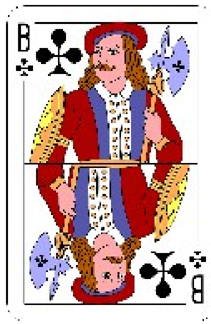
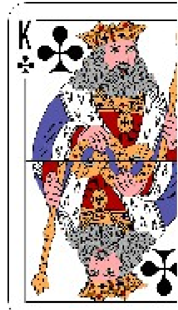
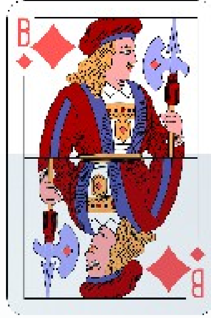
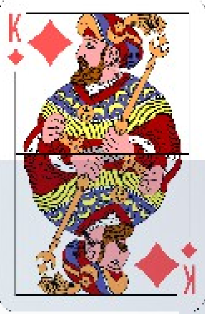
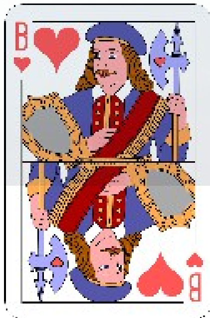
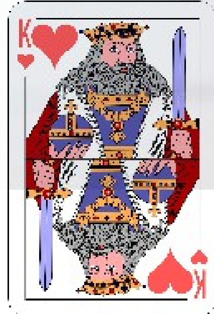
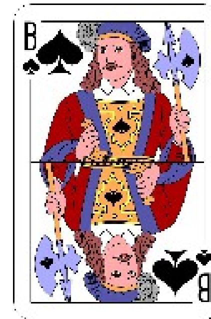
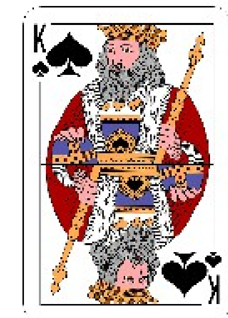



Berge Droite

Berge Gauche

Transport de Rois et Valets

Le problème

Deux berges sont séparées par une rivière, entre elles une barque (à rames : elle ne traverse pas vide) à 2 places

Sur la berge gauche, quatre couples Roi-Valet désirent aller sur la berge droite, le problème est compliqué par l'animosité de Rois : chaque Roi tuera les valets des autres rois dès qu'ils ne seront plus protégés par leur Roi

L'objectif est de dire si OUI ou NON ce transport est possible, accessoirement on souhaitera organiser le transport avec le nombre minimum de trajets de la barque

Ce problème va être noté RV4-B2

Des variantes

Deux berges : peut être remplacé par 3 îles ou 5 villes ... voire n emplacements

Quatre couples Roi-Valet : peut être remplacé par 2 ... 5 (voire m) couples (R,V)

Les couples (R,V) : peuvent devenir des familles Roi-Dame-Valet voire avec enfants

Les animosités des Rois pour les Valets des autres peuvent êtres changées pour d'autres contraintes

Le transport par barque (qui sous entend : on peut aller de droite à gauche comme de gauche à droite) peut être remplacé par des chemins monodirectionnels (un graphe orienté général), la barque à 2 places sera parfois changée pour une à 3 places

Pour commencer

La formulation "Roi-Valet" incite à essayer avec des cartes à jouer, mais le bon vieux papier-crayon (gomme) va aussi bien (même mieux : il a de la mémoire)

Deux Roi-Valet et une barque à 2 places devrait vous prendre 3 minutes (PAS 3

secondes)

Trois Roi-Valet   et une barque à 2 places devrait vous prendre … un bon moment

Ce qui va être fait

D'abord on analyse et programme RV4-B2

Ensuite : le corrigé de RV3-B2 (si vous n'aviez pas trouvé)

Puis RV4-B3

Les programmes en Python font le travail mais ne permettent pas de comprendre, la seule conclusion générale que j'ai obtenu est : "on n'y comprend rien"

Ci-dessous :
les paragraphes de texte, exemples, commentaires sont en noir, le code Python est bleu

Le cas RV4 - B2

Les données

Les individus

Les rois et valets pourraient être 0,1,2... mais il va être plus clair d'avoir :

```
gens = [["r",1],["v",1],["r",2],["v",2],["r",3],["v",3],["r",4],["v",4]]
```

et on manipulera aussi leurs codes :

```
codesgens = [0,1,2,3,4,5,6,7]
```

selon les besoins on fera la navette entre gens et codesgens

Les animosités entre DEUX personnes vont être traitées par la fonction :

```
def compatibles2(i,j):
   if (gens[i][0] == gens[j][0]) : return 1  # deux R ou deux V sont compatibles
   elif (gens[i][1] == gens[j][1]): return 1  # deux de la même famille 1..4 vont ensemble
   else: return 0                              # sinon : 0 dit  "incompatibles"
```

Cette fonctions sera facilement adaptable aux cas avec des Dames ou avec un "grand nombre de familles" (la seule difficulté est de savoir ce que l'on veut)

Des utilitaires

Les fonctions qui suivent "doivent exister quelque part" dans des bibliothéques Python, mais j'ai trouvé plus rapide (c'est mon choix) de refaire que de fouiller

```
def prefixe(a,liste): return [a]+liste #  Exemple : prefixe(8,[1,5,9]) = [8, 1, 5, 9]
```

parties : les parties de l'ensemble (liste supposée sans répétitions)
parties2 : les parties à 2 éléments

```
def parties(liste):
    if len(liste) == 0: return []
    elif len(liste) == 1: return [[],liste]
    else: return parties(liste[1:])+[prefixe(liste[0],x) for x in parties(liste[1:])]
```

Exemple : parties([5,1,"u"]) = [[], ['u'], [1], [1, 'u'], [5], [5, 'u'], [5, 1], [5, 1, 'u']]

```
def parties2(liste):
    if len(liste) <= 1: return []
    elif len(liste) == 2: return [liste]
    else: return parties2(liste[1:])+[[liste[0],x] for x in liste[1:]]
# Exemple : parties2([5,1,"u"]) = [[1, 'u'], [5, 1], [5, 'u']]
```

filtre et mapG : "mes" restes de programmation antérieure (caml, Mathematica, Maple) il doit y avoir des moyens de faire ça autrement en Python

```
def filtre(test,liste):
    if len(liste) == 0: return []
    elif test(liste[0]): return [liste[0]]+filtre(test,liste[1:])
    else: return filtre(test,liste[1:])
```

Exemple

```
def mul5(x):
    if x%5==0:return 1
    else : return 0
```

filtre(mul5,[1,10,100,23]) renvoit [10, 100]

```
def mapG(f,liste):
    if len(liste) == 0: return []
    else: return [f(liste[0])]+mapG(f,liste[1:])
```

Exemple : mapG(mul5,[1,2,5,9]) vaut [0, 0, 1, 0]

```
parties2Gens = parties2([0,1,2,3,4,5,6,7])
```

il y en a 28 pour la barque (il restera à voir la compatibilité)

```
solitaires = [[x] for x in codegens]
```

(les solitaires aussi peuvent prendre la Barque)

encore un utilitaire qui doit préexister

```
def appartient(x,liste):
   if len(liste) == 0: return 0
   elif liste[0] == x: return 1
   else: return appartient(x,liste[1:])
```

Les tests de compatibilité pour le passage d'un équipage de Barque

Parmi les 256 parties on cherche lesquelles sont compatibles avec les Rois, il doit y avoir moyen de faire plus court que ce qui suit, mais je me trompais sans cesse, d'où ces fonctions "longues"

```
def compatibleR1(liste):
   if appartient(0,liste) and appartient(3,liste) and not(appartient(2,liste)): return 0
   elif appartient(0,liste) and appartient(5,liste) and (not(appartient(4,liste))): return 0
   elif appartient(0,liste) and appartient(7,liste) and (not(appartient(6,liste))): return 0
   else:  return 1
```

La première ligne dit "si R1 (c'est 0) est là, et V2 (c'est 3) et pas R2 (c'est 2) alors 0" à savoir : non compatible avec R1.

On écrit de même compatibleR2, compatibleR3 et compatibleR4, enfin :

```
def survitR(liste): # on fait les quatre tests
   return filtre(compatibleR4,filtre(compatibleR3,filtre(compatibleR2,filtre(compatibleR1,liste))))
```

```
Gauche = survitR(toustas)      # il y a 96 groupes de départ à priori  possibles
bonsB = filtre(compListe2,parties2Gens)+solitaires  # bons pour la Barque (il en reste 24)
Droite = survitR(toustas)              # les mêmes 96
```

Maintenant, pour chaque groupe X de Gauche, on va voir

1°) si son complémentaire Y survit

2°) si l'un des 24 bonsB que je nomme C

   a) est contenu dans cet X

   b) laisse (à son départ dans B) un X\C qui survit sur la berge Gauche

   c) et produit un YuC qui survit sur la berge Droite

Le 1°) : la survie de Y

```
def complementaire(liste):  # c'est ici que Y est calculé depuis X
   vide = []
   for x in codesgens:
      if (not x in liste):
         vide = vide+[x]
   return vide

def compatibleR(liste):
```

```
    return (compatibleR1(liste) and compatibleR2(liste) and compatibleR3(liste) and
compatibleR4(liste))

def survitC(x):
        return compatibleR(complementaire(x))

Gauche2 = filtre(survitC,Gauche)  # il n'y en a plus plus que 46
Droite2 = filtre(survitC,Gauche)
```

Le 2°) a) : quels C de bonsB sont contenus dans mon X

Désormais je trouve plus commode de coder mes parties par leurs applications caractéristiques

```
zero = [0, 0, 0, 0, 0, 0, 0, 0]    # zero code la partie vide

Tous = [1, 1, 1, 1, 1, 1, 1, 1]

def CodeOfPartie(liste):  # Exemple : CodeOfPartie([1,5]) vaut [0, 1, 0, 0, 0, 1, 0, 0]
    nul = [0, 0, 0, 0, 0, 0, 0, 0]
    for x in liste: nul[x] = 1
    return nul

def PartieOfCode(c):  # Exemple : PartieOfCode(zero) vaut []
    nul=[]
    for k in range(len(c)):
        if c[k]==1: nul=nul+[k]
    return nul

def CardinalCode(liste): # il y a sûrement une fonction qui donne la somme des termes d'une liste ...
    if len(liste) == 0: return 0
    elif liste[0] == 1: return (1+CardinalCode(liste[1:]))
    else: return CardinalCode(liste[1:])
```

avec ce codage l'intersection c'est juste le produit

```
def InterCode(a, b):
    nul = [0, 0, 0, 0, 0, 0, 0, 0]
    for k in codesgens: nul[k]=a[k]*b[k]
    return nul
```

avec ce codage l'inclusion c'est juste 'inférieur ou égal' qui passe tout seul aux listes :

```
def InclusCode(a, b):
    return a <= b

def ListePossibles(g, liste): # un échantillon juste après
```

```
    if len(liste) == 0: return []
    elif InclusCode(liste[0], g): return [liste[0]]+ListePossibles(g, liste[1:])
    else: return ListePossibles(g, liste[1:])
```

ListePossibles(CodeOfPartie(Gauche2[24]), mapG(CodeOfPartie, bonsB))
[[0, 0, 0, 0, 0, 0, 1, 1], [0, 0, 0, 0, 1, 0, 1, 0], [0, 0, 1, 0, 1, 0, 0, 0], [0, 0, 1, 0, 0, 0, 1, 0], [1, 0, 1, 0, 0, 0, 0, 0], [1, 0, 0, 0, 1, 0, 0, 0],
[1, 0, 0, 0, 0, 0, 1, 0], [1, 0, 0, 0, 0, 0, 0, 0], [0, 0, 1, 0, 0, 0, 0, 0], [0, 0, 0, 0, 1, 0, 0, 0], [0, 0, 0, 0, 0, 0, 1, 0], [0, 0, 0, 0, 0, 0, 0, 1]]

Le 2°) b) : il doit rester une partie qui soit compatibleR

```
def retire(c, x):
    y = [0, 0, 0, 0, 0, 0, 0, 0]
    for k in range(len(c)):
        if c[k] == 1: y[k]=0
        else: y[k]=x[k]
    return y
```

Exemple : retire([0,1,0,1,0,1,0,1],[1,1,0,1,1,1,1,1]) vaut [1, 0, 0, 0, 1, 0, 1, 0]
la syntaxe Tous - x ne marche pas

```
def ListePossibles2(g):
    bons = []
    a_tester = ListePossibles(g, mapG(CodeOfPartie, bonsB))
    for z in a_tester:
        if compatibleR(PartieOfCode(retire(z, g))):
            bons = bons+[z]
    return bons
```

Exemple : ListePossibles2([1, 0, 1, 0, 1, 0, 1, 1]) vaut
[[0, 0, 0, 0, 0, 0, 1, 1], [0, 0, 1, 0, 1, 0, 0, 0], [1, 0, 1, 0, 0, 0, 0, 0],
[1, 0, 0, 0, 1, 0, 0, 0], [1, 0, 0, 0, 0, 0, 0, 0], [0, 0, 1, 0, 0, 0, 0, 0],
[0, 0, 0, 0, 1, 0, 0, 0], [0, 0, 0, 0, 0, 0, 0, 1]]

Le 2°) c) : les voyageurs et l'autre côté doivent être compatibleR

```
def complementaire(x): # on aurait pu utiliser 'retire'
    z = zero
    for k in range(len(x)):
        z[k] =1-x[k]
    return z
```

```
def union(x,y): # il doit bien y avoir un truc comme 'max'
    z = [0,0,0,0,0,0,0,0]
    for k in range(len(x)):
        if (x[k] or y[k]): z[k] = 1
    return z

def ListePossibles3(g):
    bons = []
    a_tester = ListePossibles2(g)
    for z in a_tester:
        if compatibleR(PartieOfCode(union(z, complementaire(g)))):
            bons = bons+[z]
    return bons
```

ListePossibles3([1, 0, 1, 0, 1, 0, 1, 1]) ... l'essai est bon

```
CodesGauche2 = [CodeOfPartie(x) for x in Gauche2] # Gauche2 : les 46 restant
CG2 = CodesGauche2 # juste un alias
```

le dernier est CG2[45] qui vaut [1,1,1,1,1,1,1,1] = le départ

ListePossibles3(CG2[45]) : il y en a 14 : quatre Valets seuls, quatre R+V, six paires de Valets

Les gm sont des [Groupe,Message] :

la partie Groupe est le code d'un sous ensemble de codegens

la partie message est la liste des occupations antérieures de la barque (c'est empilé avec le passé en queue et le dernier vu en tête)

VoyageGD(gm) : gm est UN (Groupe,Message) du côté Gauche, on va calculer ses successeurs du côté droit

initialisation : CG2[45] = [1, 1, 1, 1, 1, 1, 1, 1] ("ils sont tous là")

le message est [], il n'y a pas encore eu de passage de la Barque

```
CoteG0 = [[CG2[45], []]]
VoyageGD(CoteG0[1]) # donne tous les gm successeurs possibles de de CG2[45]
```

L'éxécution donne :

VoyageGD(CoteG0[0]) =
[[[0, 0, 0, 0, 0, 0, 1, 1], [[6, 7]]], [[0, 0, 0, 0, 0, 1, 0, 1], [[5, 7]]],
[[0, 0, 0, 0, 1, 1, 0, 0], [[4, 5]]], [[0, 0, 0, 1, 0, 1, 0, 0], [[3, 5]]],
[[0, 0, 0, 1, 0, 0, 0, 1], [[3, 7]]], [[0, 0, 1, 1, 0, 0, 0, 0], [[2, 3]]],
[[0, 1, 0, 1, 0, 0, 0, 0], [[1, 3]]], [[0, 1, 0, 0, 0, 1, 0, 0], [[1, 5]]],
[[0, 1, 0, 0, 0, 0, 0, 1], [[1, 7]]], [[1, 1, 0, 0, 0, 0, 0, 0], [[0, 1]]],
[[0, 1, 0, 0, 0, 0, 0, 0], [[1]]], [[0, 0, 0, 1, 0, 0, 0, 0], [[3]]],
[[0, 0, 0, 0, 0, 1, 0, 0], [[5]]], [[0, 0, 0, 0, 0, 0, 0, 1], [[7]]]]

```
CoteD1 = VoyageGD(CoteG0[0])  # on donne un nom à ce qui précède
```

On s'occuppe de l'enchaînement des passages

Maintenant il faut de la mémoire : pour éviter de recommencer ce que l'on a déjà fait

Pour cela on crée deux "mémoires" : les positions (possibles ou non) à G et D
dès qu'on en aura traité une on la qualifiera de déjavu
La numérotation se fait 'par numération en base 2' : voir la fonction NumOfCode ci-dessous

```
def NumOfCode(c):
   puiss2 = [1, 2, 4, 8, 16, 32, 64, 128]
   somme = 0
   for k in range(8):
      somme = somme+puiss2[k]*c[k]
   return somme

def FaitNul(n):    # servira à initialiser les mémoires
   Nul = []
   for k in range(n):
      Nul = Nul+[0]
   return Nul
```

CGDV (CotéGaucheDéjaVu) est une liste de booléens disant si un état gauche a déjà été vu
CDDV (CotéDroitDéjaVu) est une liste de booléens disant si un état droit a déjà été vu

```
CGDV = FaitNul(256) # initialisation
CDDV = FaitNul(256)
```

On enregistre la position de départ

```
CGDV[NumOfCode([1, 1, 1, 1, 1, 1, 1, 1])] = 1  # c'est bien enregistré
```

VoyageDG traite UN des termes de CD1 (CôtéDroit1) qui sont des dm =[d,m] : droite,message
de même pour VoyageGD

Ci-dessous un rapport d'hésitions :
Si je met CGDV et CDDV en variables globales : il faut tout réévaluer depuis le début à chaque fois
Si je met CGDV et CDDV en dernier champ des "d" ou "g" : cela me fera une couche de [et] de plus
Comme :
1) *mon nombre de positions possibles est "faible" (46)*
2) *on a (au moins) une position nouvelle à chaque itération*

je suis sûr que mes calculs vont être "courts", et j'utilise donc 2 variables globales : CGDV et CDDV initialisées à [[],[],.....[]] et qui vont progressivement se remplir

```
def EnregistreD(position):
   CDDV[NumOfCode(position)] = 1

def EnregistreG(position):
   CGDV[NumOfCode(position)] = 1

def VoyageDG(undm):  # "undm" désigne UN Droite Message"
   bons = []
   a_tester = ListePossibles2(undm[0])
   for z in a_tester:
      zg = union(z, complementaire(undm[0]))
      if compatibleR(PartieOfCode(zg)) and not(CGDV[NumOfCode(zg)]):
         bons = bons+[[zg, undm[1]+[PartieOfCode(z)]]]
         EnregistreG(zg)
   return bons

def VoyageGD(ungm):   # "ungm" désigne UN Gauche Message"
   bons = []
   a_tester = ListePossibles2(ungm[0])
   for z in a_tester:
      zg = union(z, complementaire(ungm[0]))
      if compatibleR(PartieOfCode(zg)) and not(CDDV[NumOfCode(zg)]):
         bons = bons+[[zg, ungm[1]+[PartieOfCode(z)]]]
         EnregistreD(zg)
   return bons
```

Exemple : VoyageDG(CD1[0]) vaut [[[1, 1, 1, 1, 1, 1, 1, 0], [[6, 7], [6]]]]

```
def ToutVoyageDG(d):  # on va appliquer VoyageDG à tout un côté droit
   if len(d) == 0: return []
   elif len(d) == 1: return VoyageDG(d[0])
   else: return VoyageDG(d[0])+ToutVoyageDG(d[1:])

def ToutVoyageGD(g):   # on va appliquer VoyageGD à tout un côté gauche
   if len(g) == 0: return []
   elif len(g) == 1: return VoyageGD(g[0])
   else: return VoyageGD(g[0])+ToutVoyageGD(g[1:])

def ToutVoyageGDG(g): # tout un Aller-Retour
     return ToutVoyageDG(ToutVoyageGD(g))
```

Maintenant il n'y a plus qu'à exécuter :

CD1=ToutVoyageGD(CG0)
CG2=ToutVoyageDG(CD1)
CD3=ToutVoyageGD(CG2)
CG4=ToutVoyageDG(CD3)
CD5=ToutVoyageGD(CG4)
CG6=ToutVoyageDG(CD5)

CD7 ... <u>ne marche plus :</u> conclusion "RV4 B2" n'a pas de solution

On vérifie en scrutant CG6 :

CG6 vaut [[[1, 1, 1, 1, 0, 0, 1, 1], [[6, 7], [6], [3, 5], [3], [4, 6], [6, 7]]],
[[1, 1, 1, 1, 1, 1, 0, 0], [[6, 7], [6], [3, 5], [3], [4, 6], [4, 5]]],
[[1, 1, 1, 0, 1, 0, 1, 0], [[6, 7], [6], [3, 5], [3], [1, 3], [1]]],
[[1, 0, 1, 1, 1, 0, 1, 0], [[6, 7], [6], [3, 5], [3], [1, 3], [3]]],
[[1, 0, 1, 0, 1, 1, 1, 0], [[6, 7], [6], [3, 5], [3], [1, 3], [5]]],
[[1, 0, 1, 0, 1, 0, 1, 1], [[6, 7], [6], [3, 5], [3], [1, 3], [7]]],
[[1, 1, 0, 0, 1, 1, 1, 1], [[6, 7], [6], [3, 5], [5], [2, 6], [6, 7]]],
[[0, 0, 1, 1, 1, 1, 1, 1], [[6, 7], [6], [1, 3], [3], [0, 6], [6, 7]]]]

Tous les successeurs des termes de CG6 sont des cas de Droite déjaVus, il n'y a plus rien de neuf à faire

Le cas RV3 - B2

Solution de RV3 B2

R1 V1 R2 V2 R3 V3		
	→ V2 V3 →	
		V2 V3
	← V3 ←	
R1 V1 R2 R3 V3		
	→ V1 V3 →	
		V1 V2 V3
	← V1 ←	
R1 V1 R2 R3		
	→ R2 R3 →	
		R2 V2 R3 V3
	← R2 V2 ←	
R1 V1 R2 V2		
	→ R1 R2 →	
		R1 R2 R3 V3
	← V3 ←	
V1 V2 V3		
	→ V1 V2 →	
		R1 V1 R2 V2 R3
	← V1 ←	
V1 V3		
	→ V1 V3 →	
		R1 V1 R2 V2 R3 V3

Le cas RV4 - B3

Il n'y a rien à changer … sauf la taille de la barque

La fonction parties2 (les parties à 2 éléments) doit être remplacée par parties3 … alors on écrit carrément une fonction partp qui donne les parties à p éléments (celle là je me suis

beaucoup trompé en l'écrivant)

```
def partp(liste,p):
    if p==0: return [[]]
    elif  len(liste) <= p-1: return []
    elif  len(liste) == p: return [liste]
    else: return partp(liste[1:],p)+[[liste[0]]+x for x in partp(liste[1:],(p-1))]
```

Le résultat final est différent : on peut !! la fin qui avait échoué dans le cas RV4-B2 est remplacée par :

```
      CG2=ToutVoyageDG(CD1)
      CD3=ToutVoyageGD(CG2)
      CG4=ToutVoyageDG(CD3)
      CD5=ToutVoyageGD(CG4)
      CG6=ToutVoyageDG(CD5)
      CD7=ToutVoyageGD(CG6)
      CG8=ToutVoyageDG(CD7)
      CD9=ToutVoyageGD(CG8)
      CD9 contient :
[[[1, 1, 1, 1, 1, 1, 1, 1], [[3, 5, 7], [5, 7], [1, 5, 7], [1], [2, 4, 6], [6, 7], [0,
6], [3], [1, 3, 7]]]]
```

Le cas RV5 - B3

Le résultat final est le même : on peut !! Cette fois-ci c'est CD11 (CôtéDroit 11) qui le premier contient tout le monde

Le cas RV6 - B3

Programmation sans changement ... le résultat final n'est pas le même : on ne peut pas !! C'est au douzième aller-retour que … on n'a plus rien, nos Rois et Valets ne peuvent aller que vers des positions déjà vues

Divers 0

Je ne sais plus d'où vien ce problème … je sais que c'est le premier programme que j'ai écrit en Python

Divers 1

Si jamais quelqu'un était bloqué : solution de RV2-B2, des voyages sont

V1,V2 V1 R1,R2 R1 R1,V1

Divers 2

Un cas général 'trop facile' : RVn-B4 il est évident que c'est faisable, mais minimiser le nombre de voyages n'est pas clair : les programmes précédents le font, si donc on a la patience de programmer et exécuter RV6-B4, RV8-B4, RV10-B4, RV12-B4, RV14-B4 … on doit bien arriver à trouver l'idée

Divers 3

J'ai essayé de prouver des généralités : quand RVp-Bq est possible alors RV(p-1)-Bq l'est aussi, cela semble bien évident, mais je n'y suis pas arrivé

Ce qui reste à faire

Il reste à en faire un "jeu dynamique" : les divers R et V sont sur une berge, à la souris on les met dans la Barque, à la souris on envoie la Barque de l'autre côté … si un Valet se fait tuer : on a perdu

Défaut de ce jeu : dans les cas 'impossible' on va bien perdre son temps

Remède : après avoir choisi le graphe (Exemple : un triangle de 3 îles avec deux barques monodirectionelles et une bidirectionelle sur le troisième côté, 4 familles de (R,D,V) avec des choix d'assassinats), on dit au programme de jouer seul 15 (par exemple) fois puis de nous montrer la situation atteinte.

Le joueur aura alors à reproduire cela (ou à tout ramener en place comme dans le cas du Rubik's cube)

pour l'instant … je ne sais pas faire, merci aux lecteurs savants pour leurs conseils : Pygame ? Tkinter ?

daniel.goffinet42@gmail.com