CR TP

N.B

n.b : s'il n'est précisé, noter que le temps d'exécution de la requête est quasi-nul

n.b : non confondrons les termes de tableau & matrice pour parler de la grille de jeu du morpion

---

TP1

Question 1 - Familiarisation avec le problème du Taquin 3x3

1.2.a

```
% Avec la clause :
final_state_4([[1, 2, 3, 4],
             [5, 6, 7, 8],
             [9, 10, 11, 12],
             [13, 14, 15, vide]]).
% La requête suivante renverra la situation finale F du Taquin 4x4 :
?- final_state_4(S).
S = [[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12], [13, 14, 15, vide]].
```

1.2.b

1. Détermine la position de l'élément *d* dans la matrice *Ini* (cf. un initial_state déclaré avant)

```
% l'état initial en question
initial_state([ [b, h, c],
                [a, f, d],
                [g,vide,e] ]).

% C'EST L'EXEMPLE PRIS EN COURS
% h1=4,   h2=5,   f*=5
```

```
?- initial_state(Ini), nth1(L,Ini,Ligne), nth1(C,Ligne, d).
Ini = [[b, h, c], [a, f, d], [g, vide, e]],
L = 2,
Ligne = [a, f, d],
C = 3 .
% position trouvée pour l'élément d : [L,C] = [2,3] = 2° ligne, 3° colonne
```

2. Détermine l’élément positionné colonne / index 2 de la ligne 3 dans la matrice Fin (*final_state* (Fin)).

```
% l'état final étant :
final_state([[a, b,  c],
             [h,vide, d],
             [g, f,  e]]).
```

```
?- final_state(Fin), nth1(3,Fin,Ligne), nth1(2,Ligne,P).
Fin = [[a, b, c], [h, vide, d], [g, f, e]],
Ligne = [g, f, e],
P = f.
% l'élément trouvé est f (dans Fin, c'est bien l'élément à la position [Ligne,Colonne]
=[3,2].
```

1.2.c

```
% prédicat pour savoir si une pièce P est bien placée dans U0 par rapport à F :
is_in_place(P) :-
  initial_state(U0),
  final_state(F),
  % position de P dans Ini = [I1,I2]
  nth1(I1,U0,X),
  nth1(I2,X,P),
  % vérification de la position de P = [I1,I2]
  nth1(I1,F,Ligne),
  nth1(I2,Ligne,P).

% tests unitaires
% avec c, élément bien placé
?- is_in_place(c).
true .

% avec a, élément mal placé
?- is_in_place(a).
false.

% avec P, doit renvoyer tous les éléments de U0 bien placés par rapport à F, soit c,d,
```

```
e, et g.
?- is_in_place(P).
P = c ;
P = d ;
P = g ;
P = e.
```

1.2.d

```
?- initial_state(Ini), rule(A, 1, Ini, S2).
% A sera le coup joué
% 3 coups possibles depuis l'état initial : up, left, right
% les situations suivantes seront S2
Ini = [[b, h, c], [a, f, d], [g, vide, e]],
A = up,
S2 = [[b, h, c], [a, vide, d], [g, f, e]] ;
Ini = [[b, h, c], [a, f, d], [g, vide, e]],
A = left,
S2 = [[b, h, c], [a, f, d], [vide, g, e]] ;
Ini = [[b, h, c], [a, f, d], [g, vide, e]],
A = right,
S2 = [[b, h, c], [a, f, d], [g, e, vide]] ;
false.
```

1.2.e et 1.2.f

```
% pour regrouper les 3 situations successeurs dans une liste L :
?- initial_state(Ini), findall(S2, rule(X, 1, Ini, S2), L).
Ini = [[b, h, c], [a, f, d], [g, vide, e]],
L = [[[b, h, c], [a, vide, d], [g, f, e]], [[b, h, c], [a, f, d], [vide, g, e]], [[b,
 h, c], [a, f, d], [g, e, vide]]].
% L composée d'éléments de la forme [Situation_résultante]

% pour regrouper les situations successeurs avec la règle associée :
?- initial_state(Ini), findall((X,S2), rule(X, 1, Ini, S2), L2).
Ini = [[b, h, c], [a, f, d], [g, vide, e]],
L2 = [(up, [[b, h, c], [a, vide, d], [g, f, e]]),  (left, [[b, h, c], [a, f, d], [vid
e, g, e]]),  (right, [[b, h, c], [a, f, d], [g, e|...]])].
% L2 composée d'éléments de la forme [Règle, Situation_résultante]
```

Question 2 - Développement des 2 heuristiques

La première heuristique se base sur le nombre de pièces mal placées dans la situation U par rapport à la situation finale F.

2.1.a - Développement de l’heuristique 1

```
% coordonnees détermine [Ligne,Colonne]=[L,C] la position de l'élément Elt dans la mat
rice Mat
coordonnees([L,C], Mat, Elt) :-
        nth1(L,Mat,X), nth1(C,X,Elt).
% malplace renvoie true si les coordonnées de P sont les mêmes dans U et dans F (simil
aire question 1.2.c)
malplace(P,U,F) :-
    coordonnees([L,C],U,P), not(coordonnees([L,C],F,P)).
% heuristique1a pour déterminer H, la valeur de l'heuristique associée à la situation
 U en se servant du nb d'éléments de la liste des éléments mal placés.
heuristique1a(U, H) :-
        final_state(Fin), findall(X, (malplace(X, U, Fin), X\= vide), L), length(L,H).

% tests unitaires
% depuis la situation initiale, on doit avoir 4 pièces mal placées, soit H=4.
?- initial_state(Ini),heuristique1a(Ini,H).
Ini = [[b, h, c], [a, f, d], [g, vide, e]],
H = 4.
% depuis la situation finale, on ne doit avoir aucune pièce mal placée, soit H=0.
?- final_state(F),heuristique1a(F,H).
F = [[a, b, c], [h, vide, d], [g, f, e]],
H = 0.
```

2.1.b - Version récursive de l’heuristique 1

```
% heursitique1b(+U,?H)
heuristique1b(U,H):-
    final_state(F),
    heuristique1b(U,F,0,H).
heuristique1b([],[],Acu,Acu). % cas d'arrêt, matrices vides, l'heuristique H=Acu
heuristique1b([LU|RU],[LF|RF],Acu,H):-
    h1b_Ligne(LU,LF,H2),
    Acu2 is Acu + H2,
    heuristique1b(RU,RF,Acu2,H).
% h1b_Ligne calcule l'heuristique H pour une ligne donnée (on calcule l'heuristique su
r la situation U en regardant ligne par les ligne les différences entre U et F)
h1b_Ligne([],[],0). % cas d'arrêt, H=0 pour une ligne vide
h1b_Ligne([E|LU],[E|LF],H) :-
    h1b_Ligne(LU,LF,H).
h1b_Ligne([EU|LU],[EF|LF],H) :-
    h1b_Ligne(LU,LF,H2),
    EF\=EU,
    EU\= vide,
    H is H2+1.
h1b_Ligne([vide|LU],[EF|LF],H) :- % gestion du cas 'vide' (on le saute)
    h1b_Ligne(LU,LF,H),
    EF\= vide.

% tests unitaires
% depuis la situation initiale, on doit avoir 4 pièces mal placées, soit H=4.
?- initial_state(Ini),heuristique1b(Ini,H).
Ini = [[b, h, c], [a, f, d], [g, vide, e]],
H = 4 .
```

```
% depuis la situation finale, on ne doit avoir aucune pièce mal placée, soit H=0.
?- final_state(F),heuristique1b(F,H).
F = [[a, b, c], [h, vide, d], [g, f, e]],
H = 0 .
```

2.2 - Développement de l’heuristique par distance de Manhattan

Cette heuristique se base sur le nombre de déplacements à effectuer depuis la situation U pour arriver à la situation finale F.

```
% dm détermine la distance de Manhattan D pour l'élément Elt, entre la situation U et
 la situation F
dm(Elt,U,F,D):-
      coordonnees([L,C],U,Elt),
      coordonnees([L1,C1],F,Elt),
      D is (abs(L-L1)+abs(C-C1)).

% tests unitaires :
% entre l'initial state et le final state, les distances de Manhattan (DM) pour chaque
élément sont : a=1, b=1, c=0, d=0, e=0, f=1, g=0, h=2
?- initial_state(U), final_state(F), dm(Elt,U,F,D).
% b
U = [[b, h, c], [a, f, d], [g, vide, e]],
F = [[a, b, c], [h, vide, d], [g, f, e]],
Elt = b,
D = 1 ;
% h
U = [[b, h, c], [a, f, d], [g, vide, e]],
F = [[a, b, c], [h, vide, d], [g, f, e]],
Elt = h,
D = 2 ;
% c
U = [[b, h, c], [a, f, d], [g, vide, e]],
F = [[a, b, c], [h, vide, d], [g, f, e]],
Elt = c,
D = 0 ;
% a
U = [[b, h, c], [a, f, d], [g, vide, e]],
F = [[a, b, c], [h, vide, d], [g, f, e]],
Elt = a,
D = 1 ;
% f
U = [[b, h, c], [a, f, d], [g, vide, e]],
F = [[a, b, c], [h, vide, d], [g, f, e]],
Elt = f,
D = 1 ;
% d
U = [[b, h, c], [a, f, d], [g, vide, e]],
F = [[a, b, c], [h, vide, d], [g, f, e]],
Elt = d,
D = 0 ;
% g
```

```
U = [[b, h, c], [a, f, d], [g, vide, e]],
F = [[a, b, c], [h, vide, d], [g, f, e]],
Elt = g,
D = 0 ;
% vide (la gestion du non calcul de la DM pour vide se fait dans le prédicat heuristiq
ue2, la valeur renvoyée ici n'est donc jamais prise en compte=)
U = [[b, h, c], [a, f, d], [g, vide, e]],
F = [[a, b, c], [h, vide, d], [g, f, e]],
Elt = vide,
D = 1 ;
% e
U = [[b, h, c], [a, f, d], [g, vide, e]],
F = [[a, b, c], [h, vide, d], [g, f, e]],
Elt = e,
D = 0.
% ********************
% sum est un prédicat développé pour faire la somme des DM déterminées ci-dessus, four
nies en entrée dans une liste d'éléments à additionner
    sum([],0):-!.

    sum([T|Q],Somme) :-
      sum(Q,S),
      Somme is T + S.

% tests unitaires
?- sum([],S).
S = 0.

?- sum([1,1,1,1],S).
S = 4.

?- sum([1,1,1,0],S).
S = 3.

?- sum([0],S).
S = 0.

?- sum([-2,1,5,-4],S). % inutile pour nous, une DM ne peut être négative
S = 0.

% ********************
% heuristique2 renvoie la somme des DM entre la situation U et la situation finale F
   heuristique2(U, H) :-
      final_state(F),
      findall(X,(malplace(X,U,F),X \= vide),L),
      findall(D,(member(X,L),dm(X,U,F,D)),L2),
      sum(L2,H).

% tests unitaires
% depuis la situation initiale, on doit avoir 5 déplacements à effectuer, soit H=5.
?- initial_state(Ini),heuristique2(Ini,H).
Ini = [[b, h, c], [a, f, d], [g, vide, e]],
H = 5.
% depuis la situation finale, on ne doit avoir aucun déplacement, soit H=0.
?- final_state(F),heuristique2(F,H).
F = [[a, b, c], [h, vide, d], [g, f, e]],
H = 0.
```

Question 3 - Implémentation de A*

3.2 - A* adapté aux structures AVL choisies

```
% problème en développant le prédicat affiche_solution comme nous le voulions. Le préd
icat aetoile retournait dans le cas général (3° cas, voir dessous), et bouclait longte
mps avant de se terminer. Nous n'avons pas su expliquer la provenance de ce problème,
 ni le corriger (même en changeant les prédicats, avec des !, etc). affiche_solution s
era donc simplement un affichage de Qfinal. Dans l'idée, nous aurions récupéré les cou
ps joués et les aurions affichés à la suite, pour donner la combinaison de coups gagna
nte.
affiche_solution(Sol):-
    put_flat(Sol), nl.
```

```
% détermine la liste des successeurs d'une situation S (coups jouables) ainsi que les
 coûts associés (F,H,G) et la règle appliquée A
expand([[_Fu,_Hu,Gu],S],Lsuc):-
    G is Gu+1,
    findall((S2,A,[F,H,G]), (rule(A, 1, S, S2), heuristique2(S2,H), F is G+H), Lsuc).
```

Pour développer *loop_successors*, il a fallu développer un prédicat intermédiaire *comp_F* pour comparer deux valeurs de F concernant une même situation.

```
% si la nouvelle valeur de F est meilleure que l'ancienne Old_F, il faut la mettre à j
our dans Pf et Pu
comp_F(Old_F,Pf,Pu,S,[F,H,G],Pere,A,Pf_new,Pu_new):-
    F < Old_F,
    suppress([S,[_F,_H,_G],_Pere,_A],Pu,Pui),
    suppress([[_,_,_],S],Pf, Pfi),
    insert((S,[F,H,G],Pere,A),Pui,Pu_new),
    insert([[F,H,G],S],Pfi,Pf_new).

% si la nouvelle valeur de F n'est pas meilleure, il n'y a rien à changer
comp_F(Old_F,Pf,Pu,_S,[F,_H,_G],_Pere,_A,Pf,Pu):-
    F >= Old_F.
```

```
% loop_successors(+Suc,+Pf,+Pu,+Q,+Pere,?Pf_out,?Pu_out).
% cas d'arrêt, pas de successeurs, Pf et Pu inchangés
loop_successors([], Pf, Pu, _Q, _Pere, Pf, Pu).

% situation déjà développée, on skip
loop_successors([(S,_A,[_F,_H,_G])|Suc],Pf,Pu,Q,Pere, Pf_out, Pu_out):-
    belongs([S,_Val,_Pere,_Coup],Q),
    loop_successors(Suc,Pf,Pu,Q,Pere, Pf_out, Pu_out).
```

```
% S est connu dans Pu, on le met à jour si le cout trouvé (pour F) est inférieur au pr
écédent
loop_successors([(S,A,[F,H,G])|Suc],Pf,Pu,Q,Pere, Pf_out, Pu_out):-
    belongs([S,[Old_F,_H,_G],_Pere,_A],Pu),
    comp_F(Old_F,Pf,Pu,S,[F,H,G],Pere,A,Pf_new,Pu_new),
    loop_successors(Suc,Pf_new,Pu_new,Q,Pere, Pf_out, Pu_out).

% S est une situation nouvelle, on insère dans Pu et Pf
loop_successors([(S,A,[F,H,G])|Suc],Pf,Pu,Q,Pere, Pf_out, Pu_out):-
    insert([S,[F,H,G],Pere,A],Pu,Pu_new),
    insert([[F,H,G],S],Pf,Pf_new),
    loop_successors(Suc,Pf_new,Pu_new,Q,Pere, Pf_out, Pu_out).
```

Le dernier prédicat développé (hormis le main, qui sera détaillé après) a été *aetoile*, en reprenant les 3 cas possibles :

1. Pf et Pu vides, c’est-à-dire qu’il ne reste plus d’états à développer

```
aetoile([],[],_):- print("PAS DE SOLUTION : L ETAT FINAL N EST PAS ATTEIGNABLE !").
```

2. L’élément min de Pf correspond à la situation finale (prédéfini)

```
aetoile(Pf,Pu,Q):-
  final_state(Fin),
  suppress_min([[F,H,G],Fin], Pf, _Pf2), % on regarde si le min de Pf est l'état Fin
  suppress([Fin,[F,H,G],Pere,A], Pu, _Pu2), % on récupère le Pere et le coup A
  insert([[[F,H,G],Fin],G,Pere,A], Q,Q_new), % on l'ajoute à Q et on affiche la soluti
on
  affiche_solution(Q_new).
```

3. Cas général, dans lequel on étudie l’élément min de Pf, on développe puis traite ses successeurs et l’on rappelle aetoile avec la nouvelle situation (les nouveaux Pf, Pu et Q)

```
aetoile(Pf,Pu,Qs):-
  suppress_min(U, Pf, Pf2), % M état de coût F minimal
    writeln(U),nl,
    U=[[F,H,G],S],
  suppress([S,[F,H,G],Pere,A], Pu, Pu2), % on enlève aussi M dans Pu (synchronisation)
    expand(U,Lsuc), % on trouve tous les successeurs
    loop_successors(Lsuc, Pf2, Pu2, Qs, U, Pf_out, Pu_out), % on traite les successeur
s
    insert([U,G,Pere,A], Qs,Q_new), % on ajoute l'état traité dans Q
    aetoile(Pf_out, Pu_out, Q_new). % on fait la récursion
```

Pour le main,

```
main :-
  initial_state(Ini),
  heuristique2(Ini,H0),
  G0 is 0, % on n'a rien fait pour l'instant
  F0 is G0 + H0,
  empty(Pf),
  empty(Pu),
  empty(Q),
  insert([[F0,H0,G0],Ini],Pf,Pfi), % on ajoute en parallèle dans Pf et dans Pu)
  insert([Ini,[F0,H0,G0], nil, nil],Pu,Pui),
  aetoile(Pfi, Pui, Q),

% Application

% avec la situation initiale initial_state([ [b, h, c], [a, f, d], [g,vide,e] ]).
?- main.

Élément min de Pf :
[[5,5,0],[[b,h,c],[a,f,d],[g,vide,e]]]
Successeurs dudit élément :
[([[b,h,c],[a,vide,d],[g,f,e]],up,[5,4,1]),([[b,h,c],[a,f,d],[vide,g,e]],left,[7,6,
1]),([[b,h,c],[a,f,d],[g,e,vide]],right,[7,6,1])]

Élément min de Pf :
[[5,4,1],[[b,h,c],[a,vide,d],[g,f,e]]]
Successeurs dudit élément :
[([[b,vide,c],[a,h,d],[g,f,e]],up,[5,3,2]),([[b,h,c],[a,f,d],[g,vide,e]],down,[7,5,
2]),([[b,h,c],[vide,a,d],[g,f,e]],left,[7,5,2]),([[b,h,c],[a,d,vide],[g,f,e]],right,
[7,5,2])]

Élément min de Pf :
[[5,3,2],[[b,vide,c],[a,h,d],[g,f,e]]]
Successeurs dudit élément :
[([[b,h,c],[a,vide,d],[g,f,e]],down,[7,4,3]),([[vide,b,c],[a,h,d],[g,f,e]],left,[5,2,
3]),([[b,c,vide],[a,h,d],[g,f,e]],right,[7,4,3])]

Élément min de Pf :
[[5,2,3],[[vide,b,c],[a,h,d],[g,f,e]]]
Successeurs dudit élément :
[([[a,b,c],[vide,h,d],[g,f,e]],down,[5,1,4]),([[b,vide,c],[a,h,d],[g,f,e]],right,[7,3,
4])]

Élément min de Pf :
[[5,1,4],[[a,b,c],[vide,h,d],[g,f,e]]]
Successeurs dudit élément :
[([[vide,b,c],[a,h,d],[g,f,e]],up,[7,2,5]),([[a,b,c],[g,h,d],[vide,f,e]],down,[7,2,
5]),([[a,b,c],[h,vide,d],[g,f,e]],right,[5,0,5])]

Solution, Q :
[[[5,0,5],[[a,b,c],[h,vide,d],[g,f,e]]],5,[[5,1,4],[[a,b,c],[vide,h,d],[g,f,e]]],righ
t]
[[[5,1,4],[[a,b,c],[vide,h,d],[g,f,e]]],4,[[5,2,3],[[vide,b,c],[a,h,d],[g,f,e]]],down]
[[[5,2,3],[[vide,b,c],[a,h,d],[g,f,e]]],3,[[5,3,2],[[b,vide,c],[a,h,d],[g,f,e]]],left]
[[[5,3,2],[[b,vide,c],[a,h,d],[g,f,e]]],2,[[5,4,1],[[b,h,c],[a,vide,d],[g,f,e]]],up]
```

```
[[[5,4,1],[[b,h,c],[a,vide,d],[g,f,e]]],1,[[5,5,0],[[b,h,c],[a,f,d],[g,vide,e]]],up]
[[[5,5,0],[[b,h,c],[a,f,d],[g,vide,e]]],0,nil,nil]
true .

% avec comme état initial l'état final (Fin)
?- main.

Élément min de Pf:
[[[0,0,0],[[a,b,c],[h,vide,d],[g,f,e]]],0,nil,nil]
Solution, Q :
[[[0,0,0],[[a,b,c],[h,vide,d],[g,f,e]]],0,nil,nil]
true .

% les autres tests ont été effectués dans la partie 3.3 ci-après.
```

3.3 - Analyse expérimentale

```
main :-
  initial_state(Ini),
  heuristique2(Ini,H0), % ou heuristique1a, selon le choix réalisé
  G0 is 0,
  F0 is G0 + H0,
  empty(Pf),
  empty(Pu),
  empty(Q),
  insert([[F0,H0,G0],Ini],Pf,Pfi),
  insert([Ini,[F0,H0,G0], nil, nil],Pu,Pui),
  statistics(walltime, [TimeSinceStart | [TimeSinceLastCall]]), %
  aetoile(Pfi, Pui, Q),
  statistics(walltime, [NewTimeSinceStart | [ExecutionTime]]), %
  write('Execution took '), write(ExecutionTime), write(' ms.'), nl. % pour l'affichag
e du temps de calcul de la requête aetoile(Pfi,Pui,Q)
```

Nous avons encadré le prédicat *aetoile* entre deux *statistics* qui vont nous permettre de calculer le temps d’éxécution de la requête en fonction du prédicat. Nous allons donc pouvoir comparer, pour chaque état initial proposé, l’heuristique la plus optimale.

```
initial_state([ [b, h, c],       % C'EST L'EXEMPLE PRIS EN COURS
                [a, f, d],       %
                [g,vide,e] ]).   % h1=4,   h2=5,   f*=5

%Heuristique1a  : Execution took 5 ms.
%Heurisitique2 : Execution took 4 ms.
```

```
initial_state([ [ a, b, c],
                 [ g, h, d],
```

```
                [vide,f, e] ]). % h2=2, f*=2
%Heuristique1a: Execution took 1 ms.
%Heuristique2: Execution took  1 ms
```

```
initial_state([ [b, c, d],
                [a,vide,g],
                [f, h, e]  ]). % h2=10 f*=10

%Heuristique1a: Execution took 21 ms.
%Heuristique2: Execution took  6 ms
```

```
 initial_state([ [e, f, g],
                [d,vide,h],
                [c, b, a]  ]). % h2=24, f*=30
% ERROR: Out of local stack
```

```
 initial_state([ [f, g, a],
                [h,vide,b],
                [d, c, e]  ]). % h2=16, f*=20

% No results but Astar with Manhattan distance (heuristique2) is much quicker
```

Au global, l'heuristique 2, utilisant les distances de Manhattan, est la plus intéressante. Elle est plus rapide au global, et plus fiable / représentative de la situation. Là où, pour l'heuristique 1a, il est question de nombre de pièces mal placées, mais rien ne dit qu'elles seront "simples" à remettre en place ou non. C'est donc, pour toutes les raisons précédentes, une heuristique moins optimale.

TP2

Question 1 - Familiarisation avec le TicTacToe 3x3

1.2

```
?- situation_initiale(S), joueur_initial(J).
S = [[_42, _48, _54], [_66, _72, _78], [_90, _96, _102]],
J = x.
% initialise le "tableau" du morpion (vide, cf. matrice 3x3 libre), et attribue à J le
"x"
?- situation_initiale(S), nth1(3,S,Lig), nth1(2,Lig,o).
```

```
S = [[_206, _212, _218], [_230, _236, _242], [_254, o, _266]],
Lig = [_254, o, _266].
% met un "o" dans le 2° élément (nth1(2,Lig,o)) de la 3° ligne de S (nth1(3,S,Lig))
% ne retourne pas de nouvelle matrice, une variable de S est juste fixée à présent
```

2.2 - Prédicat *alignement*

Trois types d'alignement possibles :

```
alignement(L, Matrix) :- ligne(L,Matrix).
alignement(C, Matrix) :- colonne(C,Matrix).
alignement(D, Matrix) :- diagonale(D,Matrix).
```

1. **Ligne**

```
ligne(L, M) :-
  nth1(_E,M,L).
% on utilise nth1 pour récupérer les éléments (_E) de la ligne L, dans la matrice M
```

2. **Colonne**
    a. on transpose la matrice du morpion et répète la fonction d'alignement des lignes (on sait déjà le faire et c'est + simple).

```
transpose([[]|_], []).
transpose(Matrix, [Row|Rows]) :- transpose_1st_col(Matrix, Row, RestMatrix),
                                 transpose(RestMatrix, Rows).
transpose_1st_col([], [], []).
transpose_1st_col([[H|T]|Rows], [H|Hs], [T|Ts]) :- transpose_1st_col(Rows, Hs, Ts).

colonne(C,M) :-
  transpose(M,Mt),
  ligne(C,Mt).
% transpose et transpose_1st_col pour transposer M, puis on fait appel à ligne
```

3. **Diagonale**
    a. "concept" opposé pour les deux diagonales *(K++ vs K--)*

```
diagonale(D, M) :- % première diagonale
  premiere_diag(1,D,M).
premiere_diag(_,[],[]).
premiere_diag(K,[E|D],[Ligne|M]) :-
  nth1(K,Ligne,E),
```

```
  K1 is K+1,
  premiere_diag(K1,D,M).


diagonale(D, M) :- % seconde diagonale
    seconde_diag(D,M).
seconde_diag(D, M) :- % on récupère la taille de la matrice (on décrémente, ici on che
rche le point de départ pour K)
  length(M,K),
  seconde_diag(K, D, M).
seconde_diag(_,[],[]).
seconde_diag(K,[E|D],[Ligne|M]) :-
  nth1(K,Ligne,E),
  K1 is K-1,
  seconde_diag(K1,D,M).

% test unitaire
% Le comportement présenté dans le sujet de TP est complet (on a bien 8 solutions, cf
 3 lignes, 3 colonnes et 2 diagonales)
?- M = [[a,b,c], [d,e,f], [g,h,i]], alignement(Ali, M).
M = [[a, b, c], [d, e, f], [g, h, i]], % ligne 1
Ali = [a, b, c] ;
M = [[a, b, c], [d, e, f], [g, h, i]], % ligne 2
Ali = [d, e, f] ;
M = [[a, b, c], [d, e, f], [g, h, i]], % ligne 3
Ali = [g, h, i] ;
M = [[a, b, c], [d, e, f], [g, h, i]], % colonne 1
Ali = [a, d, g] ;
M = [[a, b, c], [d, e, f], [g, h, i]], % colonne 2
Ali = [b, e, h] ;
M = [[a, b, c], [d, e, f], [g, h, i]], % colonne 3
Ali = [c, f, i] ;
M = [[a, b, c], [d, e, f], [g, h, i]], % première diagonale
Ali = [a, e, i] ;
M = [[a, b, c], [d, e, f], [g, h, i]], % seconde diagonale
Ali = [c, e, g] ;
false.
```

Prédicat *possible*

Pour réaliser ***possible***, on définit deux cas pour ***unifiable*** :

- L'élément est libre → var(X) → unifiable
- L'élément n'est pas libre → ground(X) → ~~unifiable~~, reste à vérifier si l'élément "appartient" à J (*x* ou *o*), auquel cas l'alignement est toujours possible

```
possible([X|L],J) :- unifiable(X,J), possible(L,J).
possible([],_).
unifiable(X,_) :- var(X).
unifiable(X,J) :- ground(X), X=J.

% tests unitaires
```

```
?- A=[_,_,_], possible(A,x). % yes (ligne libre)
A = [_8, _14, _20] ;
false.

?- A=[x,_,x], possible(A,x). % yes (2 x et 1 variable libre)
A = [x, _14, x] .

?- A=[_,o,x], possible(A,x). % no (présence de o)
false.

?- A=[x,x,x], possible(A,x). % yes (alignement gagnant même)
A = [x, x, x].

?- A=[o,o,o], possible(A,x). % no (alignement de o)
false.
```

Prédicats *alignement_gagnant* et *alignement_perdant*

```
% alignement_gagnant
alignement_gagnant(Ali, J) :- ground(Ali), possible(Ali,J). % l'alignement est ground
 (pas de variable libre) et possible (donc tous les éléments sont ceux de J)

% alignement_perdant
alignement_perdant(Ali, J) :- adversaire(J,J2), alignement_gagnant(Ali,J2). % aligneme
nt_perdant = alignement_gagnant de l'adversaire (cf. prédicat ci-dessus)


% tests unitaires
% alignement_gagnant (le principe est symétrique pour o)
?- A=[x,x,x], alignement_gagnant(A,x). % yes (3 x)
A = [x, x, x].

?- A=[x,x,_], alignement_gagnant(A,x). % no (1 variable libre)
false.

?- A=[o,x,o], alignement_gagnant(A,x). % no (présence de o)
false.

?- A=[o,o,o], alignement_gagnant(A,x). % no (gagnant pour o)
false.

?- A=[_,_,_], alignement_gagnant(A,_). % no (variables libres)
false.

% alignement_perdant
?- A=[o,o,o], alignement_perdant(A,x). % yes (alignement de o)
A = [o, o, o].

?- A=[o,o,o], alignement_perdant(A,o). % no (alignement_gagnant)
false.

?- A=[o,x,o], alignement_perdant(A,x). % no (présence de x)
false.
```

```
?- A=[o,_,o], alignement_perdant(A,x). % no (présence de o mais variable libre)
false.

?- A=[_,_,_], alignement_perdant(A,x). % no (variables libres)
false.
```

Question 2 - Développement de l’heuristique

```
heuristique(J,Situation,H) :-   % cas 1, alignement gagnant pour J
   H = 10000,       % grand nombre approximant +infini
   alignement(Alig,Situation),
   alignement_gagnant(Alig,J), !. % cut pour ne pas rentrer dans le cas 3

heuristique(J,Situation,H) :-   % cas 2, alignement perdant pour J
   H = -10000,        % grand nombre approximant -infini
   alignement(Alig,Situation),
   alignement_perdant(Alig,J), !. % cut pour ne pas rentrer dans le cas 3

heuristique(J,Situation,H) :- % cas 3, alignement ni gagnant ni perdant
    % coups possibles pour J
    findall(Alig_g,(alignement(Alig_g,Situation),possible(Alig_g,J)),Lg),
    length(Lg,Cg),
    % coups possibles pour l'adversaire
    adversaire(J,J2),
    findall(Alig_p,(alignement(Alig_p,Situation),possible(Alig_p,J2)),Lp),
    length(Lp,Cp),
    % H = nb de coups possibles pour J - nb de coups possibles pour J2
    H is Cg-Cp.

% tests unitaires
% situation initiale, autant de coups possibles pour les deux donc H=0
?- situation_initiale(S), heuristique(J,S,H).
S = [[_56, _62, _68], [_80, _86, _92], [_104, _110, _116]],
J = x,
H = 0 ;
S = [[_56, _62, _68], [_80, _86, _92], [_104, _110, _116]],
J = o,
H = 0.

% alignement perdant
?- situation(S_perdante), J=x, heuristique(J,S_perdante,H).
S_perdante = [[o, _912, a], [o, b, _942], [o, _960, _966]],
J = x,
H = -10000.

% alignement gagnant
?- situation(S_gagnante), J=o, heuristique(J,S_gagnante,H).
S_gagnante = [[o, _920, a], [o, b, _950], [o, _968, _974]],
J = o,
H = 10000.

% aucune case libre, alignement gagnant
?- S_null=[[x,o,x],[o,x,o],[x,o,x]], J=x, heuristique(J,S_null,H).
```

```
S_null = [[x, o, x], [o, x, o], [x, o, x]],
J = x,
H = 10000.

% aucune case libre et aucun alignement gagnant / perdant
?- S_null=[[x,o,x],[o,x,o],[o,x,o]], J=x, heuristique(J,S_null,H).
S_null = [[x, o, x], [o, x, o], [o, x, o]],
J = x,
H = 0.

% J pose "x", il a plus de coups possibles que son adversaire, situation avantageuse H
>0
?- S_null=[[_,_,_],[_,x,_],[_,_,_]], J=x, heuristique(J,S_null,H).
S_null = [[_402, _408, _414], [_426, x, _438], [_450, _456, _462]],
J = x,
H = 4.

% Adversaire pose "o" au premier tour, J a moins de coups possible, H négatif (désavan
tageux)
?- S_null=[[_,_,_],[_,o,_],[_,_,_]], J=x, heuristique(J,S_null,H).
S_null = [[_402, _408, _414], [_426, o, _438], [_450, _456, _462]],
J = x,
H = -4.
```

Question 3 - Développement de l’algorithme Négamax

3.2

“*Quel prédicat permet de connaître sous forme de liste l’ensemble des couples [Coord,Situation_Resultante] tels que chaque élément (couple) associe le coup d’un joueur et la situation qui en résulte à partir d’une situation donnée.”*

Il s’agit du prédicat *successeurs(J,Etat,Succ)*

```
?- situation_initiale(S), joueur_initial(J), successeurs(J,S,Succ).
S = [[_276, _282, _288], [_300, _306, _312], [_324, _330, _336]],
J = x,
Succ = [[[1, 1], [[x, _1322, _1328], [_1340, _1346, _1352], [_1364, _1370, _1376]]],
 [[1, 2], [[_1214, x, _1226], [_1238, _1244, _1250], [_1262, _1268|...]]], [[1, 3],
 [[_1112, _1118, x], [_1136, _1142|...], [_1160|...]]], [[2, 1], [[_1010, _1016|...],
 [x|...], [...|...]]], [[2, 2], [[_908|...], [...|...]|...]], [[2, 3], [[...|...]
|...]], [[3|...], [...|...]], [[...|...]|...], [...|...]].
% Succ = liste des coups possibles, avec la matrice résultante, depuis la situation de
départ = 9 coups possibles (toutes les cases sont libres)
```

3.3 - Développement des prédicats de négamax

- Il a d’abord fallu développer le prédicat *successeur* dans “tictactoe.pl”

```
successeur(J, Etat,[L,C]) :- nth1(L,Etat,Lig), nth1(C,Lig,J).

% tests unitaires
?- joueur_initial(J), situation_initiale(S), successeur(J,S, [2,2]). % on joue en [2,
2] comme premier coup, matrice résultante S = [[_,_,_],[_,x,_],[_,_,_]]
J = x,
S = [[_158, _164, _170], [_182, x, _194], [_206, _212, _218]].

?- joueur_initial(J), situation_initiale(S), successeur(J,S, [L,C]). % on ne précise p
as [L,C], donc accepte l'ensemble des coups possibles(9).
J = x,
S = [[x, _192, _198], [_210, _216, _222], [_234, _240, _246]],
L = C, C = 1 ;
J = x,
S = [[_186, x, _198], [_210, _216, _222], [_234, _240, _246]],
L = 1,
C = 2 ;
J = x,
S = [[_186, _192, x], [_210, _216, _222], [_234, _240, _246]],
L = 1,
C = 3 ;
J = x,
S = [[_186, _192, _198], [x, _216, _222], [_234, _240, _246]],
L = 2,
C = 1 ;
J = x,
S = [[_186, _192, _198], [_210, x, _222], [_234, _240, _246]],
L = C, C = 2 ;
J = x,
S = [[_186, _192, _198], [_210, _216, x], [_234, _240, _246]],
L = 2,
C = 3 ;
J = x,
S = [[_186, _192, _198], [_210, _216, _222], [x, _240, _246]],
L = 3,
C = 1 ;
J = x,
S = [[_186, _192, _198], [_210, _216, _222], [_234, x, _246]],
L = 3,
C = 2 ;
J = x,
S = [[_186, _192, _198], [_210, _216, _222], [_234, _240, x]],
L = C, C = 3.

?- joueur_initial(J), S=[[o,_,_],[_,_,_],[_,_,_]], successeur(J,S, [1,1]). % on ne peu
t pas écrire si la variable n'est pas libre
false.
```

- Ensuite, le prédicat *meilleur* pour renvoyer le coup le plus avantageux parmi tous ceux possibles (retournés par le prédicat *successeurs*, cf. développement du prédicat *successeur* précédemment

```
meilleur([X],X). % si on n'a qu'un coup possible, c'est forcément le meilleur
meilleur([[CX,VX]|Liste_Couples], Meilleur_Couple):-
  Liste_Couples\=[],
  meilleur(Liste_Couples,[CY,VY]),
  (VX<VY-> Meilleur_Couple=[CX,VX]; Meilleur_Couple=[CY,VY]). % on garde le coup dont
 l'heursitique est la plus basse
```

- Les trois cas possibles pour le prédicat *negamax* sont :
  - La profondeur de l'arbre est atteinte, on renvoie l'heuristique pour l'état actuel.

```
negamax(J, Etat, P, P, [Coup, Val]) :-
  Coup = rien,
  heuristique(J,Etat,Val).
```

  - Le tableau est complet, on renvoie l'heuristique de l'état actuel. (en d'autres termes, J ne peut pas jouer)

```
negamax(J, Etat, _P, _Pmax, [Coup, Val]) :-
  situation_terminale(J,Etat),
  Coup = rien,
  heuristique(J,Etat,Val).
```

  - On calcule les successeurs de l'état actuel et on appelle *negamax* récursivement pour chacun de ces successeurs. On renvoie le minimum des valeurs négatives renvoyées. (i.e le tableau n'est pas complet et la profondeur maximale n'est pas atteinte)

```
negamax(J, Etat, P, Pmax, [Coup, (Val)]) :-
  P=<Pmax,
  successeurs(J,Etat,Succ),
  loop_negamax(J,P,Pmax,Succ,Liste_Couples),
  meilleur(Liste_Couples,[Coup,V2]),
  Val is -V2.
```

- Enfin, le prédicat main sera appelé et exécutera l'algorithme *negamax* en retournant C et V le coup optimal et sa valeur d'heuristique associée. Pmax peut aller de 0 à 9.

```
main(C,V, Pmax) :-
  Pmax=<9,
  joueur_initial(J), % main considéré pour le joueur x
  situation_initiale(S), % main sur un état initial (tableau vide)
  P is 0, % on commence à une profondeur nulle
  negamax(J, S, P, Pmax, [C, V]).
```

Question 4 - Expérimentation et extensions

4.1 - Meilleur coup pour des profondeurs de 1 à 9

```
% tests pour différentes valeurs de pmax
?- main(C,V,1). % tps d'exécution : NaN
C = [2, 2],
V = 4 .
% valeur de V ok, en jouant au centre on peut faire 1 ligne, 1 colonne, et les 2 diago
nales = 4 alignements possibles, 0 pour l'adversaire (pas de o).

?- main(C,V,2). % tps d'excécution : NaN
C = [2, 2],
V = 1 .
% intuitivement, le meilleur coup pour J2 serait un coin, avec 2 alignements possible
s, et 3 (une diagonale de perdue) pour J, soit V=1 semble correct.

?- main(C,V,3). % tps d'exécution : NaN
C = [2, 2],
V = 3 .

?- main(C,V,4). % tps d'exécution : 0.5s
C = [2, 2],
V = 1 .

?- main(C,V,5). % tps d'exécution : 2s
C = [2, 2],
V = 3 .

?- main(C,V,6). % tps d'exécution : 11s
C = [2, 2],
V = 1 .

?- main(C,V,7). % 7 et +, out of global stack. Programme exécuté sur un ordinateur per
sonnel, il aurait fallu le tester sur un autre. Une solution pour réduire le tps / poi
ds de calcul, serait d'utiliser la symétrie du morpion (e.g, dans un tableau vide, jou
er en [1,1], [1,3], [3,1] et [3,3] (les coins) est équivalent).
ERROR: Out of global stack
```

Dans tous les cas de profondeur, le meilleur coup à jouer est en [2,2]. C’est le seul coup qui permet de gagner à coup sûr. Pour une profondeur maximale de 9, si le

calcul est possible (*cf. pas Out of global stack*), le résultat devrait être [2,2] car à nouveau, c'est le seul qui permet à coup sûr de gagner ou de faire match nul.

Question dans le fichier negamax.pl

```
/*
A FAIRE : commenter chaque litteral de la 2eme clause de loop_negamax/5,
  en particulier la forme du terme [_,Vsuiv] dans le dernier
  litteral ?
*/
```

- J → Joueur pour lequel on exécute négamax (le meilleur coup possible pour lui sera retourné)
- P → Profondeur actuelle (entre 1 et 9 au morpion),
- Pmax → Profondeur maximale de l'arbre (9 au morpion = nb d'éléments du tableau).
- [Coup, Suiv] → Premier élément de *Succ* (ci-dessous) = [coup joué, état du tableau en conséquence]
- Succ → liste des successeurs [Coup, Etat_Suivant], retourné par le prédicat *successeurs* depuis l'état actuel
- Reste_Couples → Reste des [Coup, heuristique associée]
- A → Adversaire de J
- Pnew → Profondeur pour l'itération suivante de negamax (P + 1)
- [_, Vsuiv] → Vsuiv = valeur de l'heuristique associée au coup joué. "_" car l'association se fait directement dans l'appel de loop_negamax ([Coup,Suiv] et [Coup,Vsuiv])

```
% Pour l'exécution de main(C,V,1), on affiche [Coup,Suiv] et Vsuiv. On a bien Vsuiv =
 heuristique associée au Coup (e.g en [2,2], Vsuiv=-4 (pour l'adversaire, 4 pour J)
[[3,3],[[_852,_858,_864],[_876,_882,_888],[_900,_906,x]]]
-3
[[3,2],[[_954,_960,_966],[_978,_984,_990],[_1002,x,_1014]]]
-2
[[3,1],[[_1056,_1062,_1068],[_1080,_1086,_1092],[x,_1110,_1116]]]
-3
[[2,3],[[_1158,_1164,_1170],[_1182,_1188,x],[_1206,_1212,_1218]]]
-2
[[2,2],[[_1260,_1266,_1272],[_1284,x,_1296],[_1308,_1314,_1320]]]
```

```
-4
[[2,1],[[_1362,_1368,_1374],[x,_1392,_1398],[_1410,_1416,_1422]]]
-2
[[1,3],[[_1464,_1470,x],[_1488,_1494,_1500],[_1512,_1518,_1524]]]
-3
[[1,2],[[_1566,x,_1578],[_1590,_1596,_1602],[_1614,_1620,_1626]]]
-2
[[1,1],[[x,_1674,_1680],[_1692,_1698,_1704],[_1716,_1722,_1728]]]
-3
```

4.2 - Symétrie

Évoqué dans le 3.3

4.3 - Puissance 4

Pour développer un algorithme pour le puissance 4, il faudrait prendre en compte un tableau de taille différente (6x7), et les règles du jeu :

- On ne peut que remplir par les colonnes (la ligne sera la première case non remplie)
- Victoire par alignement de 4 jetons en ligne, colonne ou diagonale (similaire morpion)
    - Attention, cela ne correspond pas à une ligne / colonne complète (pas comme au morpion)
    - Modification de l'heuristique

4.4 - Alpha-Beta

L'idée de l'algorithme alpha-bêta serait de reprendre l'algorithme negamax en ajoutant deux paramètres (alpha et beta)

- alpha : conserve la valeur pour le *Joueur* → max(J)
- beta : conserve la valeur pour l'*Adversaire* → *min(A)*

Ces deux valeurs seront conservées pendant la récursion de l'arbre de recherche. Si on est en dehors de [alpha, beta], on peut *skip* la branche, c'est-à-dire ne plus l'étudier, car la solution optimale ne viendra pas de ladite branche.

Nous n'avons pas mené le développement de cet algorithme à son terme (updatealphabeta par exemple), mais une idée pour *alphabeta.*

Le fichier alphabeta.pl est disponible sur le Git. Le code n’étant pas abouti mais simplement une ébauche, nous ne le détaillerons pas ici.