Stratégies et Techniques de Résolution de Problèmes

Chapitre II-1 : Graphes et Algorithmes

S7 - ECL – 2A – MI

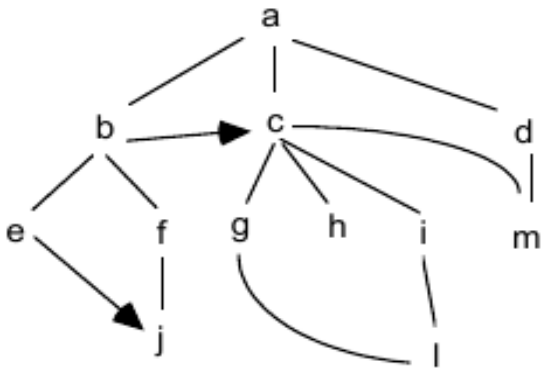
2017-2018

Alexandre Saidi

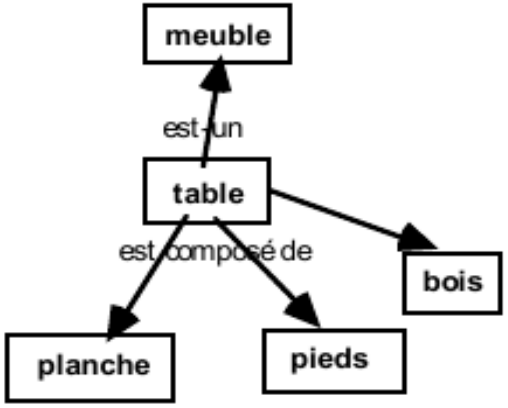
I- Introduction aux Graphes

- Moyen de structuration et de représentation (hiérarchie, composition, structure, modèle, etc.)
- Outil de modélisation de problèmes
- Une généralisation des arbres

Exemples



un graphe représentant des liens entre différents noeuds (villes)



un graphe representant des liens d'héritage et de composition

I.1- Exemples d'applications (modélisables par les graphes)

- **Efficacité des pipelines, problème de flux :**
  - Transport et distribution d'eau/pétrole/gaz/etc.
    - ➟ On peut s'intéresser au FLUX ou à MST pour simplement assurer la distribution
    - ➟ les valeurs des arcs (du graphe) : coût ou capacité.

- **Table de routage, Gestion du trafic, chemins les plus courts :**
  - Calcul des plus courts chemins entre les routeurs eux-mêmes pour savoir comment atteindre une destination par la meilleure voie.
  - Rechercher des chemins d'encombrement minimum

- **Transport et visite (messagerie) :**
  - Visiter des points de livraison pour prendre/déposer des colis.
    - ➟ le voyageur de commerce
    - ➟ le trajet d'un facteur

- **Réseaux de communication :**
  - Les réseaux avec leurs équipements (lignes tél, relais, Satellites, ...)

    à installer de façon optimale (MST)

- **Navigation aérienne** (les avions dans des couloirs au ciel !)
  - Problème : le vent, coût du survol d'un espace, le trafic...
    - ➟ Le MST (chemin de poids minimum entre 2 points)

- **Câblage de circuits imprimés**

  - Connexion des broches des puces (les broches standard, cartes différentes, ...)

    ➟ MST permet de les connecter

      – Aussi, le problème de placement des puces et les pistes les plus courtes (cf. chemins sous contraintes)

      – Problème de largeur de pistes, taille des cartes, ...

Le système de transport fermé (circuit fermé)

  - Délivrer des pièces, amener des marchandises (ou en emporter d'un magasin)

  ➟ TSP (voir aussi CLP)

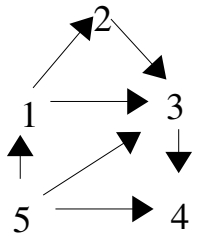
- Bin packing : …
- Etc ...

II- Graphes : quelques notions

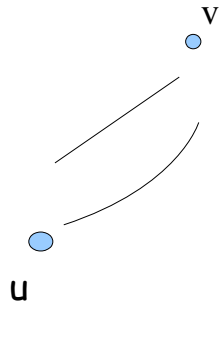
- Un graphe **G** = **(V, E)**

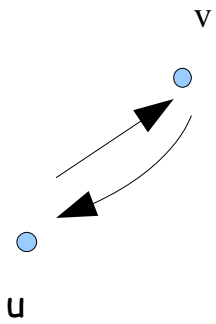
  V : ensemble de **nœuds** (*vertex*)

  E ⊆ (V x V) : ensemble d'**arêtes** ou **arcs** (*edges*)



Un Digraphe





- Chaque **arc** = une paire (v,w) ∈ **E**, v,w ∈ V

  ➟ **Arête** : double arc

  ➟ (v,w) sont *adjacents* (voisins) : w est **adjacent** de v si **(v,w)** ∈ **E**

  ➟ **v** est le *successeur* de **w** (resp. *Prédécesseur*) si v et w sont liés par un arc

- Dans certains problèmes, les nœuds représentent les variables et les arcs les relations.
- Si le couple (v, w) est **ordonné**, on aura un **graphe orienté** (**digraphe**) ➟ *directed graph*
- **Poids** (weight) : valeur d'un arc/arête

  ➟ Graphe *valué (pondéré) : les arcs / arêtes portent un poids : temps, distance, prix,...*

- **Chemin** (branche, path) : $w_1, w_2, \ldots w_n$ tel que $(w_i, w_{i+1}) \in E$

$$chemin((X_1,\ldots, X_k), (V,E)) \equiv (X_1,X_2) \in E \wedge \ldots \wedge (X_{k-1},X_k) \in E$$

- Un nœud Y est *accessible* depuis un nœud X s'il existe un chemin de X vers Y:

$$(X,Y) \in E \vee (\exists\ Z_1,\ldots Z_k : chemin((X, Z1,\ldots Zk, Y), (V,E)))$$

- ***Longueur*** *d'un chemin contenant N nœuds = nombre d'arcs = N - 1*

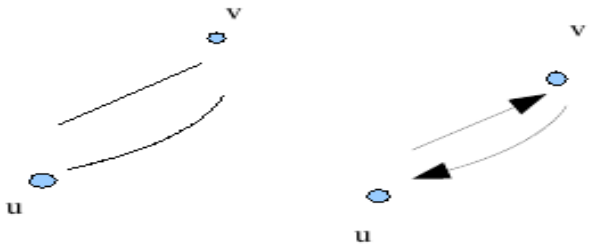
➟ (v,w) est de longueur nulle si (v, **w**) $\notin$ **E** (il n'y a pas d'arc de *v* à *w*).

- Un chemin non nul entre (v,w) dans un graphe orienté est un **circuit** (un **cycle**) si v = w.

➟ Dans un circuit, les arcs sont distincts.

- celui de gauche N'EST pas un cycle (graphe NON orienté)

- celui de droite est un cycle (dans un graphe orienté).

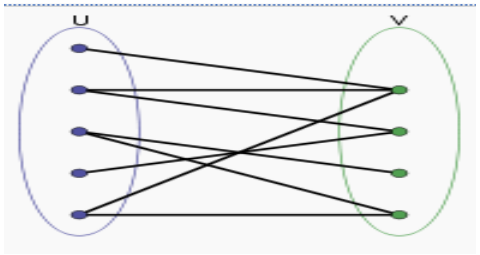


- Si le graphe contient un arc **(v,v)**, le chemin (v,v) est une **boucle** (***loop***)

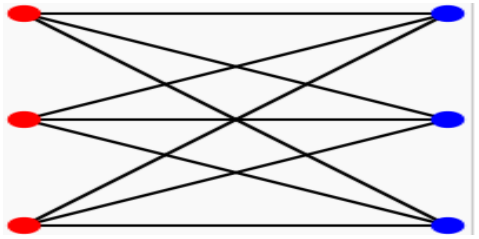
➟ une boucle ne concerne qu'un **nœud**

- Graphe **biparti** : pour le graphe G, il existe 2 sous-ensembles U, **V** tels que chaque arête a une extrémité dans U et l'autre dans V.

  biparti complet (ou **biclique**) si chaque sommet de U est relié à un sommet de V.



graphe biparti



biparti complet (tout est lié à tout)

- Un graphe orienté est **acyclique** s'il n'a pas de cycle ;

  ➟ Un **DAG** : *directed acyclic graph*.

- Un graphe non orienté est **connexe** s'il y a un chemin entre toute paire de nœuds :

  pour tout X,Y ∈ V, accessible(X,Y,(V,E))

  ➟ Un graphe orienté et connexe est appelé **fortement connexe** (*Complet* ou *Dense*)

  si **$|E| \cong O(|V|^2)$**

  → cf. un réseau d'aéroports / ferroviaire où toute paire de villes est desservie par un avion direct.

  ➟ Un graphe peut être peu connexe (peu dense)

- **Réseau**, **Arbre** : graphe connexe sans boucle (acyclique)

Ordre dans un graphe :

- La relation d'ordre binaire '$\preceq$' est réflexive, antisymétrique et transitive sur un ensemble de nœuds d'un graphe.
- La relation '$\preceq$' est dite une **relation d'ordre partiel** si elle est :

  - réflexive(S, $\preceq$) : $\forall X \in S, X \preceq X$

  - antisymétrique (S, $\preceq$) : $\forall X,Y \in S, (X \preceq Y \wedge Y \preceq X \Rightarrow X=Y)$

- transitive(S, $\preceq$) : $\forall X,Y,Z \in S, (X \preceq Y \wedge Y \preceq Z \Rightarrow X \preceq Z)$

- L'ensemble S est *totalement* ordonné sous la relation $\Re$ si tous ses couples d'éléments sont ordonnés : $(\forall X,Y \in S, X \Re Y \vee Y \Re X)$.

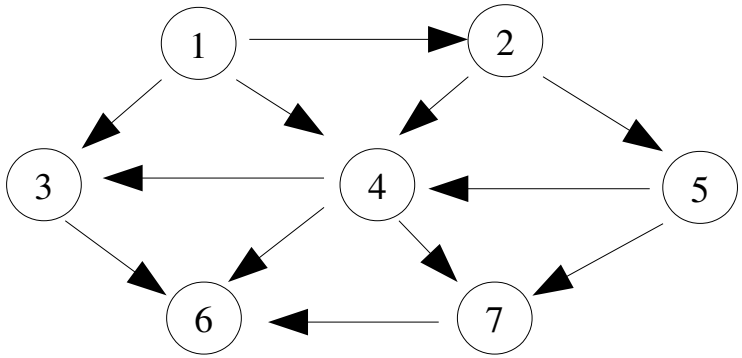
  La relation $\Re$ est alors appelé une **relation d'ordre total**.

  Par Exemple, '$\preceq$' est un ordre total sur les entiers naturels.

- La relation '$\preceq$' sur l'ensemble des nœuds S de graphe est un ordre total.
- **Treillis** : est un arbre (G connexe, acyclique) avec une relation d'ordre totale sur les nœuds.

II.1- Exemple (de graphe)

- Notion du chemin le plus court dans cet exemple :



Graphe G1 : graphe orienté du trafic

Dans ce graphe représentant un réseau de rues, si chaque intersection concerne par exemple 4 rues avec des rues à double sens, on a **|E| ≅ 4|V|**.

II.2- Structure de données pour représenter un graphe

- On peut implanter les graphes de diverses manières :
  - Par une matrice carrée
  - Par un tableau principal + tableau des adjacents
  - Par un tableau principal + listes des adjacents, ...

➠ Selon la nature du graphe (orienté ou non, valué ou non), les structures peuvent être plus ou moins complexes.

II.2.1- Représentation statique par une matrice d'adjacence

- Matrice d'adjacence avec dans chaque case une valeur booléenne (Vrai / Faux) ou une **pondération (valeur)** :
  - Espace $O(|V|^2)$
  - Convient à un graphe **dense** (i.e. $|E| \cong O(|V|^2)$)

- Si beaucoup de "faux" (ou zéro) ➟ un graphe **peu dense** (*sparse*)
  ➟ Ex. : pour 3000 carrefours ➟ matrice de 9000000 éléments avec beaucoup de zéro / faux

La déclaration précédente regroupe les deux cas :

- Graphe non valué : une matrice de booléens.
- Graphe valué : chaque case contient une valeur qui représente la valeur de l'arc (arête) reliant les deux nœuds.

  ➥ Une constante prédéfinie (par exemple -1) représente l'absence de lien.

  ➥ L'orientation des arêtes est exprimée par le contenu des intersections des lignes et des colonnes.

	A	B	C	D	E	F	G
A							
B			V				
C		F					
D		15			3		
E				-1			
F							
G							

II.2.2- Représentation dynamique par les listes d'adjacences

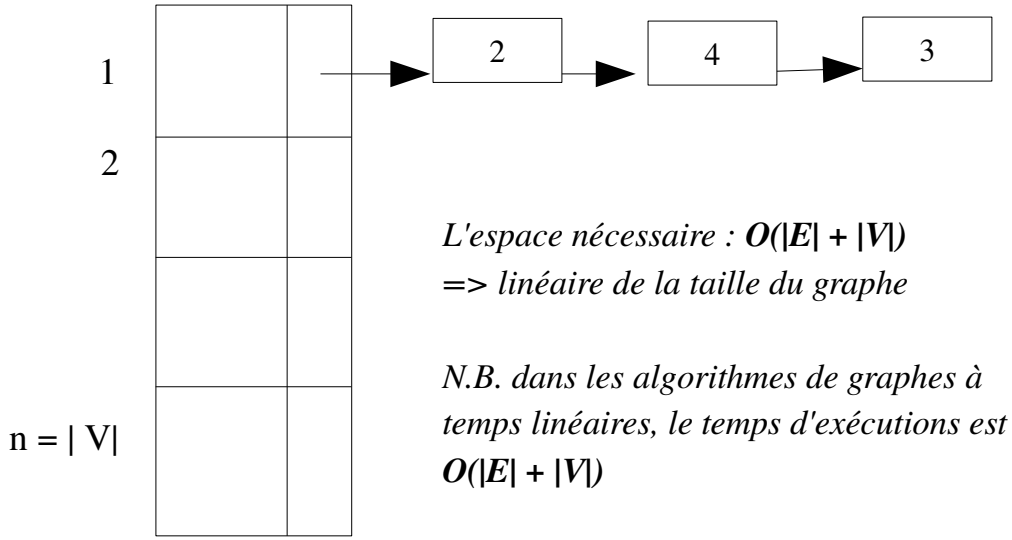
Une structure de données plus dynamique (adaptée pour un graphe peu dense) :

Pour un graphe <u>non orienté</u> :

Chaque arête *(u, v)* apparaît dans

<u>deux listes</u> d'adjacences

➟ l'espace nécessaire est **double**.



*L'espace nécessaire : **O**(|**E**| + |**V**|)*
=> linéaire de la taille du graphe

*N.B. dans les algorithmes de graphes à temps linéaires, le temps d'exécutions est **O**(|**E**| + |**V**|)*

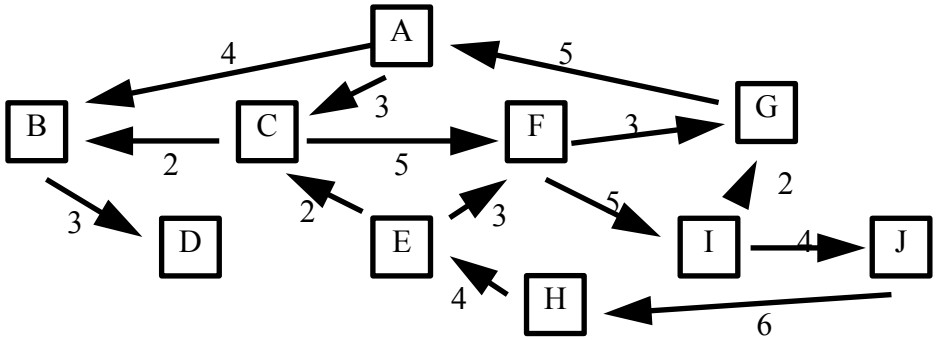
Cette représentation est la plus utilisée (donne satisfaction) en termes de complexité en temps/espace d'accès.

Remarque : le premier tableau peut être également une liste.

De même, la liste des adjacents peut être un **vecteur** (mélange statique / dynamique).

II.2.3- Représentation Hétérogène

Le graphe :



Représenté par deux tableaux / Listes :

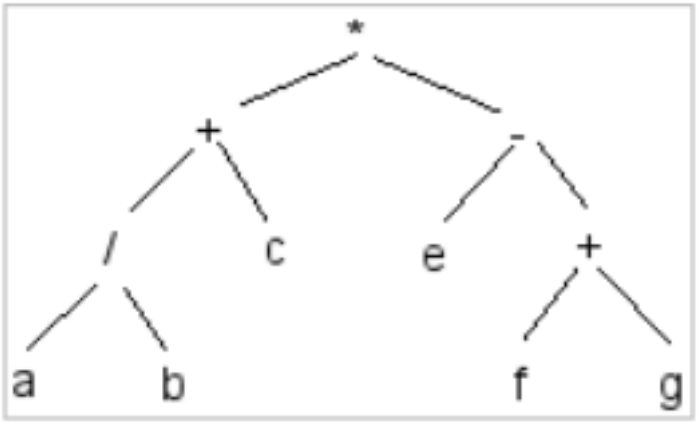
- V **pour les nœuds (toutes infos sur noeuds)**
- E pour les arcs + valeur

Dans le tableau E (à droite), la première ligne indique qu'il y a un arc du nœud A (indice 1 dans V) vers B (indice 2) avec un coût de 4.

	Noeuds
1	A
2	B
3	C
4	D
5	E
	F

début	*fin*	*coût*
1	**2**	**4**
1	**3**	**3**
2	**4**	**3**
3	**2**	**2**

Un autre exemple de représentation Hétérogène (Python)



indice	Noeuds	Fils
0	'*'	(1,6)
1	'+'	(2,5)
2	'/'	(3,4)
3	'a'	(None, None)
4	'b'	(None, None)
5	'c'	(None, None)
6	'-'	(7,8)
7	'e'	(None, None)
8	'+'	(9,10)
9	'f'	(None, None)
10	'g'	(None, None)

→ Il est possible de représenter un graphe par une liste de listes en Python.

Voir Cours 1A pour un complément.

II.2.4- Un type (TDA) Graphe

Sorte Graphe, It % It vaut *Itérateur*
Utilise ?: Bool, Élément, Liste

graphe_Vide :		➟ Graphe	
est_vide?:	Graphe	➟ Bool	
adjacents?:	Graphe x It	➟ **Liste**	-- Liste d'élément
adjacents?:	Graphe x Elément	➟ **Liste**	
recherche?:	Graphe x Elément	➟ It	
existe?:	Graphe x Elément	➟ bool	
insère?:	Graphe x Elément	➟ Graphe	
premier?:	Graphe	/➟ Élément	
premier?:	Graphe	➟ It	
suivant ?:	Graphe x Elément	/➟ Élément	
suivant ?:	Graphe x Elément	➟ It	
nœud_courant?:	Graphe	➟ It	
nœud_courant?:	Graphe	/➟ Elément	
nœud_suivant?:	Graphe x Elément	➟ Elément	
nœud_suivant:	Graphe x Elément	➟ It	

Les opérateurs sur les Itérateurs *(cf. les itérateurs)*

Début?:	Liste	➟ It
Next?:	It	/➟ It
Pred?:	It	➟ It
Déref?:	It	/➟ Élément
Valide?:	It	➟ bool

Pré-conditions et Axiomes?: ....

III- Algorithmes notables de parcours de graphes (récursifs / itératifs)

Deux types majeurs de parcours :

1- **En profondeur**

Avantages en inconvénients

2- **En largeur**

Avantages et inconvénients

⇒ Il y a également des parcours **ad-hoc**

Remarque : selon le "moment" où l'on traite l'information d'un nœud, on a différents types de traitements : *pré-ordre, post-ordre* et *mi-ordre*.

Remarque : on se place dans le cas d'un parcours en *pré-ordre (préfixé)*.

- Des deux types de parcours notables, on peut obtenir des parcours variés (*A*, *A**, …).

Voir plus loin, en particulier dans le cadre d'un parcours en **largeur**.

IV- Le principe de parcours récursif en profondeur (pré-ordre)

- Traiter chaque nœud puis traiter son premier adjacent récursivement avant de traiter les autres adjacents (cf. BE Cavalier)
- Éviter de retraiter un nœud en **marquant** les nœuds visités

- Ce parcours est semblable au parcours en profondeur des arbres.
- Ce parcours est en général assez performant mais si le graphe contient un cycle "à gauche", alors aucune réponse ne pourra être produite.

- **NB** : Pour un graphe G non_vide, **nœud_courant(G)** représente le nœud actuellement référencé (comme la racine dans un arbre) dont l'information et les adjacents :

Parcours récursif en profondeur avec marquage :

```
Procédure profondeur (G?: ref nœud) =
Début
   Si Non est_vide(G) alors
      marquer(nœud_courant(G));
      traiter(nœud_courant(G));          //traitement quelconque
      Pour X dans adjacents(nœud_courant(G))
         Si X n'est pas marqué alors
         profondeur(X);
         Fin si;
      Fin pour;
   Fin si;
Fin profondeur ;
```

- Les mécanismes de **marquage** :

  - Marquage à l'intérieur du nœud

  - Marquage par une structure de données externe au graphe (un tableau, …)

Une autre version :

```
Procédure profondeur (G?: ref nœud) =
Début
    Si Non est_vide(G) ET nœud_courant(G) n'est pas marqué alors
        marquer(nœud_courant(G));
        traiter(nœud_courant(G));          // traitement quelconque
        Pour X dans adjacents(nœud_courant(G))
            profondeur(X);
        Fin pour;
    Fin si;
Fin profondeur ;
```

Dans cette version, plus besoin de tester le marquage avant l'appel récursif.

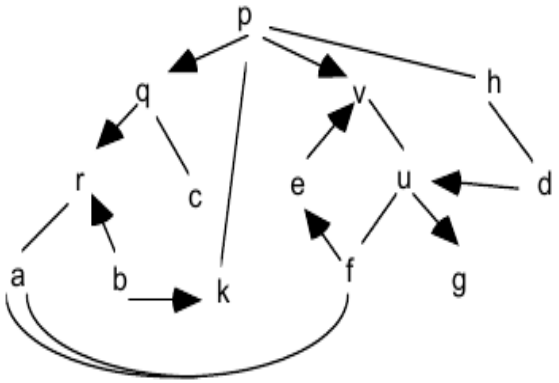
IV.1.1- Trace de parcours en profondeur

- Trace du parcours en profondeur de **p** à **u**

  profondeur(**p**) ➟ profondeur(q) ➟ profondeur(r) ➟

  profondeur(a) ➟ profondeur(f) ➟ profondeur(e) ➟

  profondeur(v) ➟ profondeur(**u**)



Principe de l'algorithme itératif en Profondeur?:

```
Procédure profondeur_iteratif(G)
    déclarer Pile=vide
    empiler(noeud_courant(G))
    Tant que NON vide(Pile)
        Noeud ← dépiler(Pile)
        Si NON est_marqué(X) Alors          – dans certains cas, un noued peut se trouve 2 fois dans la Pile (voir trace ci-dessus).
            marquer et traiter(noeud)
            Pour X dans adjacents(Noeud)
                Si NON est_marqué(X) Alors empiler(Pile, X)    Fin Si          – empiler dans le désordre
            fin pour
        FinSi
    Fin Tant que
Fin profondeur_iteratif
```

➟ Trace de la pile de **p** cette fois à **d**:

- Trace de parcours en profondeur de **p** à **d** (on souligne si traité) :

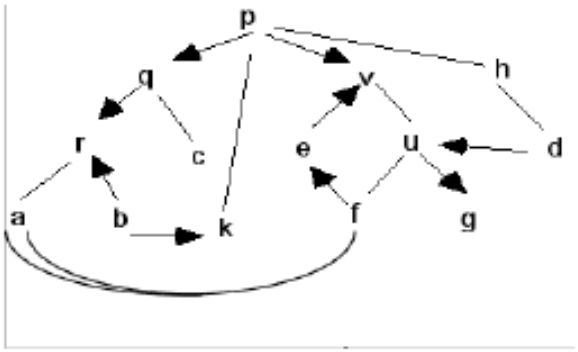
  [p] → [h,v,k,q] → [h,v,k,c,r] → [h,v,k,c,r]→ [h,v,k,c,a]→ [h,v,k,c,f]

  → [h,v,k,c,e,u]→ [h,v,k,c,e,g,v] ➟ v se trouve 2 fois dans la pile sans être marqué

  → [h,v,k,c,e, g] → [h,v,k,c,e]→ [h,v,k, c]→ [h,v, k] → [h, v]

  ➟ **v** : la 2e fois ! déjà traité. puis

  → [h] → [d] → []



V- Le principe de parcours en largeur

- Traitement par niveau : traiter chaque nœud
- Puis traiter chacun de ses successeurs avant de traiter les successeurs du prochain niveau.

- Parcours moins performant que le précédent et plus gourmand en mémoire.
- S'il existe un cycle dans le graphe, des réponses seront néanmoins produites.
- Ce parcours s'adapte mieux à une solution itérative :

  → On utilise **une file d'attente** pour la construction de la liste des nœuds à traiter.

V.1- Le Type (TDA) File

- Les opérateurs de la File d'attente *File(Élément)* utilisée :

File_vide?.		➟ File
Enfiler?.	Élément x File	➟ File
Défiler?.	File	/➟ File
Sommet?.	File	/➟ Élément
Est_file_vide?.	File	➟ Booléen

Pré-conditions?. pour f?. File;
Défiler(f)?. non Est_file_vide(f)
Sommet(f)?. non Est_file_vide(f)

Axiomes?. pour f?. File; e?. Élément
Sommet(Enfiler(e, File_vide))=e
Sommet(Enfiler(e, f))=Sommet(f)
Est_file_vide(File_vide) =vrai
Est_file_vide(Enfiler(e, f))=faux,

V.2- L'algorithme récursif de parcours en largeur

Cet algorithme fonctionne pour les graphes connexes. Voir la suite pour les autres.

```
File?: file d'attente =File_vide;
Procédure largeur_récursif (G?: ref nœud) =
Début
  Si Non est_vide(G) alors
     traiter(nœud_courant(G));        //une opération quelconque
     Pour X dans adjacents(nœud_courant(G));
              File=Enfiler(X, File);
     Fin pour;
     X=Sommet(File);
     File=Défiler(File);              // car défiler ne renvoie pas un élément (cf. TDA File)
     largeur_récursif (X);
  Fin si;
Fin largeur_récursif ;
```

N.B. : pas de **marquage.**

→ en l'absence de marquage, certains nœuds sont traités **plusieurs fois.**

Cas de graphe connexe : on travaille avec une file d'attente

```
File?: file d'attente =File_vide;
enfiler( la racine du graphe G, File)

Procédure largeur_récursif (File) =
Début
  Si Non est_vide(File) alors
     X=Sommet(File);
     File=Défiler(File);                    // car défiler ne renvoie pas un élément (cf. TDA File)
     traiter(nœud_courant(X));              //une opération quelconque
     Pour X dans adjacents(nœud_courant(G));
              File=Enfiler(X, File);
     Fin pour;
     largeur_récursif (File);
  Fin si;
Fin largeur_récursif ;
```

Initialement, il faut enfiler le nœud de départ.

Initialement, il faut enfiler le noeud de départ.

Voir la version Python du problème de la monnaie.

On peut récupérer le chemin par le mécanisme **Coming-From (CF).**

```
File?: file d'attente =File_vide;
enfiler(la racine du graphe G, File)
CF : tableau indicé par les nœuds initialisé à 0;
CF[Départ]=Départ

Procédure chemin_largeur_récursif (File) =
Début
  Si Non est_vide(File) alors
     X=Sommet(File);
     File=Défiler(File);                    // car défiler ne renvoie pas un élément (cf. TDA File)
     traiter(nœud_courant(X));          //une opération quelconque
     Pour X dans adjacents(nœud_courant(G));
              File=Enfiler(X, File);
              CF[X]=nœud_courant(G)
     Fin pour;
     largeur_récursif (File);
  Fin si;
Fin largeur_récursif ;
```

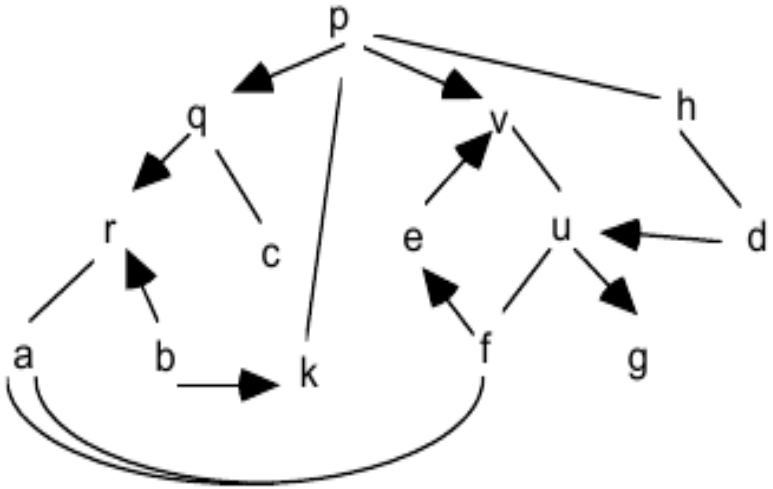
V.3- L'algorithme itératif de parcours en largeur

```
File?: file d'attente=File_vide;
Procédure largeur_itératif ( G?: graphe)
déclarer File=vide
Début
  Si Non est_vide(G) alors
        Enfiler(nœud_courant(G), File);
  Fin si;
  Tant que Non est_vide(File)
        N=Sommet(File);
        File =Défiler(File);
        traiter(N);     // ou visiter(N)
        Pour X dans adjacents(N);
              File=Enfiler(X, File);
        Fin pour;
  Fin Tant que;
Fin largeur_itératif ;
```

N.B. : version sans marquage

V.3.1- Trace de parcours en largeur

- **Exemple** : différents parcours illustrés sur le graphe suivant.



Trace de la version non marquée.

Parcours en largeur de **p** à **u**

(noter le cycle P ➟ ... ➟ P ...) :

largeur(**p**) ➟ largeur(q) ➟ largeur(k) ➟

largeur(v) ➟ largeur(h) ➟ largeur(r) ➟

largeur(c) ➟ largeur(p) ➟ largeur(**u**)

L'évolution de la File lors de ce parcours (ajout tant que le sommet ≠u) :

[] [p] [q, k, v, h] [k, v, h, r, c] [v, h, r, c , p] [h, r, c , p, u]

[r, c , p, u, p, d] [c, p, u, p, d, a] [p, u, p, d, a] [u, p, d, a, q, k, v, h]

➟ Destination atteinte.

V.4- L'algorithme itératif de parcours en largeur avec marquage

Ici, la File peut contenir des doublons qui ne seront pas traités une seconde fois.

```
File?: file d'attente=File_vide;
Procédure largeur_itératif_marquage ( G? graphe)
Début
  Si Non est_vide(G)
  Alors    Enfiler(nœud_courant(G), File);
  Fin si;
  Tant que Non est_vide(File)
        N=Sommet(File);
        File =Défiler(File);
        Si est_marqué(N) alors passer à l'itération suivante ;
        Sinon marquer(N);
        Fin si;
        traiter(N);     // ou visiter(N)
        Pour X dans adjacents(N);
              File=Enfiler(X, File);
        Fin pour;
  Fin Tant que;
Fin  largeur_itératif_marquage ;
```

V.4.1- Une variante parcours en largeur itératif (avec marquage)

Une seconde version avec marquage :

on marque les nœuds non marqués placés dans la file.

La file ne contiendra pas de doublon.

```
File?: file d'attente=File_vide;
Procédure premier_chemin_largeur_itératif ( G?: graphe)
Début
  Si Non est_vide(G) alors
     Enfiler(nœud_courant(G), File);
     marquer(nœud_courant(G));
  Fin si;
  Tant que Non est_vide(File)
        N=Sommet(File);
        File =Défiler(File);
        traiter(N);     // ou visiter(N)
        Pour X dans adjacents(N) non marqués;
              File=Enfiler(X, File);
              marquer(X);
        Fin pour;
  Fin Tant que;
Fin premier_chemin_largeur_itératif ;
```

VI- Applications des parcours de graphes

Rappel : pour éviter les boucles dans les algorithmes :

➟ marquage des nœuds ou encore mémorisation du trajet.

VI.1- Exemple simple : comptage du nombre de nœuds

```
Fonction taille(G?: ref noeud)?: entier =          // en profondeur
   nb_noeuds?: entier=0;      ADJ?: Liste(noeuds)
   Début
      Si Non est_vide(G) alors
         Si Non est_marqué(noeud_courant(G))
         Alors   marquer(X);
                 nb_noeuds=nb_noeuds + 1;
                 ADJ=adjacents(noeud_courant(G), G);
                 Pour X dans ADJ
                    nb_noeuds = nb_noeuds + taille(X);
                 Fin Pour;
         Fin si;
      Fin si;
      Retourne nb_noeuds ;
   Fin Taille;
```

VI.2- Fonction recherche d'un Élément

La fonction de recherche de l'élément **X** dans un graphe connexe

→ renvoie une référence (pointeur) sur le nœud trouvé ou bien une référence vide:

```
Fonction recherche(X: élément; G?: ref_nœud)?: ref_nœud =        //En profondeur (ref_nœud = itérateur)
 Début
    Si est_vide(G)
    Alors    retourne ref_nœud_vide;
    Fin si;
    Si est_marqué(nœud_courant(G))
    Alors    retourne ref_nœud;              // déjà visité
    Sinon
       Si (nœud_courant(G)=X) alors retourne G;
       Sinon
          marquer(nœud_courant(G));
          ADJ = adjacents(nœud_courant(G),G);
          Pour N dans ADJ
                ref_nœud It=recherche(X,N) ;
                Si (It ! =  ref_nœud_vide) alors retourne It;  Fin si;
          Fin Pour;
          Retourne ref_nœud_vide;
       Fin si;
    Fin si;
Fin recherche;
```

VI.3- Recherche de chemin en Profondeur

- La fonction ***chemin*** entre deux nœuds dans un graphe qui produit un trajet s'il y a un chemin entre les nœuds.

- Le trajet=vide si pas de chemin entre ces deux nœuds.

- On suppose que le *nœud_courant(G)=Départ* (sinon, on s'y place d'abord).

```
Fonction chemin(Dép, Arr?. nœud; G?. Graphe)?. trajet =      // en profondeur
tr?. trajet=<>;                                              // trajet vide
Début
      Si est_vide(G) alors retourne <>; Fin si;              // la liste trajet vide
      Si (Dép=Arr) alors retourne <Arr>; Fin si;
      marquer(Dép);
      ADJ=adjacents(Dép, G);
      Pour N dans ADJ
         Si Non est_marqué(N) Alors
            tr=chemin(N, Arr, G) ;                           // de la forme  <N,..., Arr> si non_vide
            Si (tr ≠ <>) alors retourne <Dép>.tr;            //cons(Dép, tr)
            Fin si;
      Fin Pour;
      Retourne <>;
Fin chemin;
```

VI.3.1- Amélioration du calcul du chemin (en Profondeur)

Autre solution (trajet en paramètre) avec une meilleure gestion du trajet.

➡ On n'a plus besoin de marquer : le trajet sert aussi de mémoire de parcours.

```
Fonction chemin(Dép, Arr?: nœud; G?: Graphe; T: ES trajet)?: booléen =        // ES vaut dire?: entrée-sortie
Début                                                                         // (passage par référence)
    Si est_vide(G) alors retourne faux; Fin si;
    Si (Dép=Arr)   retourne vrai;   Fin si;
    ADJ=adjacents(Dép, G);
    Pour N dans ADJ
        Si N ∉ T alors      T1 = T.<N>;
                            Si      chemin(N, Arr, G, T1)
                            alors   T = T1; retourne vrai;
                            Fin si;
        Fin si;
    Fin Pour;
    retourne faux;
Fin chemin;
```

Appel : T=<D>;

Si chemin(D,A,G, T)=vrai ➡ T contient le trajet

VI.4- Autres algorithmes

- Test de connectivité :

  Est-ce que tous les nœuds peuvent être atteints depuis un nœud donné.

- Extraction des composants connexes

- Test de circuit

- Ordre topologique

- ....

Une Question : dans le BE cavalier, quel est le type du parcours ?

VII- Méta-Stratégies générales de résolution

Soit un ensemble de variables $X=\{X_1, ..., X_n\}$ et leurs domaines $D=\{D_1, ..., D_n\}$.

VII.1- Les techniques qui regardent en arrière

1. **Générer-Tester** (le plus **inefficace**) :

   Ne considère pas les contraintes lors d'affectation des variables.

   Choisit $X_n=d_n \in D_n$ tel que $\{X_1, ..., X_n\}$ satisfasse les contraintes

   ➥ trop tard pour se rendre compte des mauvais choix !

2. **Retour arrière** (BackTrack):

   Restreint le choix de $d_{k+1} \in D_{k+1}$ pour la variable X_{k+1} (suivant les contraintes)

   ➥ On essaie une valeur pour X_{k+1} en vérifiant les contraintes avec les valeurs (actuelles) de $X_1 .. X_k$.

⇨ Il y a quelques variantes de Retour arrière (intelligent, etc.)

VII.2- Les techniques qui regardent en avant

3. Forward Checking (BT + Arc consistancy)

En plus de BT, le choix de d_{k+1} laisse une chance à $X_{k+2}..X_n$

➡ Dans une forme partielle, on anticipe seulement sur X_{k+2}.

Les vérifications sont faites entre la **dernière** variable instanciée et les restantes.

Arc consistancy : vérification de consistance (vérité des tests) des arcs.

4. Look Ahead (BT + Path consistancy)

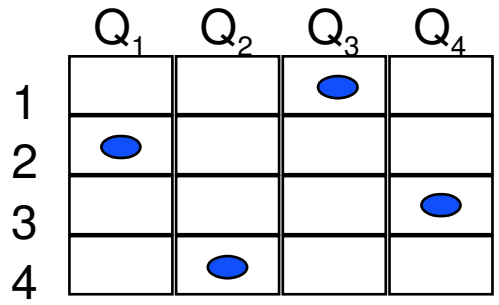
En plus de (FC), on vérifie la **satisfiabilité** d'une solution possible pour les autres variables (deux à deux) : on vérifie qu'il y aura non seulement une chance pour toute variable $X_{k+1}... X_n$ sachant $X_1..X_k$, mais qu'en plus, $X_{k+1}..X_n$ se laissent **deux à deux** une chance possible et satisfaisante. ../..

On remarque que les variables non encore instanciées sont testées deux-à-deux :

→ ce qui ne garantie pas qu'elles seront toutes compatibles.

De fait, dans certains problèmes simples, *Look Ahead* résout la totalité du problème dès les premières instanciations !

VII.2.1- Illustration : exemple N-reines



Placer 4 reines telles qu'elles ne s'attaquent pas (sur une ligne, colonne et diagonale)

Q_i = le numéro de ligne d'une reine dans la colonne i, 1 <= i <= 4

Les contraintes (*in extenso* pour la clarté) :

$Q_1, Q_2, Q_3, Q_4 \in \{1, 2, 3, 4\}$

$Q_1 \neq Q_2$, $Q_1 \neq Q_3$, $Q_1 \neq Q_4$,

$Q_2 \neq Q_3$, $Q_2 \neq Q_4$,

$Q_3 \neq Q_4$,

$Q_1 \neq Q_2-1$, $Q_1 \neq Q_2+1$, $Q_1 \neq Q_3-2$, $Q_1 \neq Q_3+2$,

$Q_1 \neq Q_4-3$, $Q_1 \neq Q_4+3$,

$Q_2 \neq Q_3-1$, $Q_2 \neq Q_3+1$, $Q_2 \neq Q_4-2$, $Q_2 \neq Q_4+2$,

$Q_3 \neq Q_4-1$, $Q_3 \neq Q_4+1$

	Q_1	Q_2	Q_3	Q_4
1				
2				
3				
4				

Méthode Générer-tester :

Au total, 256 évaluations :

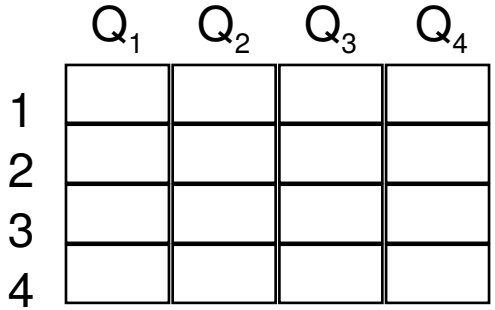
64 échecs avec **Q1=1** (4 x 4 x 4 = 64 possibilités pour Q2, Q3 et Q4)

....

---

un total de **115 évaluations** pour trouver la première solution.

Méthode Retour arrière :



Méthode Forward Checking (FC) : une couleur par Qi.

De gauche à droite et du haut vers le bas :

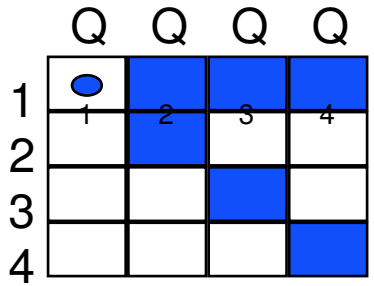
Q1=1 permet d'éliminer les cases bleues (et laisse {3, 4} à Q2)

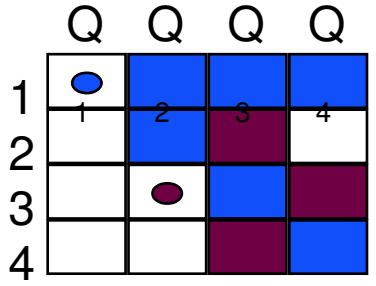
puis Q2=3 élimine les cases bordeaux (ne laisse rien à Q3);

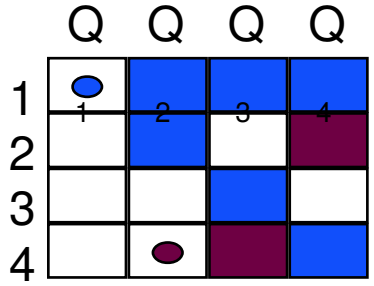
➡ on défait Q2=3

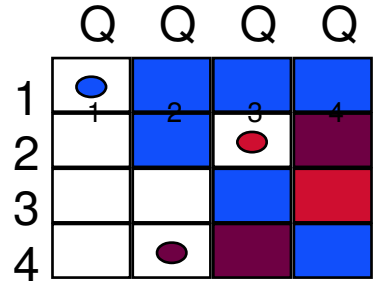
puis Q2=4 et Q3=2 ne laisse pas de chance à Q4.

On défait Q1=1 .....

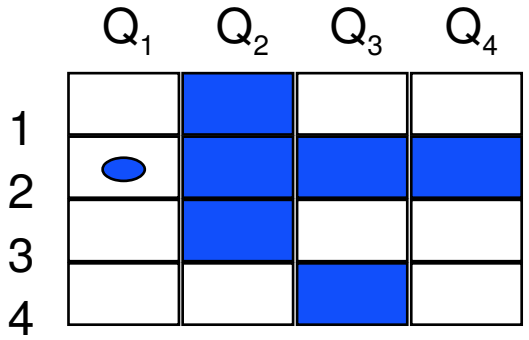


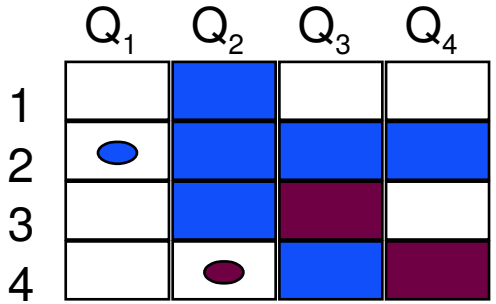


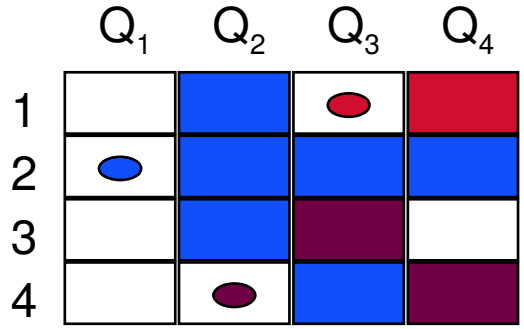


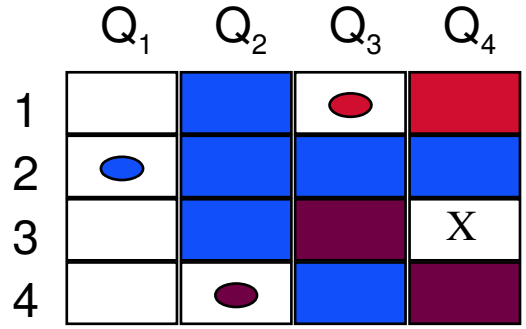


Suite FC après le placement de Q2, un examen indiv. de Q3 puis Q4 (anticipation)





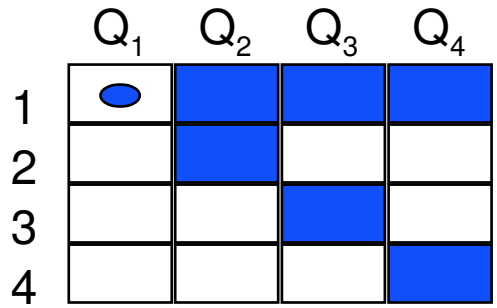


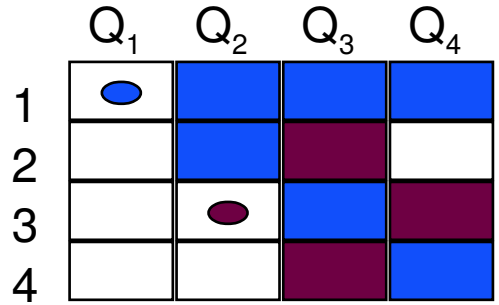


Méthode Look Ahead (LA): les couleurs impriment l'étendu des Qi.

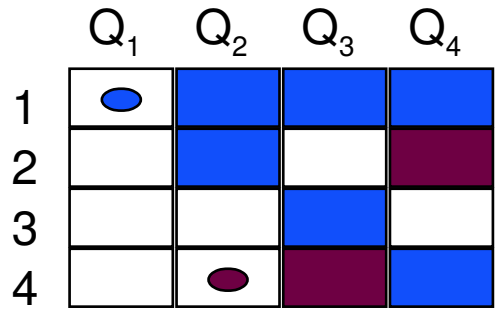
Q1 placé en <1,1> : cela laisse une chance 2-à-2 aux autres

Le placement de Q2 en <3,2> **n'aboutira pas** pour les mêmes raisons que dans la stratégie CF.





- On revient en arrière et on place Q2 e, <4,2>.
- Mais Q2 en <4,2> ne laisse pas une chance (2-à-2) à Q3 et Q4 : leur seule possibilité est incompatible :

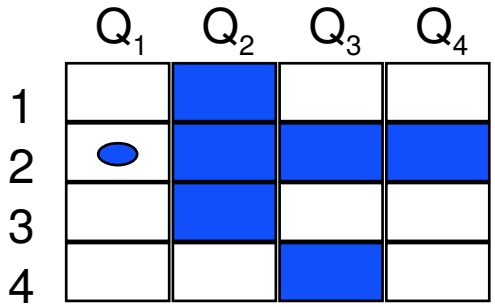


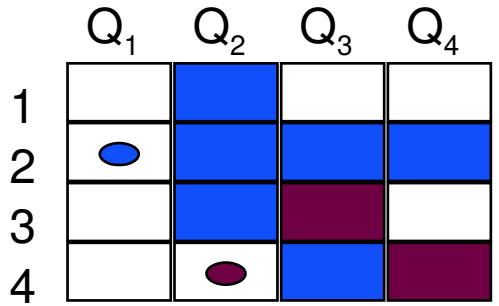
Échec de placement de Q2 **si Q1 est en <1,1> :** on revient en arrière. ../..

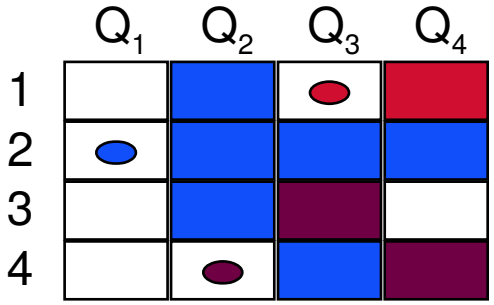
On passe à la 2e possibilité de Q1 : Q1 en <2,1> laisse à Q2 la case <2,4>.

Après le placement de Q2 en <4,2> **:** on vérifie une chance individuelle pour Q3 et Q4

+ une chance de respecter les contraintes (2 à 2) entre Q3 et Q4.







VIII- Aspects pratiques du Back-tracking

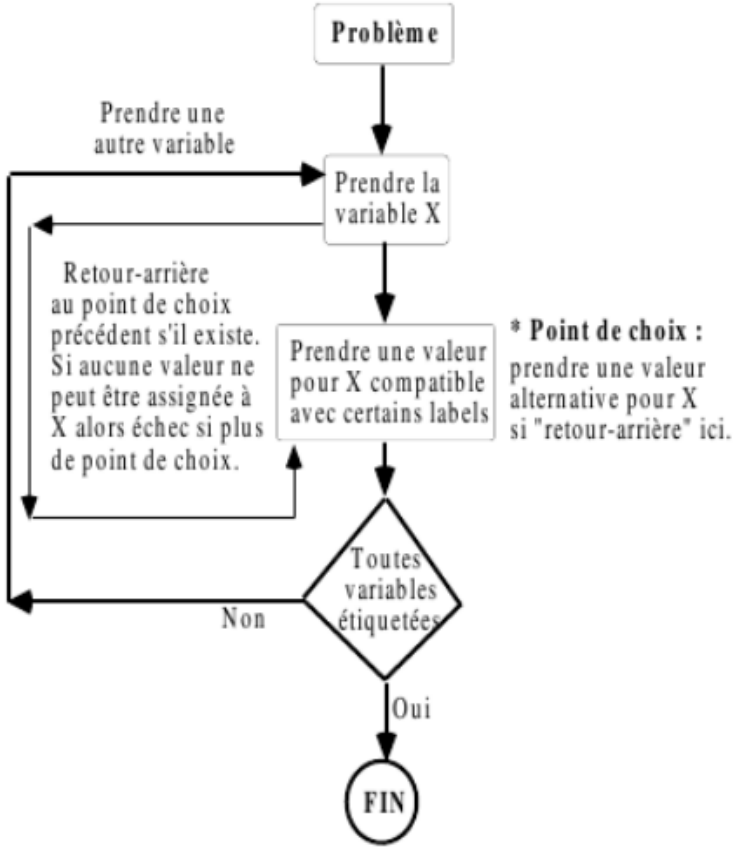
- L'opération de base dans la résolution :

  - prendre une variable à la fois

  - lui donner une valeur à la fois

  - s'assurer que le label ainsi constitué est compatible avec tous les autres labels construits jusqu'à ce stade.

- L'algorithme "retour-arrière chronologique" (*chronological back-tracking* ***BT***) est une stratégie générale de recherche largement utilisée dans la résolution des problèmes.



Contrôle de l'algorithme de retour-arrière chronologique

VIII.1- Algorithme de principe du Back-tracking

- On détaille ici davantage les aspects pratiques de la stratégie "retours arrières" (AES, Back-Tracking).

- Les algorithmes de parcours peuvent mémoriser les transitions possibles depuis un nœud donné.

- La stratégie générale **Back-Tracking** (BT = retour arrière ou encore stratégie à essais successifs) permet, au besoin, de retrouver toutes les solutions à un problème donné.

- Un exemple typique de cette stratégie est une balade dans un labyrinthe.

```
Procédure Back_track(Graphe G, nœud N)        // également appelé " algorithme à essais successifs "
   Si N=destination Alors signaler succès;
   Sinon  Pour tout N' successeur de N dans G
                Si          Prometteur(N')
                Alors       Back_track(G, N')
                Fin Si
          Fin Pour
   Fin Si
Fin Back_track
```

La fonction ***Prometteur(nœud N)*** vérifie si le ne nœud N permet d'avancer en respectant les contraintes.

Par exemple, dans le cas du problème N-reines, vérifier si le placement d'un pion (une reine) ne remet pas en cause les contraintes du problème.

VIII.2- Applications de BT

Des exemples d'illustration de BT sont :

- Problème de N-reines
- La somme des sous-séquences (une version simplifiée du problème Sac-à-dos)
- Coloration de graphes
- Problème de circuit Hamiltonien
- Sac à dos généralisé
- Parcours du Cavalier (BE)
- Etc.

VIII.2.1- Exemple-1 : placements sous conditions (N-reines)

Représentation :

On décide de consacrer une colonne par pion (simplification des conditions).

Un tableau *Colonne[1..N]* d'entiers donnera la solution où *Colonne[i]* contiendra le numéro de la ligne où le pion est placé.

L'algorithme suivant donnera toutes les solutions au problème de N_reines.

```
Procédure N_reines (Colonne?: tableau [1..N] d'entier, indice?: index)
  Si Prometteur (Colonne, indice) Alors
     Si (indice = N)  Alors renvoyer  Colonnes[1..N]          // Une solution
     Sinon
        Pour j=1..N                                  // Donnera toutes les solutions
           Colonne[indice+1] = j;
           Si N_reines (Colonne, indice+1)        // Pour  toutes les solutions, ne conserver de ces 2 lignes  que N_reines (Colonne, indice+1)
           Alors  renvoyer  Colonnes[1..N]
           FinSi                  // Pas besoin de modifier la case Colonne[indice+1], sa valeur  sera modifiée dans cette boucle.
        Fin Pour
     Fin Si
  Fin Si
Fin N_reines
```

La fonction *Prometteur* vérifie les conditions de placement

```
Bool Prometteur(Colonne?: tableau [1..N] d'entier, i:index)
  k=1;
  est_prometteur = vrai;
  Tant que k < i   ET  est_prometteur
     Si (Colonne[i] = Colonne[k]) OU (abs(Colonne[i] - Colonne[k]) = i - k)
     Alors
        est_prometteur = faux
     Fin Si
     k = k+1
  Fin Tant que
  Retourner est_prometteur;
Fin Prometteur
```

Rappel : le tableau *Colonne* contiendra les No lignes où des pions seront placés.

```
// Initialiser la tableau Colonne à (p.ex. -1). Non obligatoire car l'indice 0 permettra à  Prometteur de ne pas tester
Appel?: N=taille(Colonne) ;
        afficher(N_reines(Colonne, 0));
```

N.B. : Si on décide d'utiliser FC, la fonction *Prometteur* vérifiera si l'occupation d'une case laisse un voisin possible à cette case.

N.B. : voir aussi le BE sur le Cavalier et les heuristiques utilisées.

Le code Python de cette version :

```
N=5 # On va jsq indice N      (indice 0 non utilisé)
def N_reines(Lst_num_Colonnes, indice) :
    if Prometteur_reuse_this(Lst_num_Colonnes, indice) :
        if indice == N :
            return Lst_num_Colonnes
        else :
            for j in range(1,N+1) :
                Lst_num_Colonnes[indice+1]= j
                if N_reines(Lst_num_Colonnes, indice+1) :
                    return Lst_num_Colonnes

def Prometteur(Lst_num_Colonnes, indice) :
    k=1 ; est_prometteur = True
    while k < indice and est_prometteur : # quand indice=1 : on n'a encore rien fait
        if Lst_num_Colonnes[indice] == Lst_num_Colonnes[k] or \
                    abs(Lst_num_Colonnes[indice] - Lst_num_Colonnes[k]) == indice - k :
            est_prometteur = False
        k += 1
    return est_prometteur
```

```
# MAIN
Lst_num_Colonnes=[-1 for i in range(N+1)]
# Lst_num_Colonnes[1]=1
print(N_reines(Lst_num_Colonnes, 0)) # On aura [-1, 1, 3, 5, 2, 4]
```

Une version AES de ce même algorithme (et Python) :

```
def N_reines_AES(Lst_num_Colonnes, indice_a_traiter) :
   if indice_a_traiter == len(Lst_num_Colonnes) : return Lst_num_Colonnes
   for val_colonne in range(1,len(Lst_num_Colonnes)) :
      Lst_num_Colonnes[indice_a_traiter] = val_colonne
      if Prometteur(Lst_num_Colonnes, indice_a_traiter) :
         if N_reines_AES(Lst_num_Colonnes, indice_a_traiter+1) :
            return Lst_num_Colonnes
         else : Lst_num_Colonnes[indice_a_traiter] = -1
   return None

def Prometteur(Lst_num_Colonnes, indice) :        // la même que la précédente.
   k=1 ; est_prometteur = True
   while k < indice and est_prometteur :
      if Lst_num_Colonnes[indice] == Lst_num_Colonnes[k] or \
        abs(Lst_num_Colonnes[indice] - Lst_num_Colonnes[k]) == indice - k :
         est_prometteur = False
      k += 1
   return est_prometteur
# ----------
```

```
# MAIN
Taille=5
Lst_num_Colonnes=[-1 for i in range(Taille+1)]
print(N_reines_AES(Lst_num_Colonnes, 1))
# On obtient [-1, 1, 3, 5, 2, 4]        sachant que l'indice 0 n'est pas utilisé (on commence à 1)
```

VIII.2.1.a- Complexité de N reines (en termes du nombre d'états visités)

D'une manière générale (stratégie gen-test), si l'on dessine un arbre (espace d'états) où les nœuds représentent les états successifs (en partant de i=0), on aura :

- 1 nœud (i=0) à la racine
- N nœuds au niveau 1
- N^2 nœud au niveau 2.
- N^N au niveau N

Le total des nœuds de l'espace d'états = $1+N+N^2+...+N^N = \frac{N^{N+1}-1}{N-1}$ = **$O(N^N)$.**

Un exemple : pour N=8, il y aura un total de 19 173 961 nœuds (combinaisons / états possibles).

Remarque : on peut considérer les nœuds prometteurs en évitant de placer un pion sur une colonne déjà occupée à l'aide de la structure de données choisie (ici, le tableau *Colonne*).

Par exemple, pour N=8, on aura la série 1 + 8 + 8 x 7 + 8 x 7 x 6 + ...+ 8!

Pour N généralisé, on aura la série 1 + N + N(N-1) + N(N-1)(N-2) + ... + N! = e. N! = O(N!)

Une **comparaison** des complexités montre le gain important, pour N grand.

- Soit *Algo1* : un algorithme de parcours en profondeur de l'espace **sans BT** : **$O(N^N)$**
- *Algo2* : ci-dessus : génère *N!* Candidats qui placent chaque pion à une ligne et colonne différente.

N	*nœuds vérifiés par Algo1*	*nœuds vérifiés par Algo2*	*nœuds vérifiés par un BT simple*	*nbr nœuds prometteurs trouvés par BT*
4	341	24	61	17
8	19 173 961	40320	15 721	2057
12	$9.73 * 10^{12}$	$4.79 * 10^{8}$	$1.01 * 10^{7}$	$8.56 * 10^{5}$
14	$1.2 * 10^{16}$	$8.72 * 10^{10}$	$3.78 * 10^{8}$	$2.74 * 10^{7}$

Le tableau montre l'efficacité de la méthode BT + Prometteur.

Remarque sur une technique inspirée de la programmation sous contraintes :

Les stratégies FC et LA peuvent être implantées par une technique de **propagation de contraintes**.

→ L'algorithme N-reines peut profiter de cette technique.

Dans ce cas, lorsqu'un pion est placé, on élimine, pour les autres pions les valeurs qu'ils ne pourront pas prendre.

Par exemple, en plaçant un pion en <1,1>, on peut immédiatement éliminer les cases sur la ligne 1, colonne 1 et la diagonale.

De plus, cette technique autorise l'utilisation des méta-Stratégies telles que *Look-Ahead*.

VIII.2.2- Exemple-2 : le problème de la somme des sous ensembles

Cet exemple est REPRIS dans la section "Programmation Dynamique".

Un cas simplifié du problème de **Sac-à-dos**.

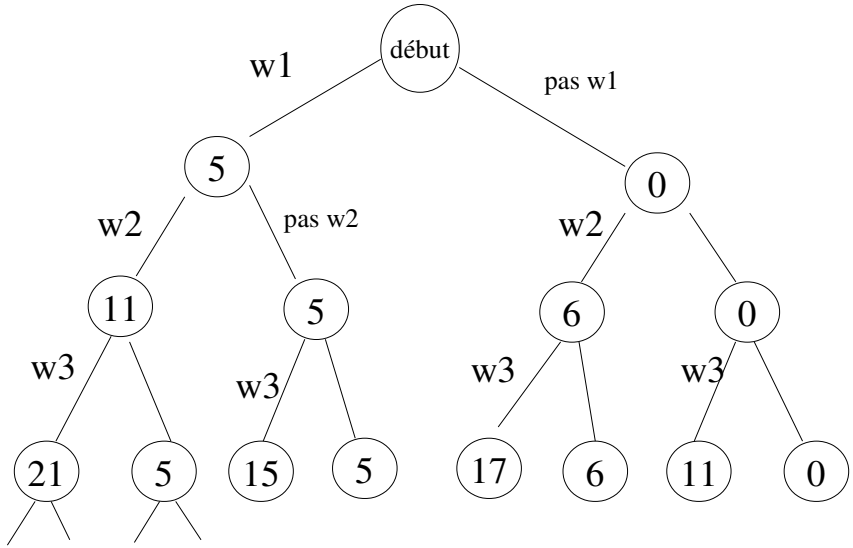
Énoncé : un voleur muni d'un sas-à-dos vole des objets de valeur dans une maison.

- Ces objets ont chacun un poids unique et génère un profit (valeur en cas de vente).
- Le but du voleur est de prendre un maximum d'objets sans dépasser la capacité de son sac tout en maximisant le profit.
  - Dans le cas du problème actuel, on considère **le même profit pour chaque objet**.

Ce cas simplifié est connu sous le nom du *problème de la somme des sous ensembles*.

Remarque : ce problème est différent du problème des sommes du chapitre 1 où la séquence de somme maximum considérée devait être contiguë. Pour ce problème-là, une variante aura été de trouver la séquence la plus courte/longue avec une somme sous contrainte (e.g. la somme restant inférieure à une certaine limite).

Revenons à notre problème : soit **N** le nombre d'objets et par **W** le poids maximum que le sac à dos pourra supporter sans se déchirer.



Exemple (avec son espace d'états = arbre en face) :

N=5 et W=21 avec les objets de poids :

w1= 5, w2 = 6, w3 = 10, w4 = 11, w5 = 16

➥ Trois solutions possibles :

{w1, w2, w3}, {w1, w5} et {w3, w4}

Un algorithme BT classique examinera toutes les combinaisons possibles (tous sous ensembles : complexité O(N!))

Améliorations : une sorte de **propagation de contraintes** pour tenir compte de :

- Soit **Poids_so_far** = la somme des poids des objets déjà ramassés (w_1 ... w_i)

  Si *Poids_so_far + wi_{+1} > W,* l'objet w_{i+1} ne sera pas sélectionné

  ➥ On dira que le nœud w_{i+1} n'est pas Prometteur. Pas la peine d'aller plus loin.

- Soit **Total_poids_restants** = la somme des poids des objets non encore sélectionnés

  Après avoir inclus un nœud (w_j) :

  Si *Poids_so_far + Total_poids_restants < W* alors on sait par avance que le nœud ac tuel ne sera pas Prometteur (car on ne pourra jamais être égal à W; ce qui est le but).

Une dernière information peut être également utile :

- Si l'ajout du poids de l'objet *wj* est tel que *Poids_so_far =W*, il est évident qu'aucun autre nœud (état suivant l'actuel état) ne doit être recherché (car poids uniques).

Ces trois cas peuvent être illustrés sur l'espace d'état précédent.

L'algorithme **Somme_des_sous_ensembles** suivant utilisera les données suivantes :

N : nombre d'objets

$\mathbf{w_i}$: le poids de l'objet i (par le tableau w[1..N])

Poids_so_far : la somme des poids des objets déjà ramassés

Inclus : tableau de booléens tel que *Inclus[i] = vrai* veux dire : wi est ramassé.

Total_poids_restant : la somme des poids des objets restants

W : le poids maximum à atteindre

Rappel : tous les objets ont un même profit.

L'algorithme (BT optimisé qui est une sorte de A*) suivant donnera **toutes les solutions** pour la séquence w[1 .. i] telle que la somme des wj =W.

```
Procédure Somme_des_sous_ensembles(i?: index;  Poids_so_far, Total_poids_restants?: entier)
  Si Prometteur(i)
  Alors
     Si     Poids_so_far = W
     Alors émettre la solution Inclus[1..i]
     Sinon
        Inclus[i+1] = vrai
        Somme_des_sous_ensembles(i+1, Poids_so_far + w[i+1], Total_poids_restants - w[i+1])
        Inclus[i+1]= faux        // pour essayer d'autres solutions
        Somme_des_sous_ensembles(i+1, Poids_so_far, Total_poids_restants - w[i+1])
     Fin Si
  Fin Si
Fin  Somme_des_sous_ensembles

Bool Prometteur(i?: index) =
     Si  (Poids_so_far + Total_poids_restants >= W)  &&  (Poids_so_far = W Ou Poids_so_far + w[i+1] =< W)
     Alors retourner   Vrai
     Sinon retourner   Faux
     Fin Si
Fin Prometteur
-----------------------------------------------------------------------------------------------------
Appel?  Somme_des_sous_ensembles(0, 0, Total_tous_les_objets)
            où  Total_tous_les_objets = la sommes de tous les poids wi
```

Complexité : le nombre de nœuds dans l'espace d'états recherché par l'algorithme est

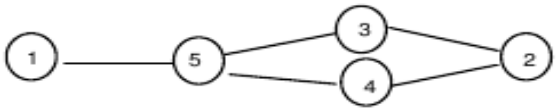
$$1 + 2 + 2^2 + \ldots + 2^N = 2^{N+1} - 1 = O(2^N)$$

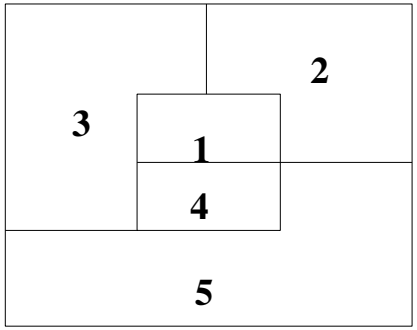
- La complexité reste la même si nous voulons **une seule solution**.
- Le **cas pire** : si la somme de tous les w_i, i=1..N < W mais w_n=W

  ➥ Ce qui nécessite de recherche **tout** l'espace de recherche.

VIII.2.3- Exemple-3 : Coloration de graphes

2 exemples de graphe





- **Objectif** : indice chromatique :

  Colorier ce graphe avec un minimum de couleurs différentes.

  Contrainte : deux nœuds voisins ne doivent pas la même couleur.

- Il y a plusieurs méthodes de coloration de graphes dont deux ci-dessous :

- **Méthode gloutonne** : on choisit une couleur puis on examine les nœuds dans l'ordre de leur numéro en affectant cette couleur à un maximum de nœuds en respectant la contrainte ci-dessus.

- **Méthode Heuristique** : même principe que ci-dessus mais le choix des nœuds se fait selon leur nombre de voisins.

  Après l'affectation d'une couleur à un nœud, on doit mettre à jour le nombre de voisins des nœuds en isolant du graphe le nœud qui vient d'être coloré.

VIII.2.3.a- Algorithme BT de coloration de graphes

Les données du problème :

- N : nombre de nœuds du graphe
- Nb_Couleurs : le nombre de couleurs disponibles
- Tab_Couleurs[1..N] : les couleurs données aux nœuds
- Graphe[1..N][1..N] : matrice d'adjacence représentant le graphe d'incompatibilités

```
Procédure coloration(i?: index)=
Si Prometteur(i)
Alors   Si  i=N
        Alors émettre la solution Tab_Couleurs[1..N]
        Sinon   Pour Couleur = 1 .. Nb_Couleurs          % tout  essayer
                   Tab_Couleurs[i+1] = Couleur;
                   coloration(i  +1);
                Fin Pour
        Fin Si
Fin Si
Fin coloration
```

```
Bool Prometteur (i:index) =
  est_prometteur=vrai ; j=1;
  Tant que j < i  &  est_prometteur
      Si Graphe[i][j] = vrai  &  Tab_Couleurs[i] = Tab_Couleurs[j]
      Alors est_prometteur = Faux
      Fin Si
      j = j + 1;
  Fin Tant que
  Retourner est_prometteur
Fin  Prometteur
```

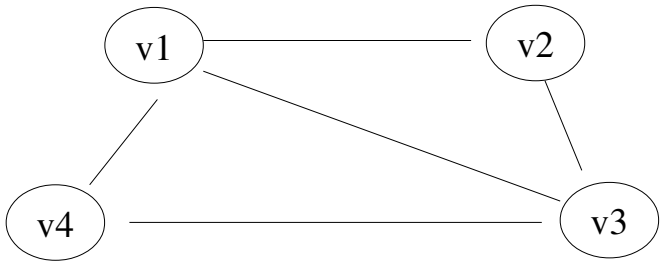
Appel? coloration(0);

N.B. : la formulation par un **graphe de compatibilités / incompatibilités** est une technique générale efficace dans de nombreux problèmes de la classe " affectation ".

→ exemple de feux de circulation

→ d'emploi du temps, ...

VIII.2.3.b- La complexité de l'algorithme BT de coloration



- Soit le graphe à colorier suivant.

- Si on dessine l'espace d'états de cet algorithme pour **M** couleurs,

on obtient l'espace-d'états partiel suivant (pour p.ex M=3) et le graphe de 4 nœuds ci-dessous.

- Les arêtes **doubles** donnent une solution utilisant 3 couleurs pour les 4 nœuds du graphe (M=3, N=4).

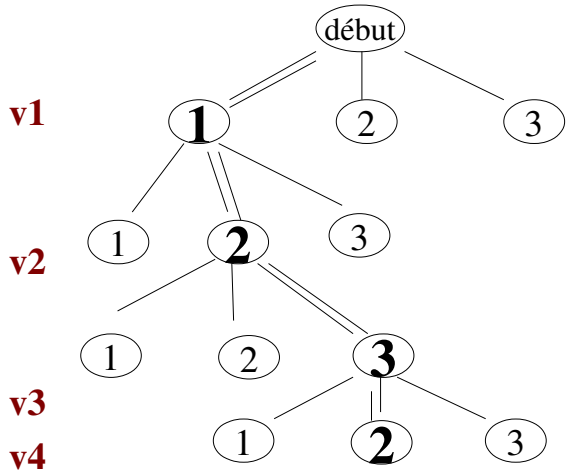
- A gauche, les Vi sont les nœuds traités à chaque niveau de cet espace (arbre ci-dessous)

- **La complexité de l'algorithme coloration** est la même que pour l'algorithme des N reines .

  - On mesure le nombre de nœuds possibles dans l'espace d'états.

  - Pour M couleurs et N nœuds, on a :

$$1+M+M^2+\ldots+M^N = \frac{M^{N+1}-1}{M-1} = O(M^N)$$



IX- Approches de conception d'algorithmes : stratégies

- Nous avons vu les (méta) Stratégies générales (*regard en arrière* et *en avant*)

  Ces principes sont applicables dans toutes stratégies

  ○

- Nous avons vu les parcours basiques de graphes
- Nous avons vu le principe de Back-Tracking

- Ci-dessous, un bilan général et la suite :
  - Stratégies Diviser et Régner
  - Stratégie Programmation Dynamique
  - Stratégies Greedy
  - Back Tracking (déjà vue)
  - Branch & Bound (proche mais à ne pas confondre avec la technique d'optimisation B&B)

N.B. : les Heuristiques telles que *Best First* (cf. coloration ou morpion optimisés), *Best Fit* (cf. Bin Packing), ... s'inscrivent dans différentes stratégies.

Elles sont bien adaptées aux parcours en largeur des graphes.

Voir plus loin pour des exemples.

IX.1- Stratégies Diviser et Régner

Le **principe de base** est de diviser le problème en sous problèmes, de traiter ces sous problèmes puis d'assembler les résultats. Les sous-problème sont en général indépendants.

IX.1.1- Exemples

- Multiplication (voir ci-dessous)
- Recherche dichotomique dans un tableau ou dans un ABOH
- Tri Fusion (Merge-Sort), Tri Rapide (Quick Sort)
- Calcul de Médiane (50eme centile ou percentile) :

  → la moitié est plus petite que la médiane, l'autre plus grand (v. chapitre 1)
- etc.

Nous avons vu plusieurs de ces exemples.

Voir le chapitre 1 de ce cours pour le Tri Fusion et le Tri Rapide.

IX.1.2- Multiplication de nombres binaires

- La stratégie Diviser-et-Régner peut se mettre en place par un schéma récurrent (pour la division, en particulier Dichotomique).
- Pour multiplier deux nombres binaires positifs, on peut procéder par cette stratégie (Carl Friedrich Gauss, 1777-1855).

Exemple : soit X=10110010 et Y=0110001. Pour les multiplier, on procède par :

XL = la moitié gauche de X = 1011 , XR = la moitié droite de X = 0010

YL = la moitié gauche de Y = 0110 , YR = la moitié droite de Y = 0001

Pour la taille n = maximum de celle de X et de Y, on procède ensuite par 3 multiplications Mul1, Mul2, Mul3 et une addition (Add) :

→ Mul1 : XL * YL ici 1011 * 0110

→ Mul2 : XR * YR ici 0010 * 0001

→ Mul3 : (XL+ XR) * (YL + YR) ici 1101 * 1001

→ Add : Mul1 * 2^n + (Mul3-Mul1-Mul2)*$2^{n/2}$+ Mul2.

L'algorithme de cette multiplication :

Fonction multiplier(X, Y? entiers positifs) : renvoie le produit X.Y
Début
n = max(taille de X, taille de Y)
Si (n = 1) renvoyer X*Y // cas basique
xL , xR = leftmost n/2 bits de X , rightmost n/2 bits de X
yL , yR = leftmost n/2 bits de Y, rightmost n/2 bits de Y
P1 = multiplier(xL , yL)
P2 = multiplier(xR , yR)
P3 = multiplier(xL + xR , yL + yR)
renvoyer $P1 \times 2^n + (P3 - P1 - P2) \times 2^{n/2} + P2$
Fin multiplier

Complexité d'un schéma récurrent (stratégie diviser-régner) : $T(n) = a\ T(\lceil n/b \rceil) + O(n^d)$ où

n = la taille du problème à chaque étape (ici 8 puis 4,2,1),

a : le cout lors de la division (ici a=3 car 3 appels récursifs),

b : facteur de division en sous-problèmes (ici b=2),

d : le cout de l'assemblage (ici d=1).

Dans le cas de la multiplication binaire, $T(n)=3\ T(\lceil n/2 \rceil) + O(n)$ ../.. (le calcul)

IX.1.3- Calcul de la complexité

On a $T(n) = 3\ T(n/2) + n$

avec $T(0) = 0$ et $T(1) = 1$

Posons $N = 2^k$. → $T(2^k) = 3\ T(2^{k-1}) + 2^k$

On pose $t_k = T(2^k)$: → $t_k = 3\ t_{k-1} + 2^k$

D'où : (1) $t_k - 3\ t_{k-1} - 2^k = 0$

Aussi (2) $t_{k-1} - 3t_{k-2} - 2^{k-1} = 0$

On divise (1) par la constante 2 puis on soustrait les deux pour faire disparaître les constantes :

(1) $t_k/2 - 3/2\ t_{k-1} - 2^{k-1} = 0$

(2) $t_{k-1} - 3\ t_{k-2} - 2^{k-1} = 0$

La soustraction donne l'équation de récurrence :

(3) $\frac{1}{2}\, t_k - 5/2\, t_{k-1} + 3\, t_{k-2} = 0 \quad \rightarrow \quad t_k - 5t_{k-1} + 6\, t_{k-2} = 0$

On pose $t_k = r^k$ et on obtient l'équation caractéristique : $r^k - 5\, r^{k-1} + 6\, r^{k-2} = 0$

D'où $r^{k-2}(r^2 - 5\, r + 6) = 0 \rightarrow r^{k-2}(r - 2)(r-3) = 0$

Les racines seront : $r = 0, 2, 3 \rightarrow t_k = c_1\ 2^k + c_2\ 3^k.$

L'exploitation des cas d'arrêt donne $c1 = -1$ et $c2 = 1$ d'où $\rightarrow t_k = 3^k - 2^k$

Sachant que $N = 2^k$, on a $k = log_2\ N$

$\rightarrow T(N) = 3^{\log N} - N = N^{\log 3} - N = O(N^{\log 3}) = O(N^{1.6})$

N.B. : pour N=1, k=0 et T(N) = O(N) car N = 1 (cas d'arrêt).

A l'autre extrémité, la complexité sera $T(N) = O(N^{\log 3})$.

Nota Bene :

Si N varie, ces termes représentent une suite géométrique d'un facteur (*raison*) de log3 (= 1.6).

La somme des termes d'une telle suite géométrique (N tend vers l'infini) est $N^{1+\log 3}$.

IX.2- Principe et Stratégie de Programmation Dynamique

- La Programmation Dynamique (Prd) est similaire à *Diviser-Régner* et découpe le problème P_k en sous problèmes P_j , $j<k$.
- La résolution des sous problèmes P_j a en général besoin d'un faible nombre d'information (par rapport à P_k).

- Mais contrairement à l'approche *descendante* du "Diviser pour Régner", la technique de la PrD procède ensuite par une approche *Ascendante* (*Buttom-up)* pour reconstituer/calculer la solution.
- Comparer par exemple la *multiplication binaire* (Ascendant) et le *calcul de la médiane* (descendant) :

→ On résout les instances simples et petites, on stock les résultats.

Plus tard, lorsque l'on a besoin de ces valeurs déjà calculées, on les utilise (pas de re-calcul) pour calculer les instances plus importantes.

- Néanmoins, une version de la PrD utilisant une approche *descendante (Top-Down)* existe également.

- N.B. : Le terme *PrD* vient de la *théorie du contrôle* où « *programmation* » veut dire que l'on utilise un tableau dans lequel les solutions sont construite

(Algorithm Design & applications : M. Goodrich & al. Wiley 2014).

Un exemple : calcul de *Fibonacci* et la version récursive versus l'utilisation d'un tableau qui stock les termes précédents).

→ Ce qui caractérise PrD est l'**approche Ascendante** :

Fib(0) et Fib(1) → Fib(2) → Fib(3) → … → Fib(N)

Les étapes de PrD :

1. Définir une propriété récursive qui donne la solution à une instance du problème
2. Résoudre une instance du problème de façon ascendante en résolvant d'abord les instances plus petites.

Quelques exemples PrD :

- Coefficient Binomial
- Algorithme de Floyd (chemins les plus courts entre toute paire de nœuds d'un graphe valué)
- Recherche dans un AVL (arbre binaire compacte et équilibré)
- Voyageur de commerce (TSP : *traveler sales Person*)
- Fib (en partant de 1 et jusqu'à N) est un exemple simple
- Factorielle est un cas trivial de PrD
- Distance d'édition (*Levenshtein*)

IX.2.1- Exemple 1 - calcul binomial

Coefficient binomial $$C_k^n = \begin{bmatrix} n \\ k \end{bmatrix} = \frac{n!}{k!.(n-k)!} \qquad pour \quad 0 \le k \le n$$

L'approche PrD de résolution de ce problème établit la **propriété récursive** :

$$C_k^n = \begin{bmatrix} n \\ k \end{bmatrix} = \left\{ \begin{array}{cc} \begin{bmatrix} n-1 \\ k-1 \end{bmatrix} + \begin{bmatrix} n-1 \\ k \end{bmatrix} & pour \quad 0 < k < n \\ 1 \quad pour \quad k=0 \quad ou & k=n \end{array} \right.$$

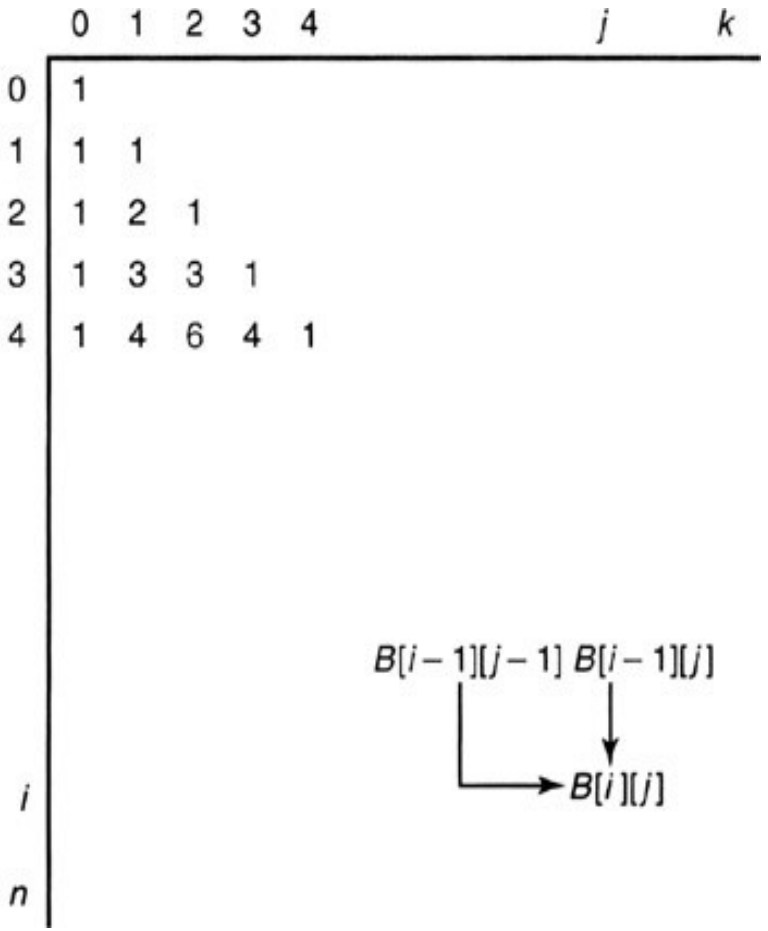
Ce qui donnera l'algorithme **basique** suivant (voir amélioration dans bin2) :

```
fonction bin(n, k?: entiers)
   Si     k=0 OU n=k
   Alors  renvoyer 1
   Sinon  renvoyer bin(n-1, k-1) + bin(n-1, k)        // Découpage (en n-1) puis assemblage (addition)
   Finsi
Fin Bin;
```

Amélioration : on utilisera une matrice B pour stocker les calculs intermédiaires :

Le contenu de cette matrice pour le calcul de **bin(4, 2)** :

bin(4, 2)	*B* [0] [0] = 1
Calcul de la	*B* [1] [0] = 1
ligne 1	*B* [1] [1] = 1
Calcul de la	*B* [2] [0] = 1
ligne 2	*B* [2] [1] = *B* [1] [0] + *B* [1] [1] = 1+1 = 2
	B [2] [2] = 1
Calcul de la	*B* [3] [0] = 1
ligne 3	*B* [3] [1] = *B* [2] [0] + *B* [2] [1] = 1+2 = 3
	B [3] [2] = *B* [2] [1] + *B* [2] [2] = 2+1 = 3
Calcul de la	*B* [4] [0] = 1
ligne 4	*B* [4] [1] = *B* [3] [0] + *B* [3] [1] = 1+3 = 4
	B [4] [2] = *B* [3] [1] + *B* [3] [2] = 3+3 = 6



L'algorithme du calcul des valeurs de la matrice B devient (en C/C++) : ../..

```
#define minimum(a, b) (a < b ? a: b)
int  bin2 (int n, int k)  // version PrD
{  int  i, j;
  int B[n] [k];
  for (i = 0; i <= n; i++)                                  // Découpage ...
     for (j = 0; j <= minimum (i, k); j++)
        if      (j = = 0 || j = = i)      B[i][ j] = 1;
        else    B[i][j] = B[i - 1][j - 1] + B[i - 1] [j];   // puis assemblage (addition) des solutions partielles
  return B[n][k];
}
```

Complexité de bin2 : pour une valeur de i, on note le nombre de passages dans la *boucle j*

Valeur de i	0	1	2	3	4	5	...	k-1	k	k+1	...	n
Nbr. de passages dans la *boucle j*	1	2	3	4	5	6		k	k+1	k+1	k+1	k+1

Le total : 1 + 2 + ... + k jusqu'à i=**k-1** + (n-k+1) fois (k+1)

= (2n-k+2)(k+1)/2 = **Θ(nk)**.

Bin2 améliore grandement la version basique qui utilise la factorielle classique.

N.B. : En s'appuyant sur la version PrD de la fact., on obtiendra une complexité **Ω(nk)**.

Exercice : faire de même en exploitant l'égalité $C_k^n = C_{n-k}^n$

IX.3- Caractéristiques de la PrD

- **Définition des sous-problèmes** et condition de **sous-problème optimal :**

On peut caractériser une solution optimale à un sous-problème en termes de solutions à ses sous-problèmes (donc sous-sous-problèmes).

Par exemple, si on doit multiplier une chaîne de matrices A_0 .... A_{n-1}

(1) Les sous-problèmes seront les différents parenthésages optimaux de la séquence

$A_i \times A_{i+1} \times \ldots \times A_j$

→ On découpe donc en sous-problèmes :

pour $A_i \times A_{i+1} \times \ldots \times A_j$, on doit trouver $(A_i \times \ldots \times A_k)(Ak_{+1} \times \ldots \times A_j)$ avec $k \in \{i, i+1, \ldots, j-1\}$

- **Caractérisation des solutions optimales** et condition de **sous-structure optimale :**

(2) Quel que soit le choix de k, (A_i x … x A_k) et (Ak_{+1} x … x A_j) doivent être résolus de façon optimale pour avoir un optimum global.

> ☞ Si cela ne devait pas être le cas, une solution globale optimale aurait une solution non optimale à un de ses sous-problèmes !
> Mais cela est impossible car on devrait pouvoir remplacer la solution non optimale par une optimale !

→ Cette observation détermine un problème d'optimisation :

la PrD est un outil d'optimisation.

- Supposons placer ***k*** là où on aura un nombre minimal de multiplications et obtenir N_{ij} une solution.

→ N_{ij} sera exprimé en termes de solutions optimales aux sous-problèmes.

- **Conception de la PrD** :

Sachant que chaque A_i est une matrice $d_i \times d_{i+1}$, de ci-dessus, on déduit :

$N_{ij} = \min_{i \le k < j} \{N_{ik} + N_{k+1\ j} + d_i\ d_{k+1}\ d_{j+1}\}$ pour *i : 0..n-1* avec $N_{ii} = 0$ (une seule matrice)

→ Nij = minimum du nombre de multiplications nécessaires pour chaque sous-expression + le nombre de multiplications pour faire la dernière multiplication de matrices.

Ex. : $A_i \times A_{i+1} \times \ldots \times A_j$

$(A_i \times \ldots \times A_k)$ $(Ak_{+1} \times \ldots \times A_j)$ avec $k \in \{i, i+1, \ldots, j-1\}$

|←trouver min→| |←trouver min→|

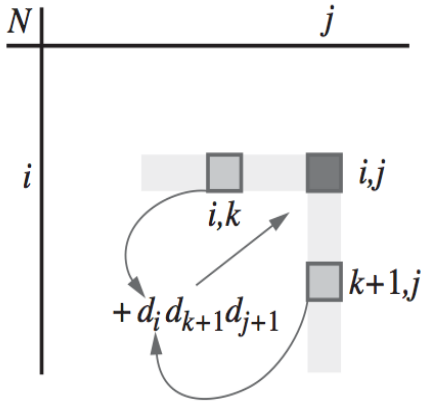
\+ |← la multiplication de ces deux matrices →|

L'expression N_{ij} ressemble à celle d'une stratégie *Diviser-Régner* mais seulement en apparence.

→ Ici, les sous-problèmes **ne sont pas indépendants** et **partagent des sous-sous-problèmes** qui nous empêchent de couper le problème initial en sous-problèmes indépendants.

On peut calculer les Nij de manière *Ascendante* (*Buttom-up*) et les stocker dans une table (cf. PrD) :

1. On initialise N_{ij} = 0 pour i = 0..n-1 puis
2. L'équation générale de N_{ij} calcule $N_{i\ i+1}$ qui <u>ne dépend que</u> de N_{ii} et de $N_{i+1\ i+1}$ qui seront à ce moment-là disponibles.
3. Avec $N_{i\ i+1}$, on aura $N_{i\ i+2}$ ...



- N_{ij} est donc calculé de manière ascendante et $N_{0\ n-1}$ sera la réponse finale.

- Ci-dessous l'algorithme dont la complexité est évaluée à $O(n^3)$.

Fonction ChaineMatrices(d0 , . . . , dn) :
Entrée : une séquence d0 . . . dn d'entiers
Sortie : pour $i, j = 0 \ldots n-1$, le nombre minimum de multiplications N_{ij} nécessaires pour calculer le produit $A_i \cdot A_{i+1} \cdots A_j$ où A_k est une matrice $d_k \times d_{k+1}$
Début
  Pour $i \leftarrow 0$ à $n-1$:
    $N_{ij} \leftarrow 0$

```
    Pour b ← 1 à n − 1 :
        Pour i ← 0 à n − b − 1 :
            j ← i + b
            N_ij ← ∞
            Pour k ← i à j − 1 :
                N_ij ← min(N_ij , N_ik + N_(k+1)j + d_i d_(k+1) d_(j+1))
Fin ChaineMatrices
```

Résumons les étapes de définition d'un schéma PrD :

(I) **Sous-problèmes simples** : le problème initial doit pouvoir être décomposé en sous-problèmes avec une *structure similaire* au problème initial (même formule d'optimalité). De plus il doit y avoir une façon *simple* de définir les sous-problèmes (avec peu d'indices et de variables).

(II) **Optimalité des sous-problèmes** : une solution optimale au problème initial doit être une composition des solutions optimales aux sous-problèmes avec une *composition simple*.

→ On ne devrait pas pouvoir trouver une solution optimale globale contenant une solution *non-optimale* aux sous-problèmes.

(III) **Recouvrement des sous-problèmes** : les solutions optimales aux sous-problèmes in dépendants peuvent contenir des *sous-problèmes communs* (donc des *sous-sous-pro blèmes* communs).

Ces recouvrements permettent d'améliorer l'efficacité de l'algorithme PrD en mémorisant les solutions à ces sous-sous-problèmes (pour ne pas refaire des calculs).

Un bon exemple est Fib(N) utilisant Fib(N-1) et Fib(N-2) déjà calculées et stockées.

On pourra utiliser des tableaux / matrices / etc.

IX.3.1- Exemple-2 : Distance de Levenshtein

La *distance de Levenshein* est une métrique permettant de calculer une *distance d'édition* entre deux mots en vue d'une (proposition de) correction orthographique.

Cette distance représente le nombre **minimum** d'opérations d'édition (*suppression*, *remplacement* et *insertion* de caractères) nécessaires pour rendre identiques les deux chaînes de caractères.

Exemples de la distance (d) :

- pour 2 mots identiques ("Ecole" et « Ecole »), d= 0
- pour "Ecole" et "Ekole", d=1 (remplacement de 'k' par 'c').

→ N.B. : une suppression de 'k' puis une insertion de 'c' donnera d=2 qui n'est pas minimale.

- pour la correction de "klavié" vers "clavier", on a d=3 :

  un remplacement de 'k' par 'c', un remplacement de 'é' par 'e' et l'ajout de 'r'.

- Les éditeurs de texte proposent de remplacer un mot erroné par un autre, ils proposent en général des mots pris dans un dictionnaire et dans l'ordre de cette distance.
  - La **complexité** de ce calcul est le produit des longueurs des 2 mots comparés.

Variantes :

Dans une variante (Daereau-Levenshtein), on peut également procéder à intervertir (transposer) deux caractères adjacents.

- La distance de *Hamming* est une variante qui ne permet que la substitution.
- La distance de Levenshtein a donné lieu à une variante appliquée en séquencement d'ADN (*Smith-Waterman*).

Le principe de PrD appliqué au calcul de la distance de Levenshtein de 2 mots est de construire une matrice contenant la distance entre tous les préfixes des deux mots.

→ Ainsi, la dernière valeur calculée donnera la distance entre les deux mots.

IX.3.2- Le Pseudo-Algorithme de Levenshtien

```
Fonction levenshtein(mot1, mot2 : tableau de caractères)   // chaque mots commence à l'indice 1
   locale : M?: matrice[0..taille mot1][0..taille mot2]    // la matrice M commence à l'indice 0
Début
   Pour i=0.. taille mot1        M[i][0]=i; Fin pour       // distance des préfixes du  mot1 aux mots vides
   Pour j=0.. taille mot2     M[0][j]=j; Fin pour          // distance des préfixes du  mot2 aux mots vides
   Pour j=1.. taille mot2                                  // le 1er caractère de chaque mot est à l'indice 1
      Pour i=1.. taille mot1
         Si (mot1[i]=mot2[j])
         Alors M[i][j]=M[i-1,j-1]                        // aucune opération nécessaire
         Sinon m[i][j]=min( M[i-1][j]+1,                 // suppression
                            M[i][j-1]+1,                 // insertion
                            M[i-1][j-1]+1)               // substitution
         Fin si
      Fin pour
   Fin pour
   m[taille mot1][taille mot2] est le résultat
Fin Levenshtien
```

L'intérêt de la stratégie **PrD** est de calculer les distances successives à l'aide de précédentes (en phase ascendante).

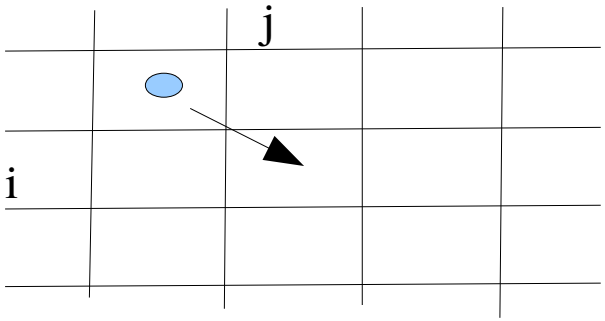
La phase descendante est ici réduite aux initialisations.

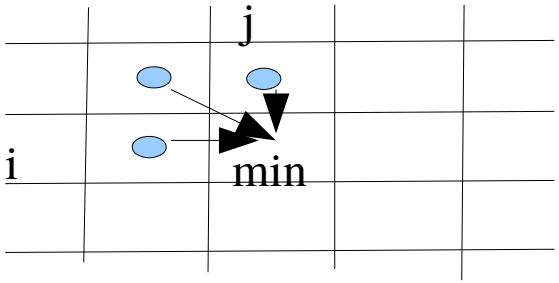
Exemple : les mots "klavié" vs "clavier"

		k	l	a	v	i	é
	0	1	2	3	4	5	6
c	1	1	2	3	4	5	6
l	2	2	1	2	3	4	5
a	3	3	2	1	2	3	4
v	4	4	3	2	1	2	3
i	5	5	4	3	2	1	2
e	6	6	5	4	3	2	2
r	7	7	6	5	4	3	**3**

← il faudra 3 opérations pour rendre "klavié" identique à "clavier"

Pour comprendre le remplissage de la matrice M, considérons les deux cas de figure de comparaison de 2 lettres (identiques ou différentes) :





mot2[j] = mot1[i] : mat[i][j]= mat[i-1][j-1]+1

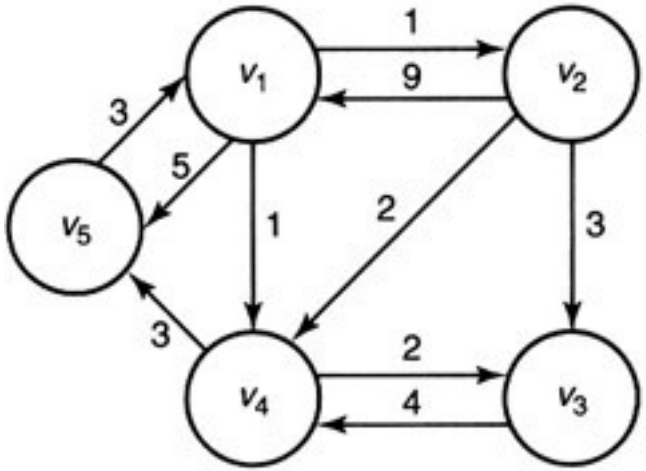
mot2[j] ≠ mot1[i] : mat[i][j]= min(des 3 cases)+1

IX.3.3- Exemple 3 : algorithme de Floyd

Propos : calcul des chemins entre toutes paires de nœuds d'un graphe.

Utilisation : trafic aérien (en l'absence de ligne directe), routage (adresses) dans les réseaux et routage de messages dans un réseau global, ...

Exemple : dans ce graphe (valué orienté), on cherche le meilleur chemin entre v1 et v3.



On a les **candidats** avec leurs **valeurs**:

longueur(<v1, v2, v3>) = 1+3=4

longueur(<v1, v4, v3>) = 1+2=3

longueur(<v1, v2, v4, v3>) = 1+2+2=5

Le chemin <v1, v4, v3> semble le plus court.

→ Un problème d'optimisation

../..

La recherche du chemin le plus court est un problème **d'optimisation** :

Il existe plusieurs chemins (**candidats**) entre deux nœuds (de **valeurs** différentes) et on souhaite calculer le plus court (de **valeur optimale**)

N.B. : il peut exister plusieurs chemins de longueur minimale entre 2 nœuds.

→ Le choix est dans ce cas aléatoire.

Un algorithme **basique** peut calculer TOUS les chemins entre TOUS couples de nœuds puis de choisir le plus court.

Un tel algorithme (pour un graphe fortement connexe) est de **complexité exponentielle** *(N-2)!* pour N nœuds.

A l'aide de la PrD, on peut obtenir une complexité cubique.

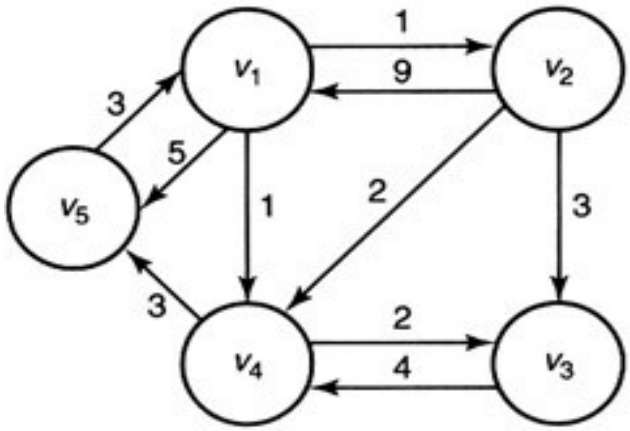
Démarche : on définit une matrice W des poids (distances directes) avec la convention :

W[i][j] = 0 *si i=j*

W[i][j] = ∞ si i et j non directement connectés

W[i][j] = la distance (arc) entre i et j

Pour le graphe précédent, on aura la matrice initiale des distances W et D = la matrice des chemins les plus courts.



	1	2	3	4	5
1	0	1	∞	1	5
2	9	0	3	2	∞
3	∞	∞	0	4	∞
4	∞	∞	2	0	3
5	3	∞	∞	∞	0

W

	1	2	3	4	5
1	0	1	3	1	4
2	8	0	3	2	5
3	10	11	0	4	7
4	6	7	2	0	3
5	3	4	6	4	0

D

On notera $D^{(k)}[i][j]$ = la longueur du chemin le *plus court* entre **i** et **j** utilisant seulement les nœuds {v1, v2, ..., vk}.

Par définition, on a (pour *N* nœuds) :

- **$D^{(0)}[i][j]$** = chemin passant par aucun autre nœud que j et j = la matrice W
- **$D^{(n)}[i][j]$** = chemin passant par n'importe quel autre nœud du graphe

  = la matrice finale D (meilleurs chemins)

Exemples :

$D^{(0)}[2][5]$ = longueur(<v2,v5>) = ∞ chemin le plus court entre v_2-v_5 en passant par v_0 = direct (v_0 fictif)

$D^{(1)}[2][5]$ = minimum(longueur(<v2,v1, v5>) , longueur(<v2,v5>)) = minimum(14, ∞) = 14

$D^{(2)}[2][5]$ = **$D^{(1)}[2][5]$** = 14 aucun chemin partant de v2 ne peut repasser par v2 (vrai pour tout graphe)

$D^{(3)}[2][5]$ = **$D^{(2)}[2][5]$** = 14 l'inclusion de v3 (seul) n'apporte rien de plus dans ce graphe

→ (<V2,V3,V5>=<V2,V5>, on ne passe pas par V4).

Donc, on obtiendra la matrice **D** depuis **W** en procédant (principe de la PrD):

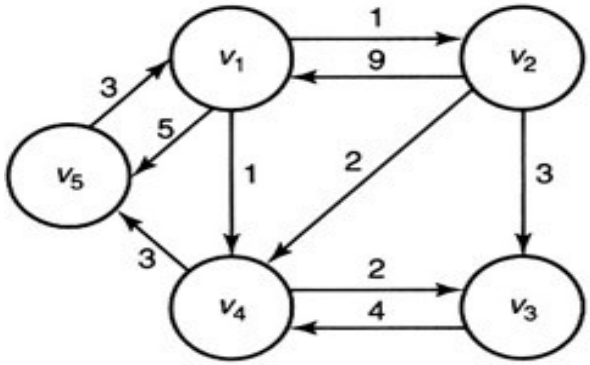
1. Définir une propriété récursive (calcul) permettant d'obtenir $D^{(k)}$ depuis $D^{(k-1)}$.
2. Résoudre une instance du problème (ascendant) en répétant l'étape (1) pour k=1 .. n.

   Ce qui crée la séquence : $D^{(0)}$, $D^{(1)}$, $D^{(2)}$, ... , $D^{(n)}$.

Deux cas peuvent se présenter pour l'étape 1:

Cas 1 : au moins un des chemins les plus courts de **i** à **j** n'utilisera pas v_k dans {v1, ..., vk} :

$D^{(k)}[i][j] = D^{(k-1)}[i][j]$. (1)



Par exemple, dans le graphe précédent :

$D^{(5)}[1][3] = D^{(4)}[1][3] = 3$

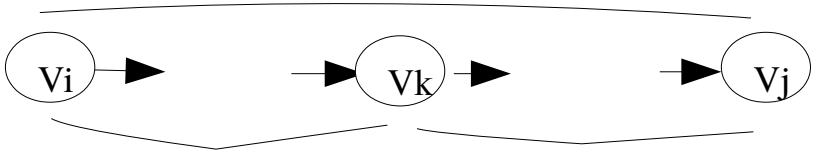
Car l'inclusion de v5 à {v1, v4, v3} n'apporte rien au chemin le plus court qui reste <v1,v4,v3>.

Cas 2 : les chemins les plus courts allant de vi à vj utilisant seulement {v1, v2, ..., vk} utilisent vk.

Dans ce cas, tout chemin le plus court est décrit dans le schéma ci-contre.

Sachant que v_k ne peut pas être un nœud intermédiaire du sous chemin vi à vk, ce sous chemin utilise seulement les nœuds $\{v_1, v_2, ..., v_{k-1}\}$.

*Le chemin **le plus court** de vi à vj utilisant seulement {v1, v2, ..., vk}*



*Le chemin le plus court de **vi** à **vk** utilisant seulement {v1, v2, ..., vk}*

*Le chemin le plus court de **vk** à **vj** utilisant seulement {v1, v2, ..., vk}*

Ce qui implique que le sous-chemin sera de la même longueur que $\mathbf{D^{(k-1)}[i][k]}$.

D'où : $D^{(k)}[i][j] = D^{(k-1)}[i][k] + D^{(k-1)}[k][j]$. (2)

Exemple : $D^{(2)}[5][3] = D^{(1)}[5][2] + D^{(1)}[2][3] = 4 + 3 = 7$.

Sachant que nous serons dans l'un des 2 cas ci-dessus, la valeur de $\mathbf{D^{(k)}[i][j]}$ est le minimum de la valeur à droite des équations (1) et (2).

Ce qui veut dire que l'on détermine $\mathbf{D^{(k)}[i][j]}$ depuis $\mathbf{D^{(k-1)}[i][j]}$ par :

$$D^{(k)}[i][j] = \text{minimum}(D^{(k-1)}[i][k], D^{(k-1)}[i][k] + D^{(k-1)}[k][j]) .$$

Ceci accomplit le cas 1 ci dessus.

Pour l'étape 2, on développera la propriété récursive de l'étape 1 pour créer la séquence $\mathbf{D^{(0)}}$, $\mathbf{D^{(1)}}$, $\mathbf{D^{(2)}}$, ... , $\mathbf{D^{(n)}}$.

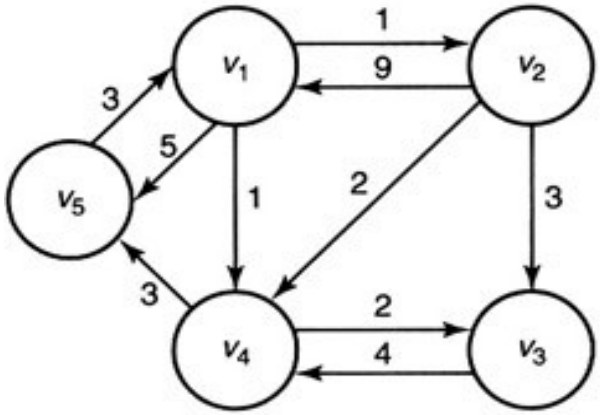
Exemple de $\mathbf{D^{(2)}[5][4]}$ =min($\mathbf{D^{(1)}[5][4]}$, $\mathbf{D^{(1)}[5][2]}$ + $\mathbf{D^{(1)}[2][4]}$) (rappel : D^0 = W la matrice initiale) :

$\mathbf{D^{(1)}[2][4]}$ = min($\mathbf{D^{(0)}[2][4]}$, $\mathbf{D^{(0)}[2][1]}$ + $\mathbf{D^{(0)}[1][4]}$) = min(2, 9+1) = 2

$\mathbf{D^{(1)}[5][2]}$ = min($\mathbf{D^{(0)}[5][2]}$, $\mathbf{D^{(0)}[5][1]}$ + $\mathbf{D^{(0)}[1][2]}$) = min(∞, 3+1) = 4

$\mathbf{D^{(1)}[5][4]}$ = min($\mathbf{D^{(0)}[5][4]}$, $\mathbf{D^{(0)}[5][1]}$ + $\mathbf{D^{(0)}[1][4]}$) = min(∞, 3+1) = 4

$\mathbf{D^{(2)}[5][4]}$ = min($\mathbf{D^{(1)}[5][4]}$, $\mathbf{D^{(1)}[5][2]}$ + $\mathbf{D^{(1)}[2][4]}$) = min(4, 3+2) = 4



Après avoir calculé tous les D^2, on continuera jusqu'à D^5.

Ces calculs donneront la matrice D.

	1	2	3	4	5
1	0	0	4	0	4
2	5	0	0	0	4
3	5	5	0	0	4
4	5	5	0	0	0
5	0	1	4	1	0

IX.3.4- Algorithme de Floyd

```
void floyd (int n, int W[][], int D[][])
{  int i, j, k;
   D = W;
   for (k = 1; k<=n; k++)
      for (i = 1; i <= n; i++)
         for (j = 1; j <= n; j++)
            D[i][j] = minimum(D[i][j], D[i][k] + D[k][j]);        // même principe utilisé dans Dijkstra (voir plus loin)
}
```

N.B. : on peut remarquer la trame de l'algorithme de Dijkstra.

La complexité de cet algorithme : $\Theta(n^3)$

La matrice **finale** (D) des chemins les plus courts :

	1	2	3	4	5
1	0	0	4	0	4
2	5	0	0	0	4
3	5	5	0	0	4
4	5	5	0	0	0
5	0	1	4	1	0

IX.3.5- Obtention des trajets (coming from)

On enregistre dans la nouvelle matrice CF[u][v] le plus grand numéro du nœud intermédiaire sur le chemin le plus court u --> v. CF[u][v]=0 si ce nœud n'existe pas.

```
void floyd2 (int n, int W[][], int D[][], int CF[][])
  { int, i, j, k;
    for(i = 1; i <= n; i++)
       for (j = 1; j <= n; j++) CF[i][j = 0;
    D = W
    for (k = 1; k <= n; k++)
       for(i = 1; i <= n; i++)
          for(j = 1; j <= n; j++)
             if (D[i][k] + D[k][j] < D[i][j])
                { CF[i][j] = k;  D[i][j] = D[i][k] + D[k][j];
                }
}
```

Affichage du chemin :

```
void affiche_path (int u, v)      // trajet pour aller de u à v
  { if (CF[ u ] [ v ] ! = 0)
     { affiche_path (u, CF[u] [v]);
       cout < < " --> " < < CF[ u ] [ v ];
      affiche_path (CF[ u ] [ v ], v); } }
```

IX.4- Programmation Dynamique et la question d'optimisation

L'algorithme de **Floyd** permet de déterminer le chemin le plus court et de le calculer.

Comme étudié ci-dessus, la construction de la solution optimale devient donc la 3e étape du développement d'un problème d'optimisation.

Les étapes de PrD appliquée à un problème d'optimisation devient :

1. Définir une propriété récursive qui donne la solution à une instance du problème
2. Calculer la valeur de la solution optimale de chaque instance
3. Construire la solution optimale de façon ascendante en utilisant les instances plus petites.

Comme toute autre méthode, la PrD peut ne pas être adaptée à un problème d'optimisation si le principe suivant n'est pas respecté :

Le principe d'optimalité : *une solution optimale à un problème contient des solutions optimales à toutes les sous instances de ce problème (et vice versa).*

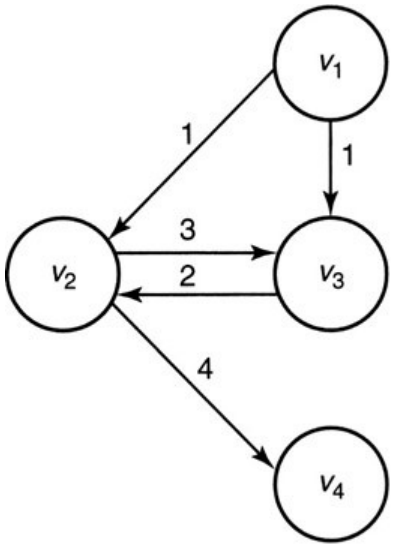
Appliquons ce principe au problème des chemins les plus courts :

le chemin le plus court v_i --> v_j en passant par le nœud intermédiaire v_k est optimale

si les chemins partiels v_i --> v_k et v_k --> v_j sont optimaux (les plus courts ici).

Il faudra donc d'abord démontrer l'application du principe d'optimalité avant de faire l'hypothèse d'optimalité de la solution avec PrD. ../..

Contre exemple : calculer les chemins **les plus <u>longs</u>** sur le graphe ci-contre.



N.B.: on se restreint à ne pas exploiter le circuit (V2,V3) dans ce graphe.

Sinon, on peut passer une infinité de fois dans ce cycle.

On a le chemin le plus long optimal entre v1 --> v4 : <v1, v3, v2, v4> = 7

Cependant, le sous-chemin v1 -> v3 n'est pas optimal car :

longueur(<v1, v3>) =1 et longueur(<v1, v2, v3>) =4 (sans circuit)

Or, la solution optimale finale <v1, v3, v2, v4> ne contient pas <v1, v2, v3>.

=> <u>Le principe d'optimalité ne s'applique pas ici.</u>

Un exemple d'application du principe est la recherche optimale dans les ABOHs (AVLs).

La solution optimale passera parfois par une optimisation des données.

IX.4.1- Une variante de Floyd : algorithme de base de Roy-Warshall

Une variante de l'algorithme de Floyd :

Calcule les plus courts chemins comportant au plus k arcs.

On calcule les deux matrices W et D de la même manière :

Etape 0 : Matrice W (0 arcs --> δ0) : pour tous les couples x, y on pose

δ0(x, y) = valeur(a) s'il existe un arc **a** entre x et y, ∞ sinon.

Etape k : Matrice D (meilleures distances de longueur k arcs)

- Pour k = 1, 2, 3, . . . , n faire
- Pour tout couple de nœuds x, y
- Pour tout sommet z successeur de x
- $\delta_{k+1}(x, y) = \min(\delta_k(x, y), \delta_0(x, z) + \delta_k(z, y))$

La preuve de justesse est apportée par une récurrence sur k.

N.B. : pour obtenir effectivement des trajets, une variante à *Coming_From* est d'utiliser une matrice *suivant[x][y]* qui contiendra le sommet qui suit x dans le trajet qui mène de x à y.

Ce qui donne l'algorithme Roy Warshall de complexité $O(n^3)$:

Soient les structures de données : les matrices W, D et CF avec

• D[x][y] : la longueur du plus court chemin de x à y de longueur k (avec mise à jour) = la matrice D.

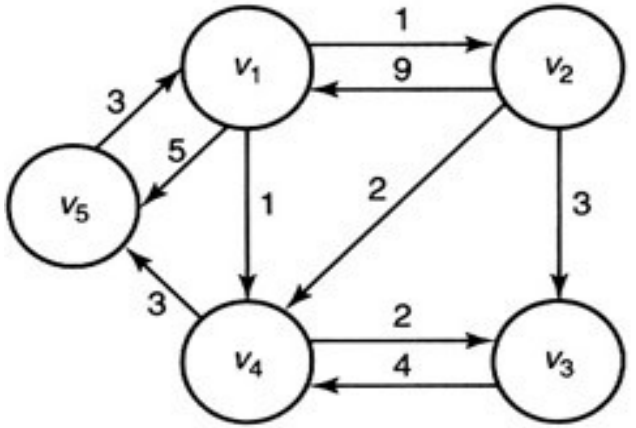
• CF[x][y] sommet qui précède x sur le plus court chemin de x à y (non traité ci-dessous).

L'algorithme (chemins les plus courts de longueur au plus k entre tout couple de nœuds):

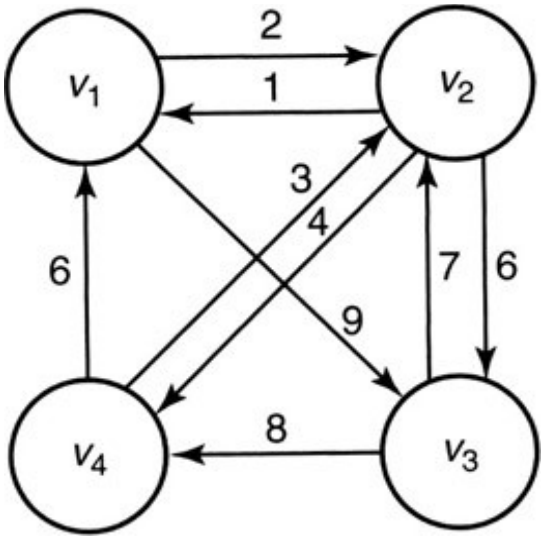
```
Pour k' de 1 à k faire
    Pour i de 1 à n faire
        Pour j de 1 à n faire
            Si    D[i][j] > D[i][k'] + D[k'][j]
            Alors   D[i][j] = D[i][k'] + D[k'][j]
```

Exemple (k=2) : la matrice des distances :

	V1	V2	V5	V4	V3
V1	999	1	5	1	4
V2	999	999	999	2	3
V5	3	4	8	4	7
V4	999	999	3	999	2
V3	999	999	999	4	999



IX.4.2- PrD et le problème du Voyageur de commerce (TSP)



Problème : trouver, dans un graphe de villes et planifier un voyage de **coût** minimal où le voyageur visite chaque nœud (ville) une seule fois et **revient** à sa base (son point de départ).

Dans l'exemple, on a :

longueur(<v1, v2, v3, v4, v1>) = 22

longueur(<v1, v3, v2, v4, v1>) = 26

longueur(<v1, v3, v4, v2, v1>) = ***21***

Comme pour le calcul des chemins les plus courts, la **complexité est exponentielle**.

PrD peut-elle être appliquée ?

On vérifie son principe :

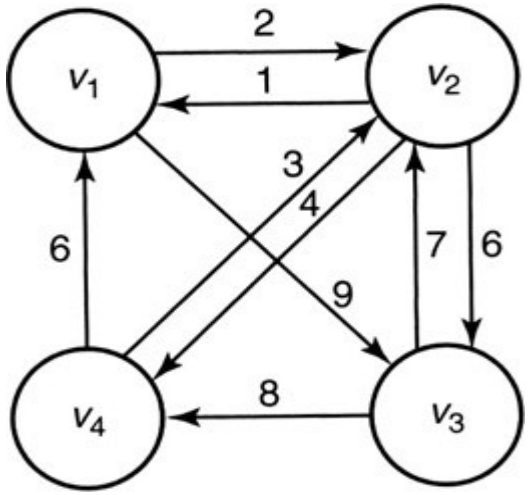
On note que si v_k est le premier nœud avant v1 sur la tourné optimale, alors le sous chemin de v_k à v_1 doit être le chemin le plus court de v_k --> v_1 qui passe une fois par tous les autres nœuds.

→ Le principe d'optimalité s'applique et la PrD peut être utilisée.

Pour ce faire, on représente la graphe par sa matrice d'adjacence **W** (cf. Floyd) :

On peut poser ce problème :

Calculer la matrice D des distances telle que **D[vi][A]** = la longueur du chemin le plus court allant de vi à **v1** en passant une seule fois par tous les nœuds de l'ensemble A (V1 = base)



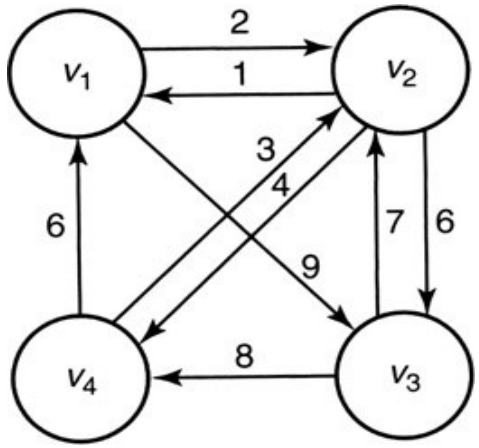
Pour le graphe, on a V={v1, v2, v3, v4}

- Si A={v3}, alors **D[v2][A]** = longueur(<v2, v3, v1>) = 15
- Si A={v3, v4}, alors **D[v2][A]** = min(longueur(<v2,v3,v4,v1>), longueur(<v2, v4, v3, v1>))

= min(20, ∞) = 20

Sachant que V-{v1, vj} contenant tous les nœuds (sauf ces 2 là) et le principe d'optimalité :

longueur de la meilleure tournée = min(W[1][j] + D[vj][V-{v1, vj}]) 2 <= j <= n



Et en général, pour i ≠1 et $v_i \notin A$:

- **D[vi][A] = min(W[i][j] + D[vj][V-{vj}])** si $A \neq \emptyset$, j : $v_j \in A$,

(∅ = ensemble vide)

- **D[vi][∅] = W[i][1]**

On utilisera cette dernière égalité (**D[vi][∅] = W[i][1]**) pour construire l'algorithme de la PrD de ce problème.

Notons alors que :

D[v2][∅] = W[2][1]= **1**

D[v3][∅] = ∞

D[v4][∅] = **6**

	1	2	3	4
1	0	2	9	∞
2	1	0	6	4
3	∞	7	0	8
4	6	3	∞	0

Ensuite, on considère tous les ensembles contenant un élément :

- **D[v3][{v2}] = min(W[3][j] + D[vj][{v2} - {vj}]) j : vj ∈ {v2}**

  = W[3][2] + D[v2][∅] = 7+1=8

De manière similaire :

D[v4][{v2}] = 3+1 = 4

D[v2][{v3}] = 6+∞ = ∞

D[v4][{v3}] = ∞ + ∞ + ∞= ∞

D[v2][{v4}] = 4+6 = 10

D[v3][{v4}] = 8+6 = 14

	1	2	3	4
1	0	2	9	∞
2	1	0	6	4
3	∞	7	0	8
4	6	3	∞	0

Considérons ensuite les ensembles à 2 éléments :

D[v4][{v2, v3}] = min(W[4][j] + D[vj][{v2, v3} - {vj}]) j : vj ∈ {v2, v3}

= min(W[4][2] + D[v2][{v3}] , W[4][3] + D[v3][{v2}])

= min(3 + ∞, ∞+8) = ∞

et de manière similaire :

D[v3][{v2, v4}] = min(7 + 10 , 8 + 4) = 12

D[v2][{v3, v4}] = min(6 + 14 , 4 + ∞) = 20

	1	2	3	4
1	0	2	9	∞
2	1	0	6	4
3	∞	7	0	8
4	6	3	∞	0

Et la longueur de la tournée optimale :

D[v1][{v2, v3, v4}] = min(W[1][j] + D[vj][{v2, v3, v4} - {vj}]) j : vj ∈ {v2, v3, v4}

= min(W[1][2] + D[v2][{v3, v4}] ,
W[1][3] + D[v3][{v2, v4}]),
W[1][4] + D[v4][{v2, v3}])

= min(2 + 20 , 9 + 12, ∞+ ∞) = 21

IX.4.3- PrD : Algorithme TSP

void *travel* (**int** n, **int** W [] [], **int** CF [] [], **int** & *minlength*)
{ **int** i, j, k; **int** D [1 .. n] [sous ensemble de $V - \{v_1\}$];
for ($i = 2; i <= n; i++$) D [i] [$\varnothing$] = W[i] [1];
for ($k = 1; k <= n - 2; k++$)
for (tout sous ensemble $A \subseteq V - \{v_1\}$ contenant k nœuds)
for (i tel que $i \neq 1$ & $v_i \notin A$)
{ D [i] [A] = *minimum* (W [i] [j] + D [j [$A - \{v_j\}$]); j: $v_j \in A$
CF[i] [A] = valeur de j qui a donné le minimum;
}
D [1] [$V - \{v_1\}$] = *minimum* (W[1] [j] + D[j] [$V - \{v_1, v_j\}$]); $2 <= j <= n$
CF[1] [$V - \{v_1\}$] = valeur de j qui a donné le minimum;
minlength = D[1] [$V - \{v_1\}$];
}

La matrice CF (Coming-From) donnera la tournée optimale.

La complexité de cet algorithme est $\Theta(n^2 \cdot 2^n)$

A titre d'indication : pour n=20 (nombre de nœuds)

- le calcul avec un algorithme brut-force et une microseconde par opération de base

  19! microsecondes = 3857 années

- le calcul avec cet algorithme : $20^2 \cdot 2^{20}$ = 21 millions d'accès aux matrices.

- avec n=60, même l'algorithme ci-dessus prendra **plusieurs années** !

Le calcul du trajet (pour le graphe) :

3	4	2

Le tableau CF contiendra :

CF[1], {v2, v3, v4}] CF[3], {v2, v4}] CF[4], {v2}]

et la tournée optimale est calculée par :

- indexe du premier nœud : CF[1][{v2, v3, v4}] =3
- on utilise 3 : CF[3][{v2, v4}] = 4
- on utilise 4 : CF[3][{v2}] = 2

Et la **tournée optimale : {v1, v3, v4, v2, v1}**

IX.4.4- PrD et le problème de Sac à Dos

- Des objets *1, 2, . . . , n* de poids *w1 , w2 , . . . , wn*
- Chacun rapporte un bénéfice *b1 , b2 , . . . , bn*
- Trouver le bénéfice maximum réalisable sachant que la charge maximale est P_0

Idée pour P_0 entier pas trop grand :

- On note B(k, p) le bénéfice maximal réalisable avec des objets *1, 2, . . . , k* et le poids maximal p
- On a pour k = 1:

$B(1, p) = 0$ si $p < w_1$ // si le poids max est inférieur au poids de l'objet (on ne prend pas !)

$= b_1$ si $p \geq w_1$

- Pour k > 1

$B(k, p) = B(k-1, p)$ si $p < w_k$ // idem mais pour le $k^{ème}$ objet

$= \max[B(k-1, p), B(k-1, p-w_k) + b_k]$ si $p \geq w_k$

N.B. : à comparer avec la version BT et BT optimisée du problème de Sac à dos.

Exemple (poids total $P_0 = 12$) : matrice des profits (on fait varier *k:1..8* et P_0 *: 0..12*) **:**

Objets	1	2	3	4	5	6	7	8
Poids	2	3	5	2	4	6	3	1
Bénéfices	5	8	14	6	13	17	10	4

$B(k,p)$	0	1	2	3	4	5	6	7	8	9	10	11	12
$k=1$	0	0	5	5	5	5	5	5	5	5	5	5	5

$B(k,p)$	0	1	2	3	4	5	6	7	8	9	10	11	12
$k=1$	0	0	5	5	5	5	5	5	5	5	5	5	5
$k=2$	0	0	5	8	8	13	13	13	13	13	13	13	13

$B(k,p)$	0	1	2	3	4	5	6	7	8	9	10	11	12
$k=1$	0	0	5	5	5	5	5	5	5	5	5	5	5
$k=2$	0	0	5	8	8	13	13	13	13	13	13	13	13
$k=3$	0	0	5	8	8	14	14	19	22	22	27	27	27

Suite ... (il y a 8 objets, k=1..8)

$B(k,p)$	0	1	2	3	4	5	6	7	8	9	10	11	12
$k=1$	0	0	5	5	5	5	5	5	5	5	5	5	5
$k=2$	0	0	5	8	8	13	13	13	13	13	13	13	13
$k=3$	0	0	5	8	8	14	14	19	22	22	27	27	27
$k=4$	0	0	6	8	11	14	14	20	22	25	28	28	33
$k=5$	0	0	6	8	13	14	19	21	24	27	28	33	35
$k=6$	0	0	6	8	13	14	19	21	24	27	30	33	36
$k=7$	0	0	6	10	13	16	19	23	24	29	31	34	37
$k=8$	0	4	6	10	14	17	20	23	27	29	33	35	38

IX.5- Stratégies Greedy

- Dans cette stratégie, une solution est trouvée sous forme d'une séquence d'étapes où le successeur d'une étape est obtenue par le **meilleur choix** optimal local.

  → La solution finale n'est pas forcément globalement optimale.

Il faudra donc procéder à une vérification d'optimalité.

IX.6- Exemples et comparaison avec PrD

Voir la 2e partie de ce chapitre pour les algorithmes cités.

- MST (*Kruskal*, *Prim*)
- Dijkstra (chemin le plus court entre un nœud et les autres)

➥ Ces deux problèmes sont traités plus loin (sections Dijkstra)

Dijkstra PrD : algorithme de Floyd

Dijkstra Greedy : algorithme (classique)

- Sac à dos : PrD : vue ci-dessus vs.

Greedy : si les objets à prendre sont par fraction (*greedy* donnera une meilleure solution).

- Scheduling : voir plus loin.

Comparaison Greedy et Prog Dyn : **dépend du problème.**

IX.7- Un exemple d'algorithme greedy : coloration de graphe

On choisit une couleur, on examine les nœuds dans l'ordre et on essaie de donner la couleur choisie à un maximum de nœuds.

N.B. : La méthode est appelée également une méthode **Gloutonne**.

- Soit **G=(V, E)** : Graphe avec |V| = nombre de nœuds

```
Procédure Coloration_g (G: ES Graphe) =          // |V| nœuds dans le graphe
Début
  Pour tout nœud N dans V
        marquer N non colorié
  Fin Pour
  K=la première couleur ;
  Tant que tous les nœuds de G ne sont pas coloriés
     Gloutonne(K, G);
     K= la couleur suivante;
  Fin tq;
Fin Coloration_g;
```

- Aucun ordre n'est pris en compte. Pour choisir les variables (les nœuds), on les considère dans l'ordre arbitraire numérique croissant 1..N

```
Procédure Gloutonne(K?. couleur; G?. ES Graphe )=
Début
  Pour chaque N dans V                    // visiter les éléments de V
     Choisir N non encore colorié         // choix quelconque
     Si aucun des adjacents de N n'est  colorié par la couleur K
     Alors          Colorier N par K
     Fin si;
  Fin Pour;
Fin Gloutonne;
```

Traces d'exécution : Gloutonne (3 couleurs) :

K=rouge G={1,2,3,4,5} ➟ 1 coloré rouge, 2 coloré rouge;

K=vert G={3,4,5} ➟ 3 coloré vert, 4 coloré vert

K= bleu G={5} ➟ 5 coloré bleu

IX.8- Comparaison BT et PrD

IX.8.1- Rappel d'exemples BT

N-reines

Somme des sous ensembles (voleur)

Coloration de graphes

Sac à dos

L'ensemble de ces exemples sont traités dans ce document.

Comparaison Prog. Dyn et BT

IX.9- Branch & Bound

IX.9.1- Exemples

- Sac à dos
- Parcours en largeur avec élagage B&B
- Best First avec élagage B&B
- TSP
- Inférence Abductive

IX.10- Un exemple Best First : coloration heuristique

Une heuristique (dite **admissible** est une heuristique qui) permet de trouver le meilleur nœud.

Dans cette méthode, au lieu de choisir les nœuds dans un ordre arbitraire, on choisira les nœuds selon leur nombre d'adjacents (leur **degré**).

On a donc besoin d'une structure supplémentaire qui contient le degré des nœuds.

```
Procédure Coloration_h (G: ES Graphe) =
Début
  Pour tout nœud N dans V
        marquer N non colorié
  Fin pour;
  K= la première couleur ;
  Tant que tous les nœuds de G ne sont pas coloriés
     Heuristique(K, G);
     K= la couleur suivante;
  Fin tq;
Fin Coloration_h;
Procédure Heuristique(K?: couleur; G?: ES Graphe)=
```

```
Début
  Pour chaque N dans V                    // visiter les éléments de V
     Tant qu'il existe un nœud N  non encore colorié,  de degré  maximum
                                et  qu'aucun de ses adjacents  n'ait reçu la couleur K
        Colorier N par K
        Réduire de 1 le degré de chacun des adjacents de N
  Fin Pour;
Fin Heuristique;
```

Trace (2 couleurs) :

K=rouge G={1,2,3,4,5}

choix=5 ➟ 5 coloré rouge;

choix=2 ➟ 2 coloré rouge;

K=vert G={1,3,4}

choix=1 ➟ la couleur verte est donnée à tous

IX.10.1- Une solution First Fail à la coloration

Cette méthode est à base de propagation de contraintes

- Colorier les nœuds produisant une solution
- Contraintes :
  - deux régions voisines (deux nœuds adjacents) ont deux couleurs différentes.
  - minimiser le nombre de couleurs.

Méthode :

- Attribuer le domaine de chaque nœud = l'ensemble des couleurs.
- Ordonner les nœuds selon leur degré de connexion (nombre de voisins).

L'algorithme de principe suivant donne une solution au problème de coloration :

```
Fonction colorer_C(G: ES Graphe;
                   (V x C)? ES   Ensemble de (nœuds x Valeur) ;
                   Dom? ES       ensemble de domaines des nœuds) =
Début
  Si G est totalement colorié alors retourner (V x C)
  Sinon
     Choisir dans (V x C) un nœud U non encore colorié
     V= V - {U};
     Pour toute couleur Ci du domaine de U
          Dom1= dans Dom, supprimer C du domaine des adjacents de U
          Résultat=colorer_C(Graphe, <V x C> , Dom1);
          Si    (Résultat ≠ Vide)
          Alors retourner Résultat;
          Fin si;
     Fin Pour;
     Retourne Vide;
  Fin si;
Fin colorer;
```

Trace d'exécution :

On représente l'ensemble du problème par le tableau suivant :

Le choix des variables se fera en fonction du nombre de contraintes puis de la taille du domaine.

Var	Adjacents	nb_C°	Domaine	Valeur
1	< 5 >	1	{c1, c2, c3}	?
2	< 3, 4 >	2	{c1, c2, c3}	?
3	< 2, 5 >	2	{c1, c2, c3}	?
4	< 2, 5 >	2	{c1, c2, c3}	?
5	< 1,3,4 >	3	{c1, c2, c3}	?

-La première variable selon ces critères est v5.

- La première valeur de son domaine = c1, ce sera sa valeur.

- On élimine c1 du domaine de chacun des adjacents de v5;

- v5 étant traitée, on la supprime des adjacents de ses voisins avec mise à jour du nombre de contraintes pour chacune.

- La variable suivante sera v2

(dont le nombre de contraintes = 2 et la taille du domaine=3.

- Puis les variables v1, v3 et v4 auront un nombre de contraintes=0 et dont c1 est éliminée du domaine.

- La solution correspond à celle de la méthode heuristique.

IX.10.1.a- Exercices sur la coloration

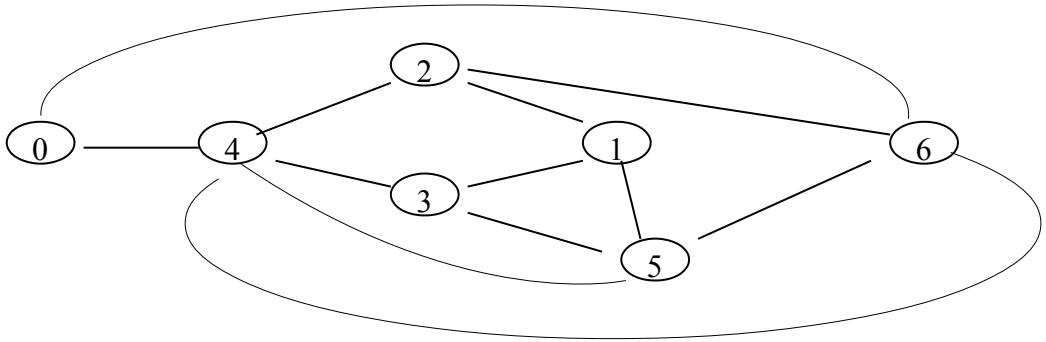
- Appliquer différentes techniques de coloration aux graphes suivants :

Réponses attendues :

Heuristique : 3 couleurs

Gloutonne : 3 ou plus selon l'ordre

- Un autre graphe :



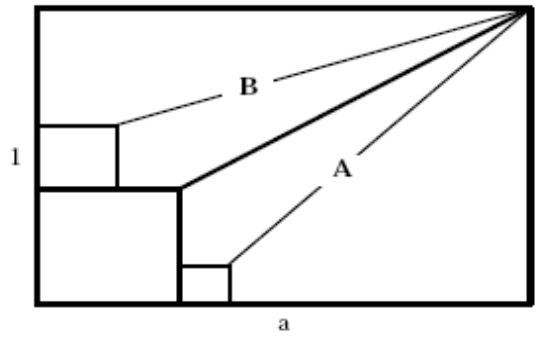
Réponses attendues :

Gloutonne : 4 à 5 couleurs utilisées selon l'ordre des nœuds (3 si l'ordre correpond à l'heuristique)

Heuristique : 3 couleurs

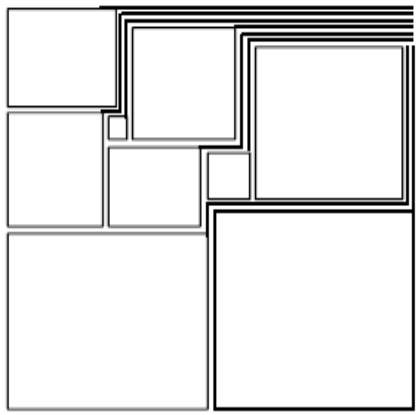
X- Le problème Bin Packing : un cas de Best First

- Soit **a** le rapport de la longueur du plus grand côté avec celle du plus petit côté du rectangle construit.



- On peut évidemment supposer que la longueur du plus petit côté est 1 et que la longueur du plus grand côté est **a**.

But : remplir un rectangle de dimensions **1**×**a** par n carrés tous distincts.



La base de l'algorithme de remplissage consiste à :

(1) placer un carré dans le coin inférieur gauche du rectangle,

(2) remplir de carrés la zone A, si elle n'est pas vide,

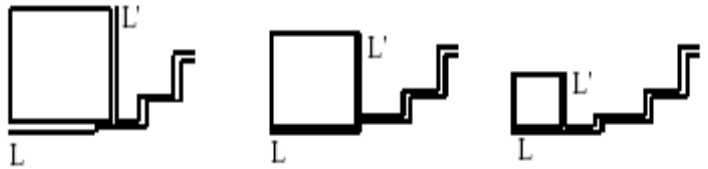
(3) remplir de carrés la zone B, si elle n'est pas vide.

Le remplissage des zones A et B se fera récursivement de la même façon :

→ placer un carré dans le coin inférieur gauche et remplir deux sous-zones.

Indications : Les zones et les sous-zones sont séparées par des lignes brisées en forme d'escalier allant du coin supérieur droit des carrés au coin supérieur droit du rectangle. Ces lignes brisées ne descendent jamais et s'il est possible d'en tracer plusieurs pour aller d'un point à un autre on considère toujours la plus basse.

Exemple : toutes les lignes de séparations correspondant à la première solution du problème lorsque $\mathbf{n = 9}$:

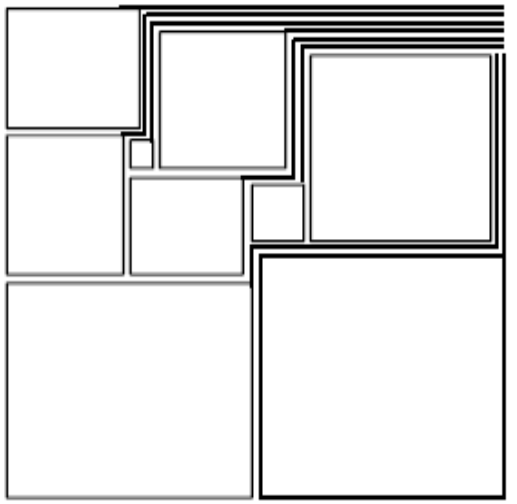


On décide de ne les accepter que s'il est possible de placer un carré de dimension $\mathrm{b} \times \mathrm{b}$ sur le début de la ligne L et que si $\mathrm{L'}$ est la ligne constituée du côté vertical droit de ce carré prolongé par la partie droite de la ligne L (voir figure).

En fait L désigne la ligne inférieure d'une zone mais de laquelle on a enlevé le premier segment vertical.

La figure montre les trois cas qui peuvent se présenter et qui se matérialiseront par trois règles.

- Soit le carré déborde sur la 1e marche, qui en fait était une fausse marche de hauteur nulle,
- soit le carré est collé contre la première marche,
- soit le carré n'est pas assez grand pour toucher la première marche.



Les solutions pour 9 carrés :

On obtient 8 réponses.

Les deux premières sont

a = 33/32 et les carrés :

<15/32, 9/16, 1/4, 7/32, 1/8, 7/16, 1/32, 5/16, 9/32>

a = 69/61 et les carrés :

<33/61, 36/61, 28/61, 5/61, 2/61, 9/61, 25/61, 7/61, 16/61>

6 autres réponses décrivent des assemblages symétriques de ceux-ci.

Indications : Pour retrouver les positions des différents carrés dans le rectangle on peut procéder ainsi.

- On remplit le rectangle en utilisant successivement chaque carré de la liste C dans son ordre d'apparition.

- A chaque étape on considère tous les coins libres qui ont la même orientation que le coin gauche inférieur du rectangle et on choisit celui qui se trouve le plus à droite pour y placer le carré.

Remarque : Il existe une littérature importante autour de ce problème.

→ Deux résultats importants :

1- Il a été montré que quelque soit le nombre rationnel $a = 1$ il existe toujours un entier n tel que le rectangle de dimension $1 \times a$ puisse être rempli par n carrés de tailles distinctes.

2- Dans le cas $a = 1$, c-à-d. lorsque le rectangle à remplir est un carré, il a été montré que le plus petit n possible est $n = 21$.

N.B. : le Bin packing regroupe plusieurs techniques : BT, PrD, Greedy, …

XI- Algorithmes de recherche et optimisation

Best-first, first-fail (dans une certaine mesure, vu aussi dans TSP), A et A* sont des stratégies d'optimisation de la recherche.

Cas **adversatif** : Un autre exemple : Min-Max / Alpha-Beta

B & B est une méta stratégie de plus haut niveau (ne dépend pas d'adversaire).

L'exemple Morpion permet d'illustrer plusieurs stratégies.

XII- La stratégie Min-Max

Stratégie employée dans les jeux entre deux adversaires.

Cas de morpion où on place des 'X' et des 'O'.

1/ Au départ, on développe l'arbre complet (Figure suivante)
2/ On trouve la meilleure case où placer la 1er 'X' en appliquant la fonction Evaluation_MinMax(S? état) où S est l'état initial.
3/ En fonction du jeu de l'adversaire qui place son 1er 'O', une bonne partie de l'arbre (7/9) devient inutile (pour cette étape).
On évalue le meilleur coup à jouer par la même fonction que en étape (2)
4/ …...

N.B. : En toute rigueur, on peut développer l'arbre à partir d'un état donné (pas seulement le départ).

Une manière pratique de construire l'arbre Min-Max est de partir des états terminaux (gain, perte ou égalité), de les évaluer par une fonction d'évaluation (+1, -1 et 0 est un exemple) et de remonter ces valeurs niveau par niveau en fonction du type (*Min* ou *Max*) du niveau, jusqu'à la racine.

La Figure ci-dessous montre un exemple partiel de l'arbre Min-Max pour le jeu morpion.

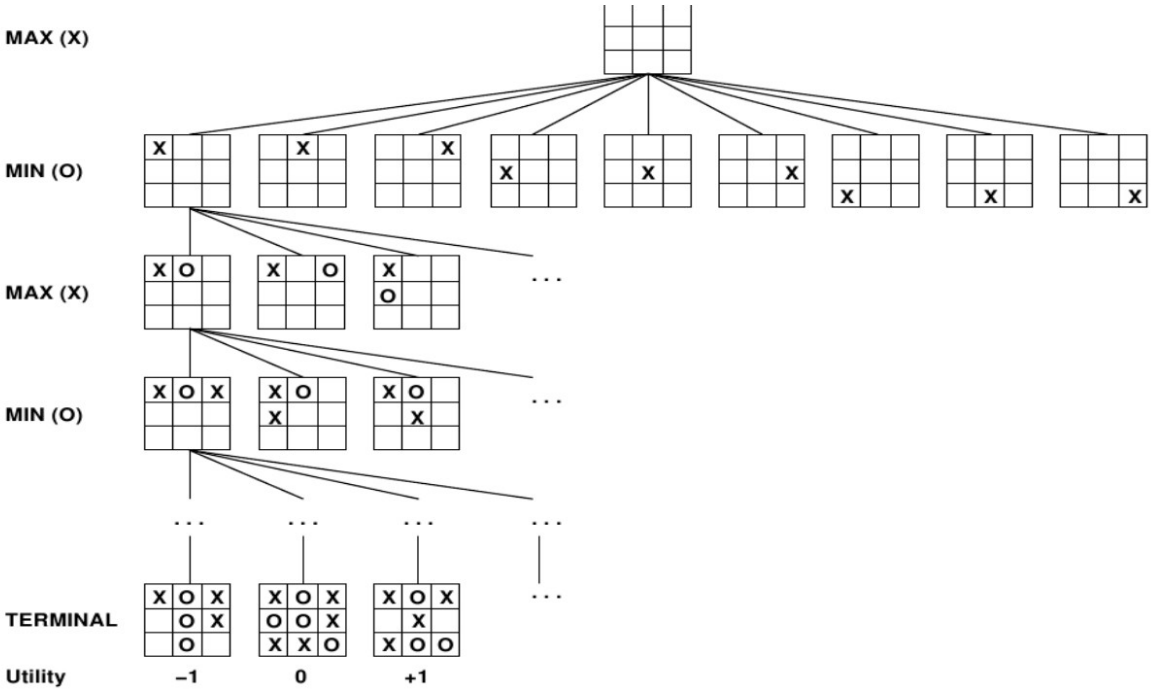
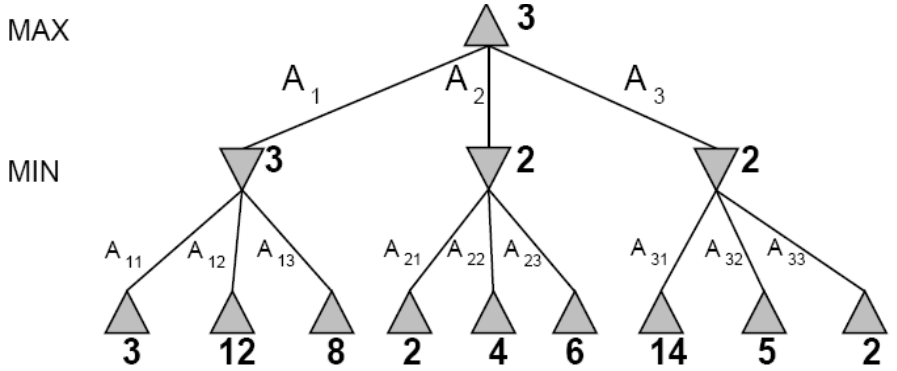


Figure ? le nœud racine au sommet est l'état initial et MAX (le X) joue le 1er. L'arbre donne les différents choix possibles pour MIN (le O) et pour MAX jusqu'à ce que l'un des états terminaux soit atteint.

La Figure suivante montre un exemple où les valeurs terminales sont remontées selon la nature du niveau.

La **décision MinMax = 3** correspondant au nœud A_{11} de l'arbre.



Cette décision **maximise** effectivement la valeur de la fonction d'évaluation (nœuds terminaux) sous l'hypothèse que l'adversaire jouerait son meilleur coup pour la **minimiser**.

Figure : Ici, les nœuds de la forme 'A' (base du triangle vers le bas) se déplacent par Max et les nœuds de la forme 'V' (base du triangle vers le haut) par Min.

Les nœuds terminaux représentent des valeurs "Utilitaires" (*Utility*) calculées pour Max.

Les autres valeurs par l'algorithme ci-dessous.

XII.1- Algorithme MinMax

Construire tout l'arbre du jeux
Évaluer chaque nœud S par la fonction *Evaluation_MinMax(S)*

Evaluation_MinMax(**S**? état)? Valeur de S

1) Si **S** est un état terminal (fin de la partie, gain/perte/égalité)
   Alors Valeur = valeur du nœud (= valeur *utility*)
2) Si **S** est un nœud Max
   Alors Valeur = maximum (Evaluation_MinMax(S')) pour tout S' = succ(S)
3) Si S est un nœud Min
   Alors Valeur = minimum (Evaluation_MinMax(S')) pour tout S', succ(S)

XII.2- Comprendre le principe de Min-Max

On considère les jeux à deux joueurs, avec :

- l'information complète (chaque joueur a la connaissance complète du jeu entier),
- sans hasard (pas de lancement de dés ou cartes par exemple),
- avec la somme nulle (la somme des gains des deux joueurs est zéro).

→ Le jeu d'échec sont un exemple typique.

- Le joueur qui commence est dénoté par A et l'autre par B.
- Considérons seulement les gains simples, c-à-d : 1 si A gagne (alors B perd), -1 si A perd (et victoire de B), 0 si égalité.
- Pour ce type de jeu, il y a toujours une **stratégie de gain**, qui est une manière de jouer pour un des joueurs qui, indépendamment des mouvements de l'adversaire, s'assure qu'il ou elle ne perd pas. Ceci signifie que A a un gain ≥ 0 (si A a une stratégie gagnante), B de gain ≤ 0 (si B a une stratégie de gain mais qui perd devant A).
- On peut montrer qu'il y a toujours une stratégie gagnante, mais il n'est pas possible de dire a priori lequel des joueurs la possède (dépend du jeu, et les deux cas se produisent).

XII.2.1- Arborescence associée

Comment une stratégie de gain (gagnante) peut être trouvée algorithmiquement ?

La démarche (algorithmique) de la mise en place constituera une preuve constructive de l'existence de cette stratégie.

Pour ce faire, on utilise une arborescence afin d'appliquer une recherche :

- La racine de cette arborescence est l'état initial du jeu.
- Les nœuds descendants de la racine sont tous états de jeux obtenus après le premier mouvement du joueur A.

- Les nœuds du niveau suivant s'obtiennent après un premier mouvement de B, et ainsi de suite, alternant les coups de A et de B.

- Les feuilles terminales (tout en bas de l'arbre) sont des états obtenu par une succession alternée (A et B) de mouvements qui représentent alors un match, après quoi il n'y a plus de mouvement (le jeu étant terminé).

- Pour chaque feuille, le gain est évalué comme ci-dessus :

1 si A gagne, –1 si B gagne et 0 pour une un match nul.

N.B. : Un état donné du jeu peut apparaître plusieurs fois dans l'arborescence.

C'est inhérent à ce modèle du jeu, puisqu'une situation donnée dans un jeu peut généralement être obtenue par des séquences différentes de mouvements.

XII.2.2- Un exemple : Nim

- L'arborescence d'un jeu peut vite devenir énorme.

- Pour présenter un cas accessible, considérons le jeu de **Nim**.

- Sous sa forme classique (popularisée par un film français dans les années 60), il y a quatre piles d'allumettes contenant respectivement 1, 3, 5, et 7 allumettes.

- Alternativement, chaque joueur enlève autant d'allumettes (mais au moins un) de seulement une des piles.

Le joueur qui enlève la dernière allumette aura perdu.

Limitons-nous à une version réduite de ce jeu avec seulement deux piles d'un et de trois allumettes.

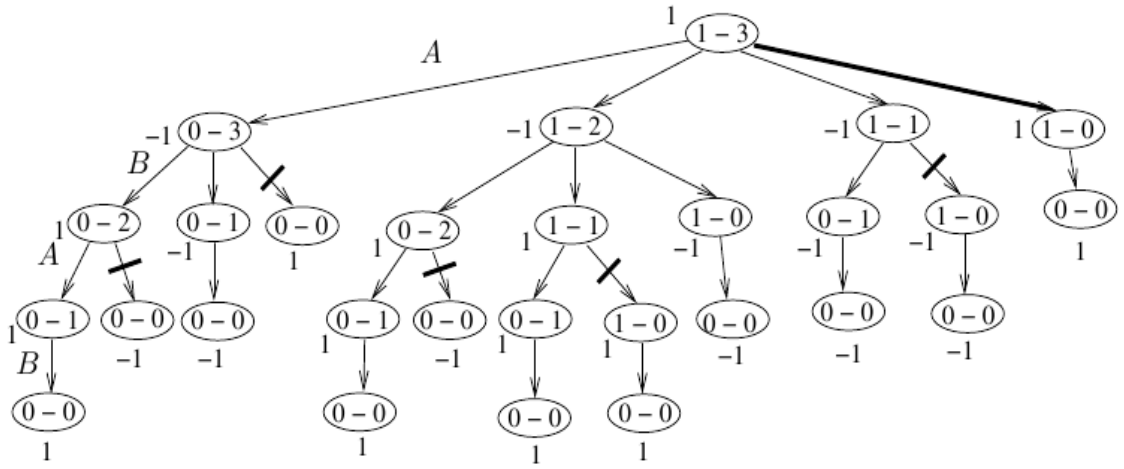


Figure : l'arborescence entière →

Chaque état :

‹allumettes Pile1 - allumettes Pile2›

→ Ex : *‹1-3›* pour l'état initial (une allumette sur la 1e pile et 3 sur la seconde).

N.B. : jeu simple mais l'arborescence est déjà un peu compliqué.

→ On voit que A doit jouer la branche de droite.

XII.3- Détails de l'algorithme de min-max

Trouver une stratégie de gain est plus difficile que l'optimisation simple d'une séquence de décisions (cf. Ci-dessus). En effet, il y a une poursuite antagonique entre les joueurs : celui qui commence (A) recherche un gain maximum, puisque c'est directement son gain.

L'autre joueur (B) essaye de réduire au minimum ce gain final puisque son gain est l'opposé de celui de A.

Le principe essentiel de l'algorithme est celui de remonter les gains depuis les feuilles (où les valeurs sont connues) vers la racine, où la valeur retournée indiquera lesquels des joueurs bénéficie d'une stratégie gagnante.

Généralement le gain retourné pour un sommet de l'arborescence égalera 1 si c'est une position de gain pour le joueur A, -1 si c'est une position perdante pour A (et donc gagnante pour B), 0 si ce n'est pas une position de gain pour aucun des joueurs.

Cette dernière situation est celle d'un jeu nul, et se produit si aucun des joueur ne fait une "erreur" :
c-à-d, quand chaque joueur évite, si possible, d'entreprendre une démarche menant à une situation de gain pour son adversaire.

Dans ce cas particulier, nous pouvons dire que chaque joueur a une stratégie de gain (une stratégie que l'on appellerait " non-perdante ").

Le principe décrit indique par lui-même comment calculer les gains.

Exemple :

Soit le cas d'un état du jeu où c'est le tour de A, et supposons que les gains de tous les nœuds dans l'arborescence ont déjà été déterminés.

La règle à appliquer est que le gain retourné à un nœud est le maximum des gains de ses nœuds descendants. Cela détermine également le "meilleur mouvement " pour A.

Une formule semblable s'applique dans le cas où c'est le tour du B, avec le *minimum* au lieu du *maximum*.

Ainsi, du fond jusqu'au sommet (des feuilles à la racine), il est possible de déterminer les valeurs de gain retournées pour chaque sommet de l'arborescence et, *in fine,* pour la racine.

Cette technique de renvoyer alternativement un minimum et un maximum explique le nom **min-max** donné à cet algorithme.

XII.4- Implantation de MinMax

Un parcours en profondeur de l'arborescence du jeu est parfaitement adaptée.

À chaque post-ordre (ou postfixe) = examen d'un nœud après ceux des ses descendants d'un nœud, la valeur de son parent est mise à jour par le maximum ou le minimum selon que c'est un mouvement par A ou pour B.

→ Car, à ce moment des calculs, tous les nœuds descendants du sommet considéré ont une valeur de gain connue.

Pour éviter de distinguer le traitement du cas de post-ordre du premier descendant d'un nœud (lorsque le nœud parent n'a pas encore de valeur calculée) des cas des autres descendants du même nœud, on applique la formule simple suivante :

Initialiser dans le pré-ordre la valeur de chaque sommet par -1 si c'est le tour de A, +1 si c'est le tour du B. Ces valeurs sont choisies telles que la valeur retournée la première fois soit automatiquement calculée.

Par exemple, à la première valeur renvoyée par le descendant d'un nœud représentant la situation où A doit jouer sera le maximum choisi entre -1 et le gain *g* renvoyé (qui sera -1, +1 ou 0). Par conséquent, le gain pris sera forcément la valeur *g*. Le même résultat sera obtenu par B avec un minimum entre +1 et un gain égal à +1, -1 ou 0.

→ Voir BE2.

Table des matières

I- Introduction aux Graphes....2

I.1- Exemples d'applications (modélisables par les graphes)....3

II- Graphes : quelques notions....6

II.1- Exemple (de graphe)....11
II.2- Structure de données pour représenter un graphe....12
II.2.1- Représentation statique par une matrice d'adjacence....13
II.2.2- Représentation dynamique par les listes d'adjacences....15
II.2.3- Représentation Hétérogène....16
Un autre exemple de représentation Hétérogène (Python)....17
II.2.4- Un type (TDA) Graphe....18

III- Algorithmes notables de parcours de graphes (récursifs / itératifs)....20

IV- Le principe de parcours récursif en profondeur (pré-ordre)....21

IV.1.1- Trace de parcours en profondeur....24

V- Le principe de parcours en largeur....26

V.1- Le Type (TDA) File....27
V.2- L'algorithme récursif de parcours en largeur....28
V.3- L'algorithme itératif de parcours en largeur....31
V.3.1- Trace de parcours en largeur....32
V.4- L'algorithme itératif de parcours en largeur avec marquage....33
V.4.1- Une variante parcours en largeur itératif (avec marquage)....34

VI- Applications des parcours de graphes....35

VI.1- Exemple simple : comptage du nombre de nœuds....35
VI.2- Fonction recherche d'un Élément....36
VI.3- Recherche de *chemin en Profondeur*....37
VI.3.1- Amélioration du calcul du chemin (en Profondeur)....38
VI.4- Autres algorithmes....39

VII- Méta-Stratégies générales de résolution....40

VII.1- Les techniques qui *regardent en arrière*....40
VII.2- Les techniques qui *regardent en avant*....41
VII.2.1- Illustration : exemple N-reines....43

VIII- Aspects pratiques du Back-tracking....50

VIII.1- Algorithme de principe du Back-tracking....51
VIII.2- Applications de BT....53
VIII.2.1- Exemple-1 : placements sous conditions (N-reines)....54
VIII.2.1.a- Complexité de N reines (en termes du nombre d'états visités)....59
VIII.2.2- Exemple-2 : le problème de la somme des sous ensembles....62
VIII.2.3- Exemple-3 : Coloration de graphes....68
VIII.2.3.a- Algorithme BT de coloration de graphes....69
VIII.2.3.b- La complexité de l'algorithme BT de coloration....71

IX- Approches de conception d'algorithmes : stratégies....72

IX.1- Stratégies Diviser et Régner....74
IX.1.1- Exemples....74
IX.1.2- Multiplication de nombres binaires....75
IX.1.3- Calcul de la complexité....77
IX.2- Principe et Stratégie de Programmation Dynamique....80

IX.2.1- Exemple 1 - calcul binomial....83
IX.3- Caractéristiques de la PrD....86
IX.3.1- Exemple-2 : Distance de Levenshtein....92
IX.3.2- Le Pseudo-Algorithme de Levenshtien....94
IX.3.3- Exemple 3 : algorithme de Floyd....96
IX.3.4- Algorithme de Floyd....103
IX.3.5- Obtention des trajets (coming from)....104
IX.4- Programmation Dynamique et la question d'optimisation....105
IX.4.1- Une variante de Floyd : algorithme de base de Roy-Warshall....108
IX.4.2- PrD et le problème du Voyageur de commerce (TSP)....111
IX.4.3- PrD : Algorithme TSP....117
IX.4.4- PrD et le problème de Sac à Dos....119
IX.5- Stratégies Greedy....122
IX.6- Exemples et comparaison avec PrD....123
IX.7- Un exemple d'algorithme greedy : coloration de graphe....124
IX.8- Comparaison BT et PrD....126
IX.8.1- Rappel d'exemples BT....126
IX.9- Branch & Bound....127
IX.9.1- Exemples....127
IX.10- Un exemple Best First : coloration heuristique....128
IX.10.1- Une solution First Fail à la coloration....130
IX.10.1.a- Exercices sur la coloration....133

X- Le problème Bin Packing : un cas de Best First....134

XI- Algorithmes de recherche et optimisation....139

XII- La stratégie Min-Max....140

XII.1- Algorithme MinMax....143
XII.2- Comprendre le principe de Min-Max....144
XII.2.1- Arborescence associée....145
XII.2.2- Un exemple : Nim....147
XII.3- Détails de l'algorithme de min-max....149
XII.4- Implantation de MinMax....152