

Stratégies de Résolution de Problèmes
BE1
AES et Parcours du Cavalier

École Centrale de Lyon
2022-2023

Alexander Saidi

I.1 BE1 : AES

Circuit Euler / cas particulier Hamiltonien

Plan :

1. *Introduction aux algorithmes à essais successifs*
2. *Parcours basique du Cavalier*
3. *Optimisation : ajout d'une bande*
4. *Optimisation : Heuristique*

Sujet du BE : réalisation du parcours d'un cavalier sur un échiquier $N \times N$.

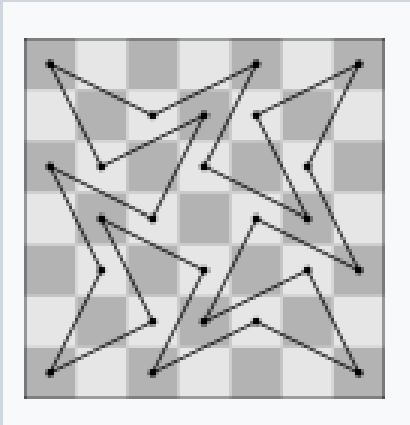
- *Par un algorithme à Essais Successif*

→ une classe d'algorithmes utilisés dans de nombreuses méthodes de résolution.

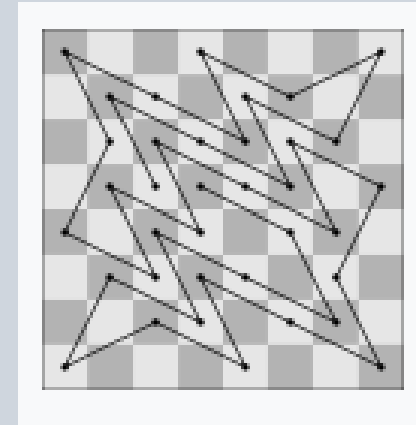
- **Travail à rendre :**

- Version de base (avec mesure du nbr de BT)
- Version Heuristique
- La version avec Bande est optionnelle

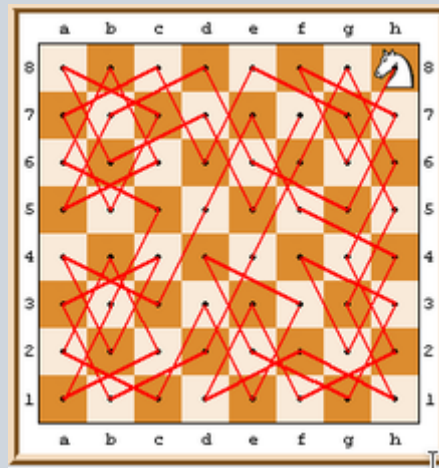
I.2 Exemples



Euler : Parcours sans croisement maximal sur un échiquier de taille 7×7
(total de 24 sauts)

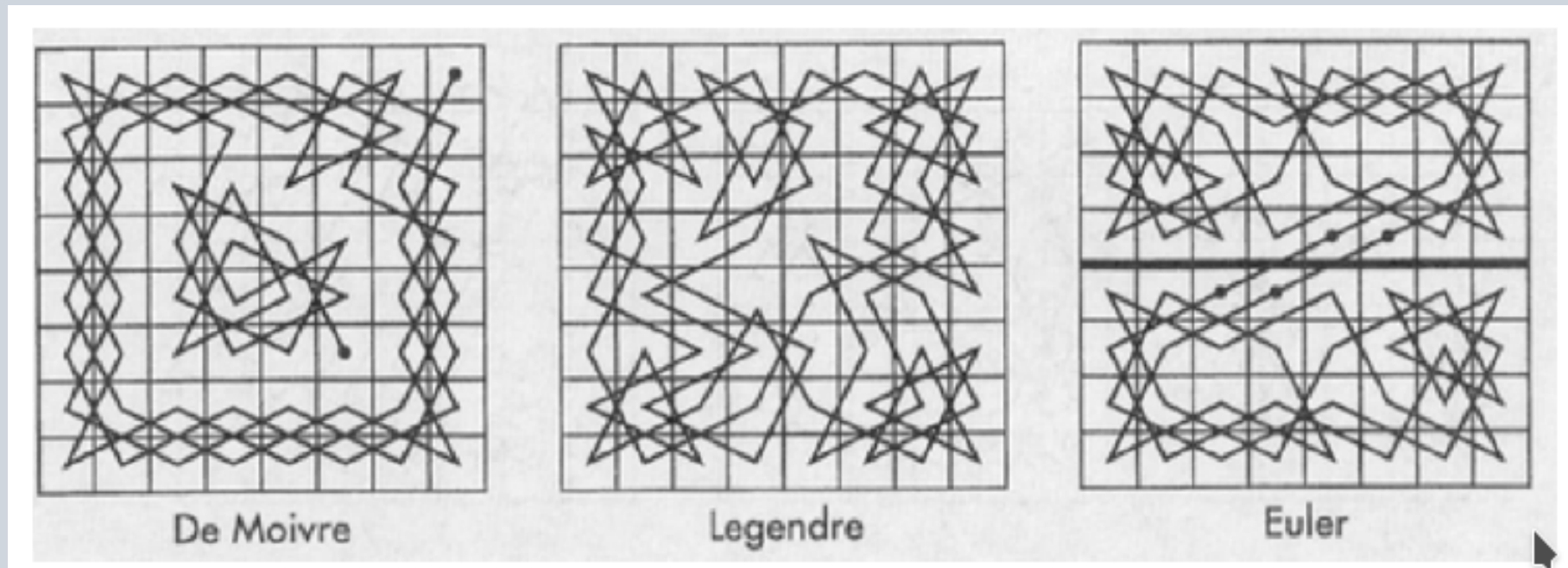


Euler : Parcours sans croisement maximal sur un échiquier de taille 7×7
(total de 35 sauts)



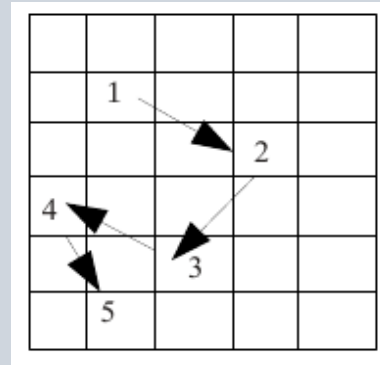
Rumi (solution en l'an 840!)

D'autres exemples de réalisation :



I.3 Le saut (le mouvement) du cavalier

- Le saut du cavalier du jeu d'échec (en L, angles toujours droit) :



Un exemple avec l'échiquier numéroté :

- départ = $\langle 0, 0 \rangle$ et un matrice de taille $N=5$ (on inscrit de 1 à 25) :

1	14	19	8	25
6	9	2	13	18
15	20	7	24	3
10	5	22	17	12
21	16	11	4	23

☞ La taille doit être $N > 4$

I.4 Principe d'un algorithme AES (parcours exhaustif DFS) de base

Algorithme de principe AES_Basique :

Données : état_courant,
+ fonction_de_transition (pour faire un mouvement) $\sigma : \text{etat} \times \text{action} \rightarrow \text{etat}$
Résultat : {Succès, Echec}

Si état_courant == état_final
Alors renvoyer **Succès**
Sinon
 Pour tout **état_successeur** = $\sigma(\text{etat_courant}, \text{une_action_possible})$:
 Si AES_Basique(état_successeur) == succès
 Alors renvoyer **Succès** (*)
renvoyer **Echec**

☞ Bien noter (*)

☞ Marquage (pour ne pas tourner en rond) : voir plus loin.

Algorithme adapté à l'échiquier :

Fonction AES_le_premier_succes_suffit;

Données : G : un graphe d'états;

Noeud_courant : le noeud (ou état) courant que l'on traite dans cet appel

Résultat : Succès ou Echec (un booléen)

début

si Noeud_courant *est un état final* **alors**

 renvoyer **Succès**

sinon

pour tous les Noeud_suivant **successeurs de** Noeud_courant *dans* G **faire**

si Prometteur(G, Noeud_suivant) **alors**

si AES_le_premier_succes_suffit(G, Noeud_suivant) = Succès **alors**

 renvoyer **Succès**

fin

fin

fin

 renvoyer **Échec**

fin

fin

☞ **Question de marquage** : pour ne pas tourner en rond !

→ Habituellement, on marque un choix (*Noeud_suivant est déjà_vu*).

→ Mais ici, Prometteur(.,.) filtre les cases déjà visitées (voir page suivante)

☞ Parcours en profondeur (**Depth First**)

I.4.1 Adaptation à un labyrinthe

Fonction AES_le_premier_succes_suffit_adapte_au_labyrinthe ;

Données : G : un graphe d'états = le plan du labyrinthe ;

Noeud_courant : le dernier couloir emprunté qui a mené à un choix / carrefour

Résultat : Succès ou Echec (*on s'en sort ou pas*)

```

début
  si Noeud_courant est un état final = la sortie en vue alors
    | renvoyer Succès
  sinon
    pour tous les Noeud_suivant = une des directions possibles à partir du Noeud_courant dans G faire
      si Prometteur(G, Noeud_suivant) = on peut effectivement emprunter la direction choisie alors
        | si AES_le_premier_succes_suffit_adapte_au_labyrinthe(G, Noeud_suivant) = Succès alors
          | | renvoyer Succès
          | fin
        fin
      fin
    renvoyer Échec
  fin
fin

```

👉 Ici, la fonction *Prometteur(.,.)* permet de choisir une voie si :

- elle n'est pas déjà empruntés
- ne mène pas à un trou / falaise
- mène à un couloir empruntable (n'est pas inondé / habité par des zombies), etc.

I.5 Adaptation au Parcours Cavalier (basique)

Fonction AES_parours_cavalier_un_succès_suffit;

Données : G : un graphe sous forme de matrice;

N : taille échiquier

Case_actuelle : < X, Y > : les coordonnées de la case qu'on visite dans cet appel,

Num_etape : entier numéro de la prochaine case,

Résultat : Succès ou Echec (un booléen) // en cas de succès, G contient le résultat

début

si Num_etape > N^2 alors

renvoyer Succès // N^2 déjà inscrit, on a réussi! **On arrête sur le 1er succès**

sinon

pour tous les Case_suivante **successeur de** Case_actuelle *dans* G **faire**

si Prometteur(G, N, Case_suivante) alors

Inscrire Num_etape dans Case_suivante (*)

si AES_parours_cavalier_un_succès_suffit(G, N, Case_suivante, Num_etape + 1) = Succès alors

renvoyer Succès

fin

Ici, **il faut effacer** Num_etape inscrit dans Case_suivante. (**)


fin

fin

renvoyer Echec

fin

fin

 **Question de marquage** : (*) = marquage, (**) = démarquage.

I.6 Aspects pratiques

Calcul pratique des voisins d'une case :

- Le graphe (l'échiquier) : une matrice $N \times N$.
- Les cases de la matrice $M[N][N]$ initialisée par -1
- La case actuelle $\langle X, Y \rangle$ dont on calcule les voisins.
- On sait qu'une case possède au plus 8 voisins (pour un saut de cavalier).
- Les successeurs possibles de la case $\langle X, Y \rangle$:
 $\langle X \pm 1, Y \pm 2 \rangle$
 $\langle X \pm 2, Y \pm 1 \rangle$

Ces valeurs reflètent le mouvement en L du cavalier.

	X		X	
X				X
		●		
X				X
	X		X	

- Déclaration (et initialisation) :

```
Tab_delta_X_Y_Y = [[1,2], [1,-2], [-1,2], [-1,-2], [2,1], [2,-1], [-2,1], [-2,-1]]
```

Exemple : le 4ème voisin de la case $\langle X, Y \rangle$:

$X + \text{Tab_delta_X_Y}[3][0]$ et $Y + \text{Tab_delta_X_Y}[3][1]$

Pour $i=0..7$

$Nx = X + \text{Tab_delta_X_Y}[i][0];$ $Ny = Y + \text{Tab_delta_X_Y}[i][1]$

I.7 Algorithme concret de parcours du cavalier

```

bool AES_parcours_cavalier_un_succes_suffit(Echiquier, Taille,
                                             Dernière_Case_Traitée, Prochain_Num_etape) :
    → // la Dernière_Case_Traitée est un couple < X, Y >

Résultat : Succès ou Echec (un booléen)      // en cas de succès, G contient le résultat

début
    si Prochain_Num_etape > Taille * Taille alors
        renvoyer Vrai      // N2 déjà inscrit, on a réussi! On arrête sur le 1er succès
    sinon
        pour tous les i = 0..7 faire
            Next_X = X + Tab_delta_X_Y[i][0];
            Next_Y = Y + Tab_delta_X_Y[i][1];
            si Prometteur(G, Taille, Next_X, Next_Y) alors
                Echiquier[Next_X][Next_Y] = Prochain_Num_etape
                si AES_parcours_cavalier_un_succès_suffit(Echiquier, Taille,
                                                            < Next_X, Next_Y >, Prochain_Num_etape+1 ) alors
                    renvoyer Vrai
                fin
                Echiquier[Next_X][Next_Y] = -1      // Ca n'a pas marché, on efface nos traces!
            fin
        fin
        renvoyer Faux      // Aucun des successeurs n'a abouti.
    fin
fin

```

Fonction main :

main(...) :

Définir la taille **N**

Créer la matrice[N][N] *Echiquier*[N][N] initialisée toutes cases à -1

Définir **<X,Y>** la case de départ (lecture au clavier)

Echiquier[X][Y]=1 // numéro de l'étape = 1

prochaine_numéro = 2 // prochaine case aura le numéro 2

si **AES_parcours_cavalier_un_succes_suffit**(*Echiquier*, N, X, Y, prochaine_numéro)

alors afficher(*Echiquier*)

sinon afficher("échec")

Finsi

Fonction prometteur :

bool **prometteur**(*Echiquier*, Taille, New_X, New_Y) :

<New_X, New_Y> est une case valide dans *Echiquier* si les tests suivants réussissent :

New_X \geq 0; New_X < Taille

New_Y \geq 0; New_Y < Taille

Echiquier[New_X][New_Y] == -1 // case non encore visitée = nœud du graphe non exploité

Renvoyer le résultat de ces tests (vrai ou faux)

☞ il y a 5 tests dans la fonction prometteur.

I.8 Un exemple

départ = $\langle 0, 0 \rangle$ et une matrice de taille 5 (on inscrit de 1 à 25) :

1	14	19	8	25
6	9	2	13	18
15	20	7	24	3
10	5	22	17	12
21	16	11	4	23

☞ La taille doit être $N > 4$

- Question de complexité : voir le sujet + cours 1.

I.9 Différentes aides

- Voir la fin du sujet.

I.10 Travail 1 à rendre

- Pour cette étape : codage de la première version. Noté maximum 12/20.
- **Prévoir des compteurs** : tentatives (nbr de successeurs testés), nbr de retours arrières.
- La Taille et la matrice peuvent être globales (prévoir "__main__")

I.11 Obtenir toutes les solutions possibles

- Cas algorithme AES générique (le seul changement est en rouge)

Fonction AES_toutes_solutions;

Données : G : un graphe ;

Noeud_courant : le noeud courant que l'on traite

Résultat : Succès ou Echec (un booléen)

début

si Noeud_courant *est un état final* **alors**

 Afficher ou enregistrer la solution qu'on vient de trouver

 Renvoyer Echec : Simuler un échec pour forcer à trouver les autres solutions

sinon

pour tous les Noeud_suivant **successeur de** Noeud_courant *dans* G **faire**

si Prometteur(G, Noeud_suivant) **alors**

si AES_toutes_solutions(G, Noeud_suivant) = Succès **alors**

 renvoyer Succès

fin

fin

fin

 renvoyer Echec

fin

fin

- Parcours en profondeur (Depth First) exhaustif : crée TOUT l'espace de recherche.
 - Donne toutes les solutions.

Et pour le cavalier (noter ce qui est en **rouge** avec possibilité de décider) :

Fonction AES_parcours_cavalier_toutes_solutions;

Données : G : un graphe sous forme de matrice;

N : taille échiquier

Case_actuelle : < X, Y > : les coordonnées de la case qu'on visite dans cet appel,

Num_etape : entier numéro de la prochaine case,

Résultat : Succès ou Echec (un booléen) // en cas de succès, G contient le résultat

début

si Num_etape > N^2 **alors**

Proposer la solution = afficher la matrice

Demander si l'on souhaite poursuivre?

Si poursuite demandée alors renvoyer échec (on simule)

Sinon renvoyer Succès

sinon

// Le reste de l'algorithme ne change pas

pour tous les Case_suivante **successeur de** Case_actuelle *dans* G **faire**

si Prometteur(G, N, Case_suivante) **alors**

Inscrire Num_etape dans Case_suivante

si AES_parcours_cavalier_toutes_solutions(G, N, Case_suivante, Num_etape + 1) = Succès **alors**

renvoyer Succès

fin

Ici, il faut effacer Num_etape dans Case_suivante. VOIR PLUS LOIN

fin

fin

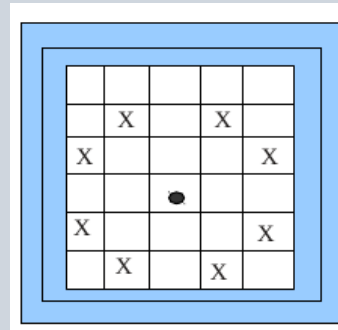
renvoyer Echec

fin

fin

I.12 Optimisations et améliorations

- Réduire le nombre de tests dans Prometteur.
- Pour ce faire :
 - Bande périphérique



- Attention aux indices

I.13 Travail 2 à rendre

- Pour cette étape : appliquer la bande.
 - Optionnel mais conseillé.
 - Bonus de 2/20

I.14 Optimisations via Heuristique

- Soit $\langle X, Y \rangle$ la case actuelle et $\langle \text{Next_X}_i, \text{Next_Y}_i \rangle$ un des 8 voisins non-visités possibles.
 - Chacune des $\langle \text{Next_X}_i, \text{Next_Y}_i \rangle$ possède à son tour jusqu'à 8 voisins libres possibles.
 - Choisir $\langle \text{Next_X}_i, \text{Next_Y}_i \rangle$ non-visitée avec le plus petit nombre de voisins libres.
- ☞ Si une solution avec ce choix existe, on l'atteindra.

Exemple : départ = $\langle 0, 0 \rangle$ (avec 0 retours arrières)

1	12	25	18	3
22	17	2	13	24
11	8	23	4	19
16	21	6	9	14
7	10	15	20	5

- Avec cette heuristique, on pourrait penser à une stratégie (quasi) **irrévocable** :
 - ➔ pas besoin de se soucier des Back-Tracks.
 - ➔ Si vous êtes certain de ne pas revenir sur vos pas, vous ne jetterez pas de miettes de pain!

☞ Comment appelle-t-on cette stratégie ? :

Best First (révocable ou non) au sein d'une stratégie d'affectation **First-Fail** (V. cours)

- La mise en place de l'heuristique nécessite une 2e matrice (contenant en **permanence** le nombre des voisins libres d'une case).

Pour $N=5$, la matrice initiale des voisins sera :

La matrice des voisins initiale :

2	3	4	3	2
3	4	6	4	3
4	6	8	6	4
3	4	6	4	3
2	3	4	3	2

- Voir le sujet du BE, le travail final à rendre + un bonus éventuel.
- ☞ Eviter de faire les calculs à la main, voir la fin du sujet BE

I.15 Travail 3 à rendre

- Réaliser cette version (avec les compteurs)
- Noté 12/20

I.16 Important

- **Question** : avec cette heuristique, travaille-t-on avec un algorithme irrévocable ?

C.à.d., ne pas organiser les BT : ne plus remettre -1 dans une case ?

Réponse : Non : il est possible qu'un choix échoue tandis qu'une solution existe.

- Il faut donc prévoir les retours arrières.

S'il y a plusieurs "meilleurs voisins", prévoir de les visiter tous.

- **Voici comment on peut faire** : quand on est en une étape avec la case actuelle $\langle X, Y \rangle$:
 - Mettre dans un tableau / liste (de taille 8 max) le nombre de voisins libres de chaque successeur possible de $\langle X, Y \rangle$ (à l'aide de la matrice des voisins)
 - Trouver le meilleur voisin dans ce tableau, soit : $\langle \text{Next}_X, \text{Next}_Y \rangle$
 - Si plusieurs meilleurs voisins (même degré de liberté) alors possibilité de BT
 - Mettre à jour la matrice des des voisins libres (-1 sur le nb voisins libres de $\langle \text{Next}_X, \text{Next}_Y \rangle$)
 - Lors d'un retour-arrière : refaire +1 sur le nombre de voisins de la case qu'on emprunte.
- Le reste est comme avant !

👉 Voir la fin du sujet pour différents codes sur l'heuristique.

Algorithme détaillé : (noter en **bleu** les ajouts/modifs)

```

bool AES_cavalier_heuristique_un_succes_suffit(Echiquier, Matrice_Nb_Voisins_libres ,
        Taille, Dernière_Case_Traitée, Prochain_Num_etape) :
    → // la Dernière_Case_Traitée est un couple (X, Y)

Résultat : Succès ou Echec (un booléen)      // en cas de succès, Echiquier contient le résultat

début
    si Num_etape > Taille * Taille alors
        | renvoyer Vrai      // N2 déjà inscrit, on a réussi! On arrête sur le 1er succès
    sinon
        // On va chercher les meilleurs voisins (plusieurs possibles avec le même degré de liberté )
        Vecteur_des_meilleurs_Voisins = Trouver_Meilleurs_Voisins_Libres (
            Echiquier, Matrice_Nb_Voisins_libres , Taille, Dernière_Case_Traitée);
        pour tous les couples (Next_X, Next_Y) de Vecteur_des_meilleurs_Voisins faire
            Echiquier[Next_X][Next_Y] = Prochain_Num_etape
            Mise_A_Jour_Voisins(Matrice_Nb_Voisins_libres, (Next_X, Next_Y))
            si AES_cavalier_heuristique_un_succes_suffit(Echiquier, Matrice_Nb_Voisins_libres , Taille,
                (New_X, New_Y), Prochain_Num_etape+1 ) alors
                | renvoyer Vrai
            fin
            Echiquier[Next_X][Next_Y] = -1      // Comme avant : ca n'a pas marché, on efface nos traces!
            Défaire_la_Mise_A_Jour(Matrice_Nb_Voisins_libres, (Next_X, Next_Y))
        fin
        renvoyer Faux      // Aucun des successeurs n'a abouti.
    fin
fin

```

I.17 Annexe : DAT

L'algorithme AES est utilisable dans de nombreux domaines dont en démonstration automatique de théorèmes (DAT).

```
def prometteur(rule, theo_a_demontrer) :
    # est ce que la partie gche de la règle correspond à theo
    return rule[0]==theo_a_demontrer

def AES_un_succes_suffira(liste_theoremes, niv_trace = 1) :
    print(f'{ ' *niv_trace*3} {niv_trace}
           On doit démontrer la CONJONCTION : {liste_theoremes}')
    if liste_theoremes == [] : return True
    # On choisit LA PREMEIRE THEOREME dans la liste des théorèmes à prouver
    theo_a_demontrer = liste_theoremes[0]

    if theo_a_demontrer in {True, False} : return AES_un_succes_suffira(liste_theoremes[1:], niv_trace + 1)

    for rule in g_regles : # On choisit LA PREMEIRE REGLE dont la tête = theo
        if prometteur(rule, theo_a_demontrer) :
            print(f'{ ' *niv_trace*3} Pour ce Theorème, on a trouvé
                  la règle {rule[0]} : {rule[1]}')
            new_base = rule[1] + liste_theoremes[1:]
            if AES_un_succes_suffira(new_base, niv_trace + 1) : return True
    return Falses
```

Utilisation :

Pour tester notre algorithme, soit une base d'axiomes :

$A \leftarrow B, C.$

$D \leftarrow C.$

$B \leftarrow \text{True}.$

$C \leftarrow \text{True}.$

```
if __name__ == "__main__" :  
    # TEST  
    g_regles1_bis = [  
        ("A", ["B", "C"]), # A : B,C  
        ("B", [True]),    # B.  
        ("C", [True]),    # C.  
        ("D", ["F"]),     # D : F  
        ("D", ["C"])      # D : C  
    ]  
    Q00_OK = ["A","D"]  
    g_regles = g_regles1_bis  
    print("La réponse à la question", Q00_OK, " = ",  
          AES_un_succes_suffira(Q00_OK, niv_trace = 1))
```

• Trace (succès) pour ["A", "D"] :

```

1  On doit démontrer la CONJUNCTION : ['A', 'D']
   Pour ce Théorème, on a trouvé la règle A : ['B', 'C']
2  On doit démontrer la CONJUNCTION : ['B', 'C', 'D']
   Pour ce Théorème, on a trouvé la règle B : [True]
3  On doit démontrer la CONJUNCTION : [True, 'C', 'D']
4  On doit démontrer la CONJUNCTION : ['C', 'D']
   Pour ce Théorème, on a trouvé la règle C : [True]
5  On doit démontrer la CONJUNCTION : [True, 'D']
6  On doit démontrer la CONJUNCTION : ['D']
   Pour ce Théorème, on a trouvé la règle D : ['F']
7  On doit démontrer la CONJUNCTION : ['F']
   Pour ce Théorème, on a trouvé la règle D : ['C']
7  On doit démontrer la CONJUNCTION : ['C']
   Pour ce Théorème, on a trouvé la règle C : [True]
8  On doit démontrer la CONJUNCTION : [True]
9  On doit démontrer la CONJUNCTION : []

```

La réponse à la question ['A', 'D'] = **True**

• Trace (échec) pour ["A", "F"] :

```

1  On doit démontrer la CONJUNCTION : ['A', 'F']
   Pour ce Théorème, on a trouvé la règle A : ['B', 'C']
2  On doit démontrer la CONJUNCTION : ['B', 'C', 'F']
   Pour ce Théorème, on a trouvé la règle B : [True]
3  On doit démontrer la CONJUNCTION : [True, 'C', 'F']
4  On doit démontrer la CONJUNCTION : ['C', 'F']
   Pour ce Théorème, on a trouvé la règle C : [True]
5  On doit démontrer la CONJUNCTION : [True, 'F']
6  On doit démontrer la CONJUNCTION : ['F']

```

La réponse à la question ['A', 'F'] = **False**

I.17.1 Pour aller plus loin

- Coder la base suivante
- Introduire les variables (voir avec l'enseignant!)

```
voiture_OK      :    batterie_OK, demarreur_OK, essence_OK.  
batterie_OK     :    klaxon_OK.  
batterie_OK     :    phares_OK.  
batterie_OK     :    bruit_demarrage.  
batterie_OK     :    jauge_OK.
```

```
jauge_OK       :    jauge_positive.
```

```
essence_OK     :    jauge_positive.  
essence_OK     :    carburant_visible.
```

```
demarreur_OK   :    bruit_demarrage.
```

```
# FAITS "constatables"  
jauge_positive.  
bruit_demarrage.
```

- ☞ Mettre en place la possibilité d'avoir une disjonction dans la requête.