poids maximal, on va
commencer par ajouter des items 1

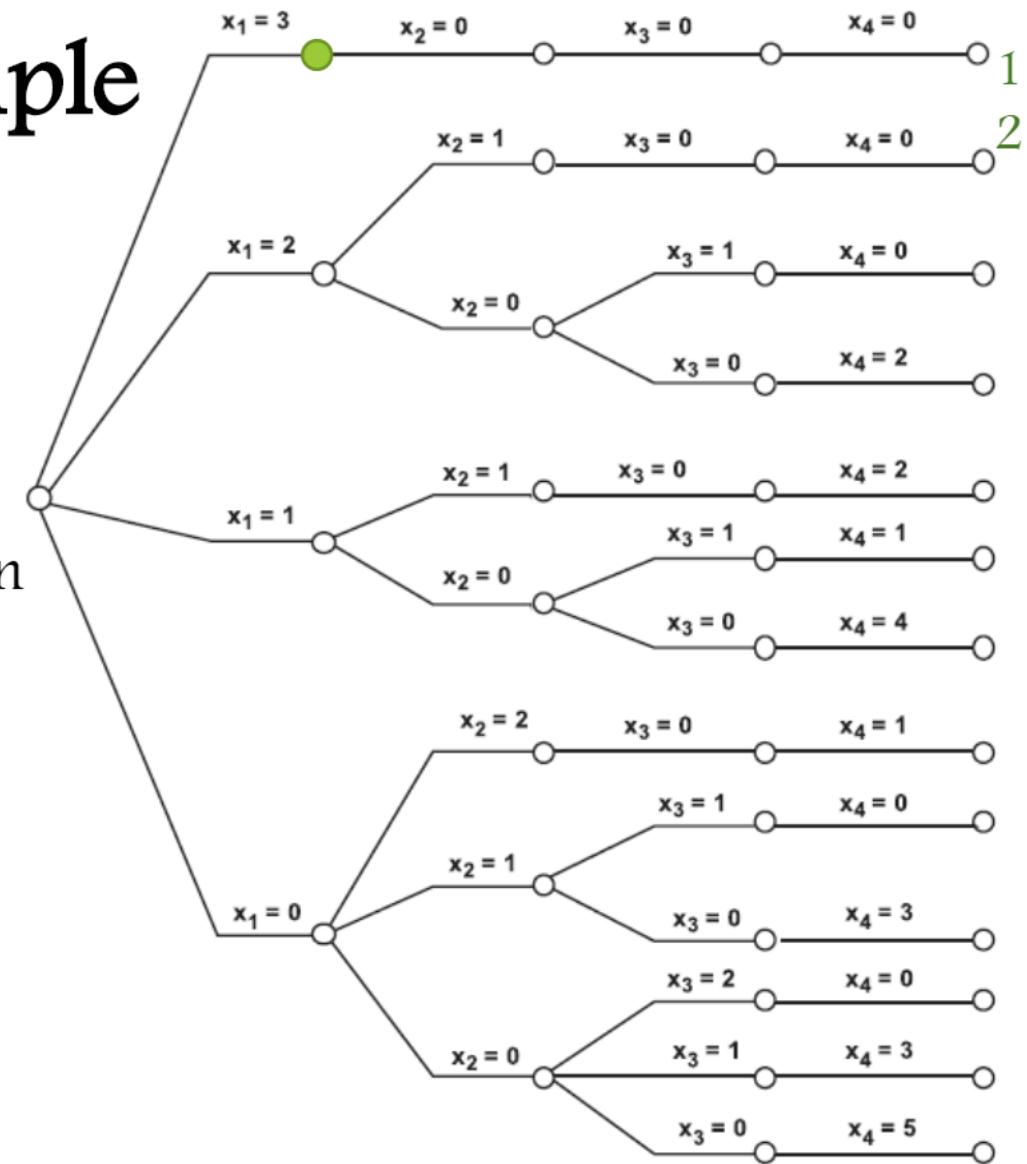


Branch and bound : exemple

	item 1	item 2	item 3	item 4
coût	4	5	6	2
poids	33	49	60	32

> $x_1 = 3$: on ne peut plus rien mettre dans le sac, on a donc une (meilleure) solution à 12

Meilleure solution : 12



Branch and bound : exemple

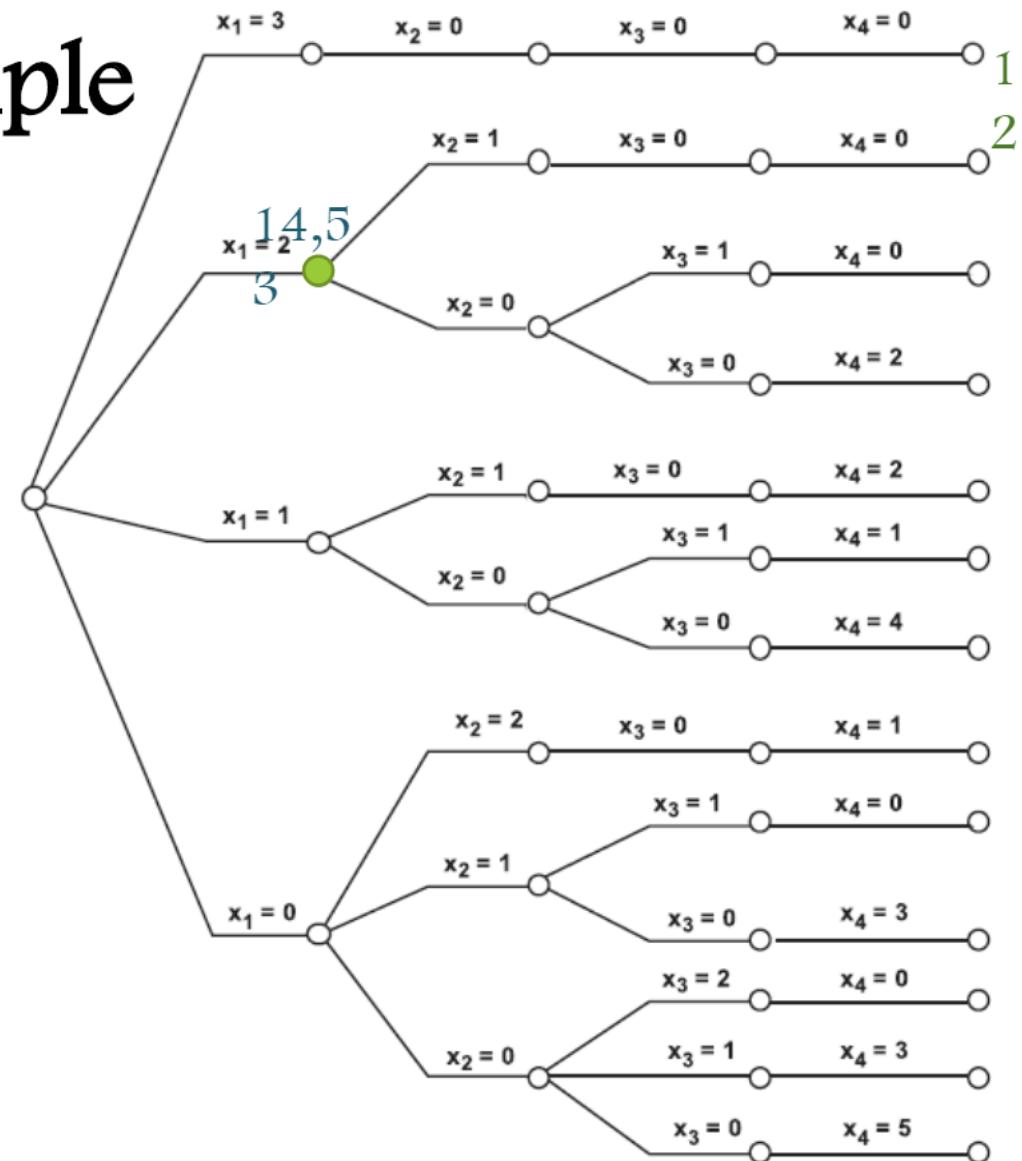
	item 1	item 2	item 3	item 4
coût	4	5	6	2
poids	33	49	60	32

> $x_1 = 2$: item 2 a maintenant le meilleur coût/poids.

$$\text{Evaluation : } 2 \cdot 4 + \frac{5}{49} \times (130 - 2 \cdot 33) = 14,53$$

$14,53 > 13 = 12+1$ donc on a potentiellement une meilleure solution, on sépare le nœud

Meilleure solution : 12

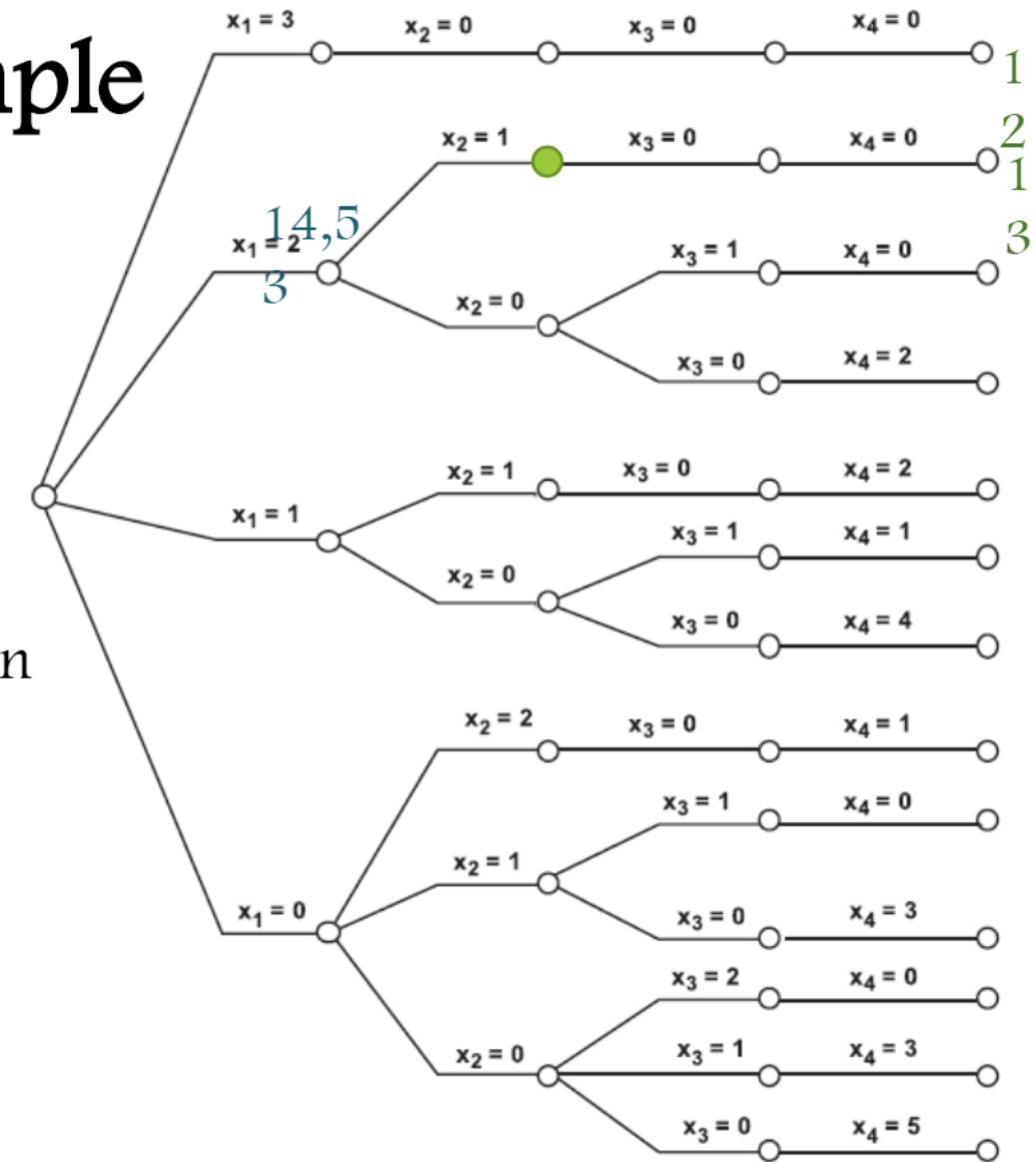


Branch and bound : exemple

	item 1	item 2	item 3	item 4
coût	4	5	6	2
poids	33	49	60	32

>> $x_1 = 2, x_2 = 1$: on ne peut plus rien ajouter,
solution à $2x_4 + 5 = 13$, c'est une meilleure solution

Meilleure solution : 13



Branch and bound : exemple

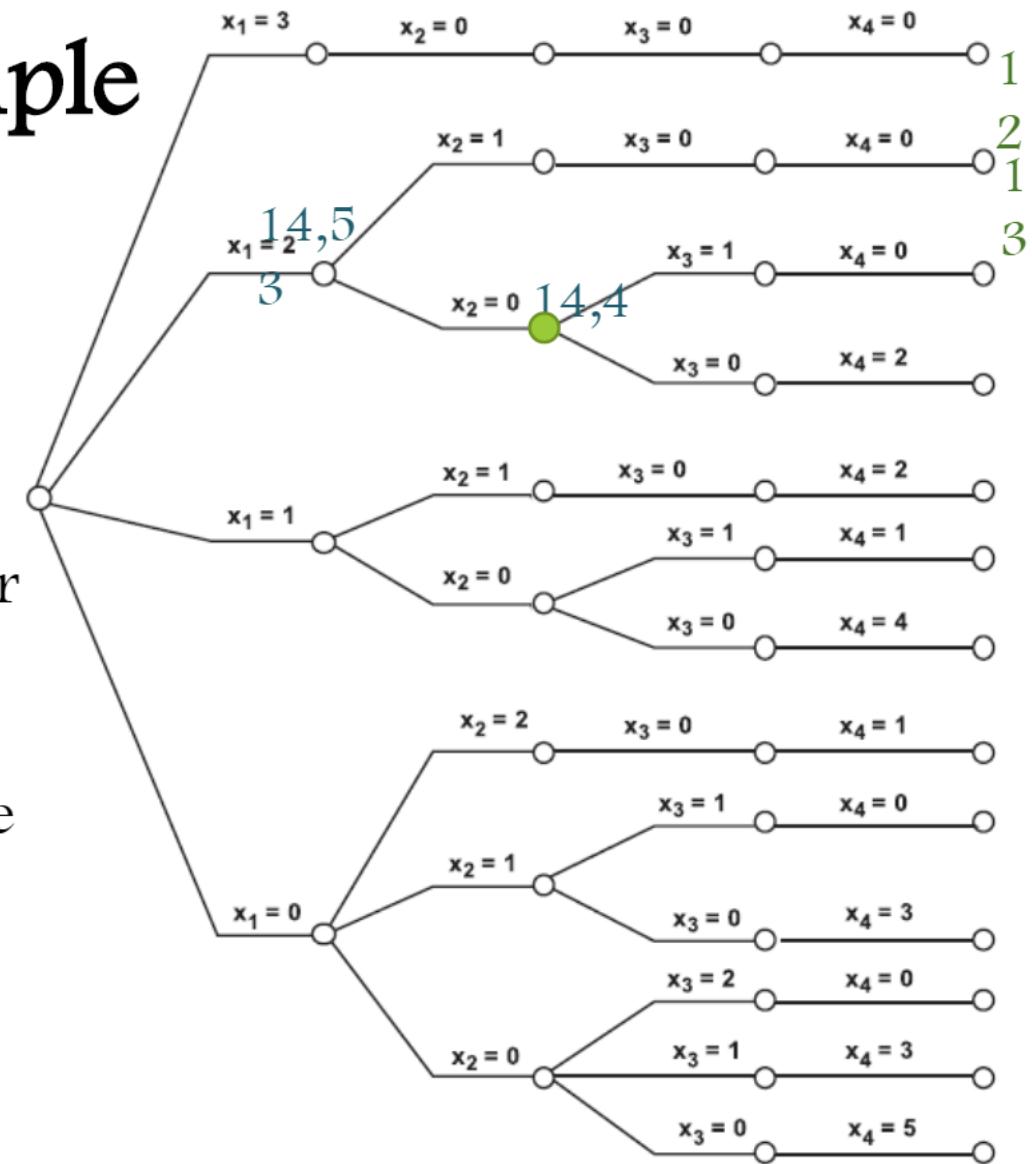
	item 1	item 2	item 3	item 4
coût	4	5	6	2
poids	33	49	60	32

>> $x_1 = 2, x_2 = 0$: l'item 3 a maintenant le meilleur coût/poids.

Evaluation : $2^*4 + 0 + 6/60 \times (130 - 2 \times 33) = 14,4$

$14 > 13+1$ donc on a potentiellement une meilleure solution, on sépare le nœud

Meilleure solution : 13

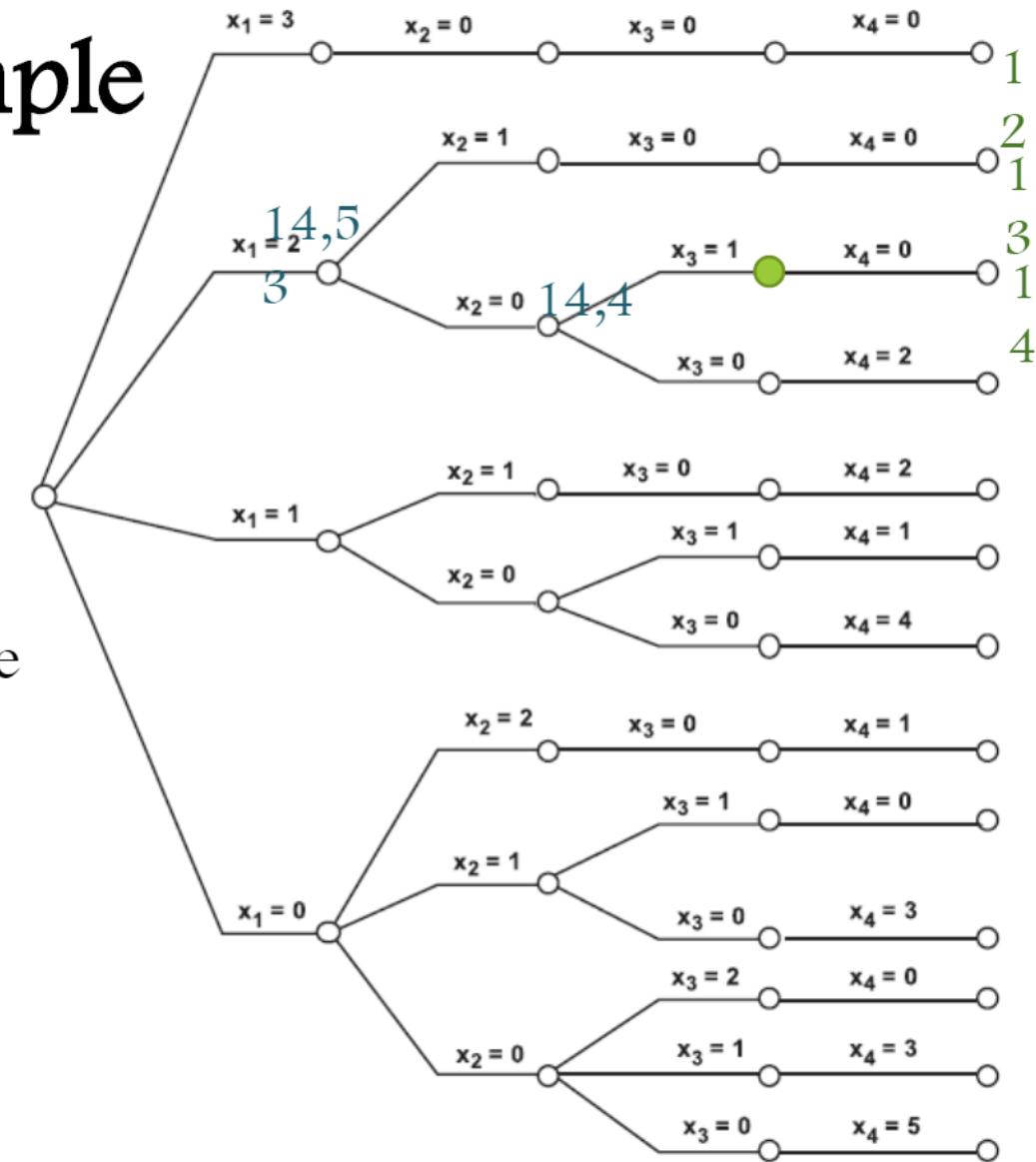


Branch and bound : exemple

	item 1	item 2	item 3	item 4
coût	4	5	6	2
poids	33	49	60	32

>>> $x_1 = 2, x_2 = 0, x_3 = 1$: on ne peut plus rien ajouter, solution à $2x_4 + 6 = 14$, c'est une meilleure solution

Meilleure solution : 14



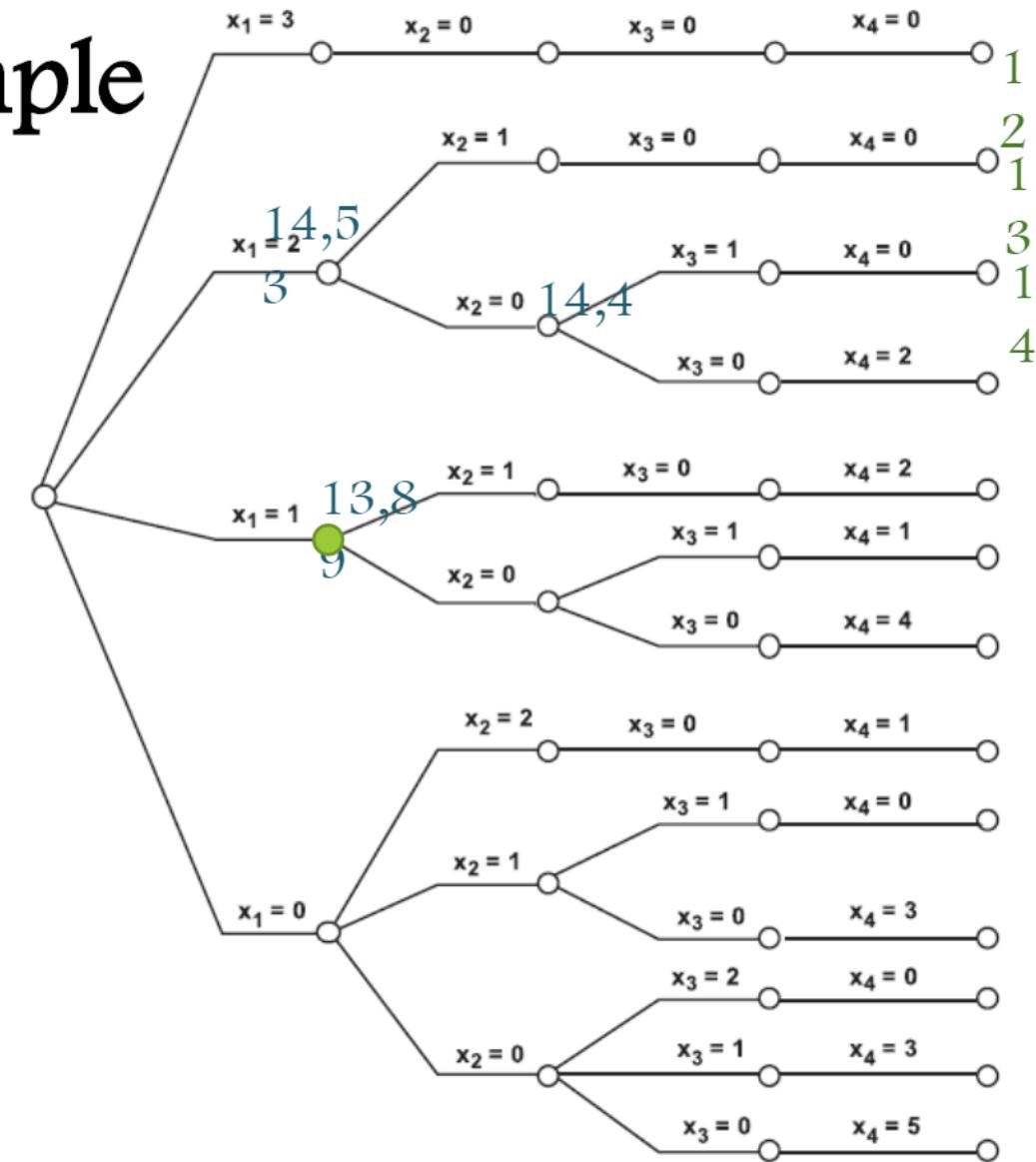
Branch and bound : exemple

	item 1	item 2	item 3	item 4
coût	4	5	6	2
poids	33	49	60	32

$> x_1 = 1$: évaluation $4 + 5/49 \times (130 - 33) = 13,89$

On n'aura pas de meilleure solution dans cette branche (branche élaguée)

Meilleure solution : 14



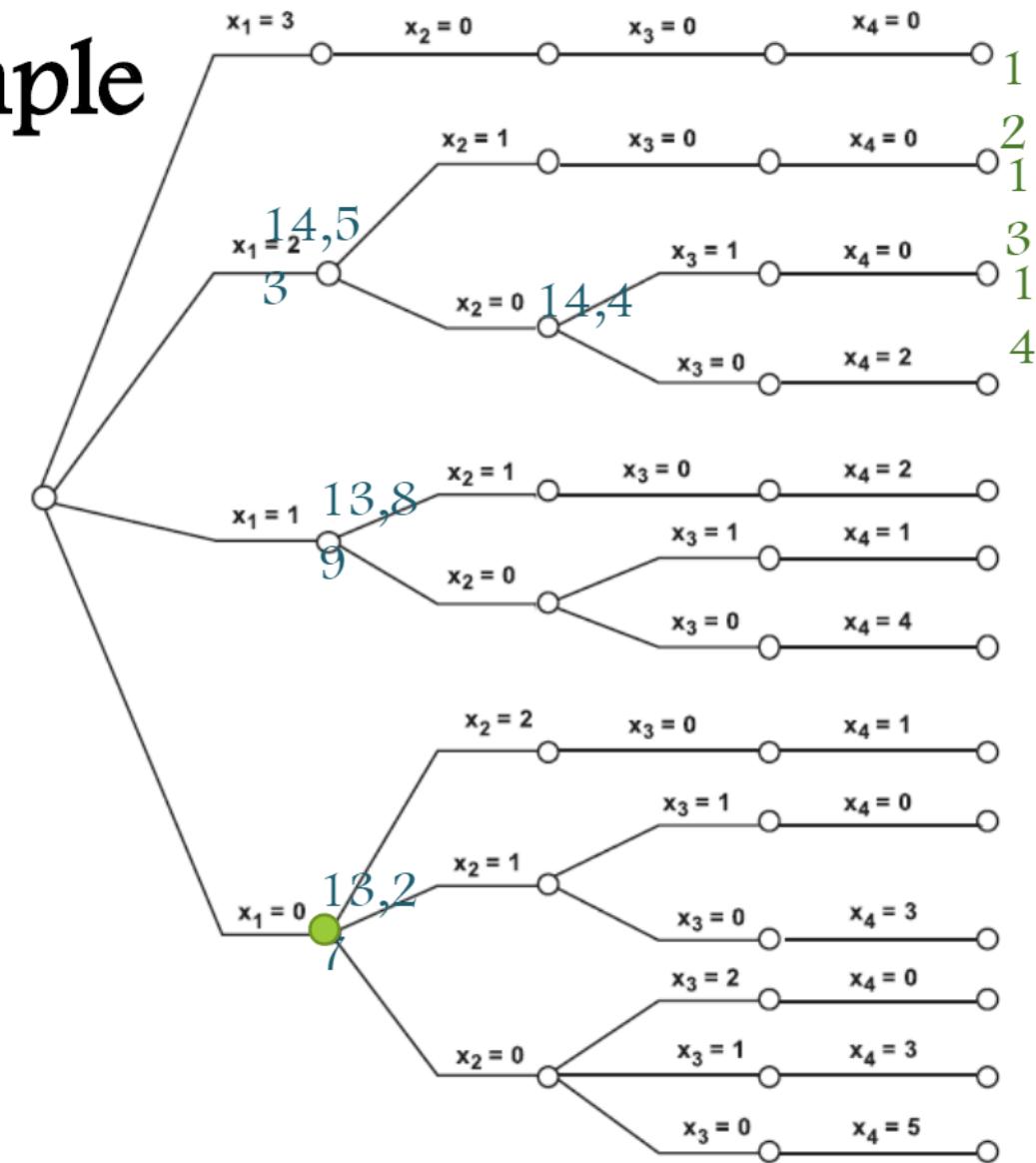
Branch and bound : exemple

	item 1	item 2	item 3	item 4
coût	4	5	6	2
poids	33	49	60	32

$> x_1 = 0$: évaluation $5/49 \times 130 = 13,27$

On n'aura pas de meilleure solution dans cette branche (branche élaguée)

Meilleure solution : 14



Branch and bound : algorithme

```
function solveBB(problem):
    activeNodes ← getActiveNodes(problem)
    best ← -∞
    while activeNodes != []:
        n = choseActiveNode(activeNodes) # à définir
        if estSolution(n):
            best ← max(best,n)
        else:
            children ← split(n)
            for f in children:
                e ← eval(f) # à définir
                if e > best:
                    activeNodes.add(f)
    return best
```