```
/*
Version : 1.02 , 24/09/2022
Version : 1.01 , 01/06/2020
Version : 1.00 , 10/10/2019

Auteur : Ric

Langage de programmation : PROLOG

Solution au problème de la traversée d'une rivière de M missionnaires et M cannibales dociles mais opportunistes
à l'aide d'un bateau d'une capacité C
La règle est simple : ils peuvent faire autant d'allers-retours qu'ils estiment nécessaires, MAIS les missionnaires ne doivent
JAMAIS se retrouver en infériorité numérique (bateau ou rives) sinon MIAM !!!

Une fois appelé SWI-PROLOG en ligne de commande avec swipl sous linux (à adapter si autre OS), une fois ce fichier
(nom_ficier.pl) compilé
avec swi-prolog par l'intermédiaire de consult(nom_fichier).
lancer en ligne de commande l'instruction jeu(M,C). où M est le nombre de missionnaires et C la capacité du bateau

La configuration la plus connue est jeu(3,2).

Une solution (liste des trajets entre les deux rives) est proposée et tous les tests ont validé à ce jour ce programme
Bien entendu, l'erreur, la malchance ou l'ignorance du développeur étant humaine :
- ce programme peut contenir des bugs mineurs ou fatals
- ce programme peut ne pas fonctionner de façon inattendue ou à cause de contraintes liées à la version courante de PROLOG
- ce programme peut être très mal conçu
- ce programme peut proposer des solutions non optimales

Vous êtes donc libre :
- d'utiliser ou non ce programme
- de récupérer ce programme et de le modifier
- de me faire part de votre version améliorée
*/

% déclaration dynamique afin de pouvoir les supprimer et les recréer
:- dynamic nb_missionnaire/1.
:- dynamic capacite_bateau/1.

% limite arbitraire imposée par précaution afin d'éviter des temps de calcul infinis dans certains cas
nb_max_missionnaire(15).

% valeurs par défaut
nb_missionnaire(3).
capacite_bateau(2).

/*
```

```
état : [nombre de missionnaires, nombre de cannibales, rive courante]
*/

% état initial
debut([N_m, N_m, 1]) :- nb_missionnaire(N_m), nb_max_missionnaire(M), N_m =< M.

% états finaux
fin([N_m, N_m, -1]) :- debut([N_m, N_m, 1]).

% définition des rives, rive de départ : 1
inverser(1, -1).
inverser(-1, 1).
sens_parcours(1, "->").
sens_parcours(-1, "<-").

/*
pour les trajets partant de la rive d'arrivée, on remplit le bateau le moins possible
*/
valeur(Rive, _, _, I, I) :- Rive < 0, !.
/*
pour les trajets partant de la rive de départ, on remplit le bateau le plus possible
*/
valeur(_, Borne_inf, Borne_sup, I, J) :- J is Borne_sup + Borne_inf - I, !.

% transition entre états
transition([NB_m_old, NB_c_old, Rive_old], [NB_m, NB_c, Rive], Transit) :-
capacite_bateau(N), nb_missionnaire(M),
Borne1 is min(NB_m_old, N), Borne2 is min(NB_c_old, N),
inverser(Rive_old, Rive),
sens_parcours(Rive_old, Sens),

% génération d'équipages pour le bateau
between(0, Borne1, I), valeur(Rive_old, 0, Borne1, I, NB_m_bateau),
between(0, Borne2, J), valeur(Rive_old, 0, Borne2, J, NB_c_bateau),

% contrôles des effectifs du bateau
(NB_m_bateau =:= 0; NB_m_bateau >= NB_c_bateau),
NB_sur_bateau is NB_m_bateau + NB_c_bateau,
NB_sur_bateau >= 1, NB_sur_bateau =< N,

% contrôles de la rive au départ du transit
NB_m_old_new is NB_m_old - NB_m_bateau,
NB_c_old_new is NB_c_old - NB_c_bateau,
(NB_m_old_new =:= 0; NB_m_old_new >= NB_c_old_new),

% contrôles de la rive à l'arrivée du transit
```

```
NB_m is M - NB_m_old + NB_m_bateau,
NB_c is M - NB_c_old + NB_c_bateau,
(NB_m =:= 0; NB_m >= NB_c),

(
Rive_old >= 0 ->
 swritef(Transit, "G(%w M ; %w C) %w B( %w M ; %w C) ? %w D : G(%w M ; %w C) -------------[B] D(%w M ; %w C)",
  [NB_m_old, NB_c_old, Sens, NB_m_bateau, NB_c_bateau, Sens, NB_m_old_new, NB_c_old_new, NB_m, NB_c])
  ;
 swritef(Transit, "G %w B( %w M ; %w C) ? %w D(%w M ; %w C) : G(%w M ; %w C) [B]------------- D(%w M ; %w C)",
  [Sens, NB_m_bateau, NB_c_bateau, Sens, NB_m_old, NB_c_old, NB_m, NB_c, NB_m_old_new, NB_c_old_new])
).


jeu(N_m, C_b) :-
N_m >= 1,
C_b >= 1,
% MAJ des valeurs par défaut
retractall(nb_missionnaire(_)), assert(nb_missionnaire(N_m)),
retractall(capacite_bateau(_)), assert(capacite_bateau(C_b)),

debut(Etat),
fsa(Etat, [], []).


%================ test de fin
fsa(Etat, Chemin, Liste_transit) :-
fin(Etat),
append([Etat], Chemin, Chemin_bis),
reverse(Chemin_bis, Chemin_complet),
reverse(Liste_transit, Liste_transit_bis),
writeln(Chemin_complet),
length(Liste_transit_bis, I), swritef(S, "Nombre de trajets : %w", [I]), writeln(S),
maplist(writeln ,Liste_transit_bis).

%================ fabrication récursive de la liste
fsa(Etat0, Chemin, Liste_transit) :-
transition(Etat0, Etat, Transit),
\+ member(Etat, Chemin),
fsa(Etat, [Etat0|Chemin], [Transit|Liste_transit]),
!.
% utilisation du cut pour ne récupérer que la première solution
```