Modélisation et Résolution du Puissance 4

Problématique : L'élaboration d'une intelligence artificielle permettant de remporter la partie dans un maximum de situations passe par la recherche des contraintes à imposer pour effectivement faire pencher la partie en sa faveur. Il s'agit donc de trouver progressivement, par l'expérience, les contraintes qui seront efficaces dans la pratique pour le Puissance 4.

Sommaire

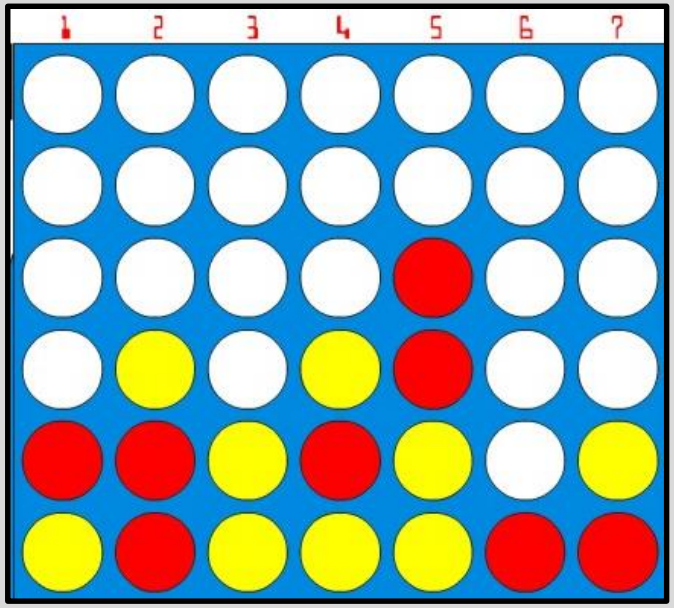
I/ Principe du jeu

- *Support :* Tableau de taille 6x7

- *Pièces utilisées :* 42 jetons,
21 pour chaque joueur

- *Déroulement :* les 2 joueurs
placent chacun leur tour un jeton
dans une des colonnes disponibles.

- *But :* Aligner 4 jetons



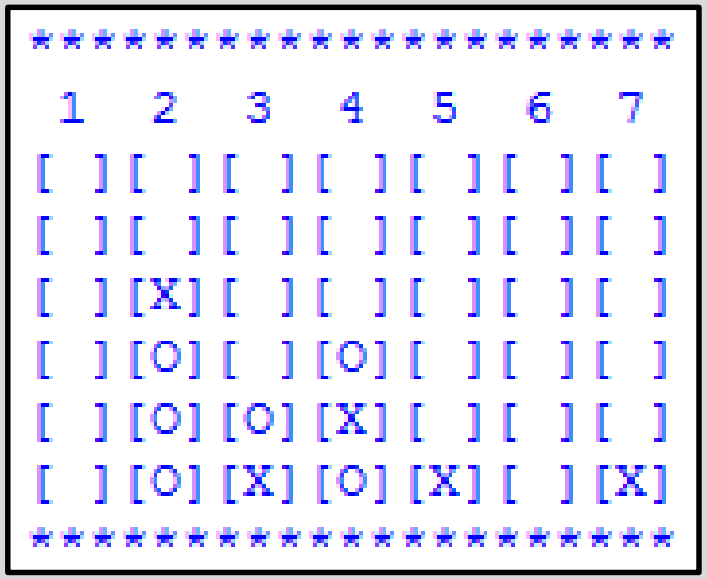
II/ Modélisation

1) Choix sur le code

- Utilisation de Python
- Algorithme utilisable par tout le monde (principalement pour les tests) en contrôlant les entrées des différents inputs.

2) Affichage

- Jetons différentiés par la forme plutôt que la couleur
- Tableau représenté par une liste de listes de chaines de caractères (‘[ ]’, ‘[X]’, ‘[O]’ selon l’état de la case)



3) Structure de l'algorithme

a) Algorithme principal

- Lancer la simulation du jeu sur plusieurs parties.
- Un seul argument n égal au nombre de personnes jouant au jeu (0, 1 ou 2)

- L'initialisation globale :
  - scores mis à 0
  - joueurs affectés
  - IA enregistrée (pour n<2)

```
def jeu(n):
    txt=1
    n=str(n)
    while n not in ["0","1","2"]:
        n=input("Il y a erreur. Entrez le nombre de joueurs (0,1 ou 2)
    n=int(n)
    sa=0
    sb=0
    continuer=1
    if n==2:
        ja=input("Nom du premier joueur? ")
        jb=input("Nom du second joueur? ")
```

- L'initialisation faite pour chaque partie :
  - tableau réinitialisé
  - autres variables remises à 0 (nombre de tours, nombre de cases occupées)

```
while continuer==1:
    if n==2:
        j1=input("Qui commence? ")
    elif n==1:
        j1=input("Qui commence? (Entrez votre nom ou bien ORDI) ")
    else:
        j1=random.choice([ja,jb])
    while j1!=ja and j1!=jb:
        j1=input("Ce n'est le nom d'aucun des 2 joueurs. Veuillez entrez qui co
    if j1==ja:
        j2=jb
    else:
        j2=ja
    sl=[]
    for i in range(0,6):
        sl.append('[ ]')
    l=[]
    for i in range(0,7):
        l.append(list(sl))
    x=0
    tour=1
    print(j1,"commence la partie")
```

- Déroulement des tours successifs :
  - enregistrement de la colonne choisie
  - vérification de la disponibilité de la colonne puis ajout du jeton.

```
print("Tour",tour,": C'est à",j1,"de jouer (jetons O)")
if (j1=="ORDI" and n==1) or n==0:
    if n==0 and j1=="ORDI_B":
        ordil=ordi(l,"[O]",txt,difb)
    else:
        ordil=ordi(l,"[O]",txt,difa)
    l=ordil[0]
    print(j1,"a choisi de jouer dans la colonne",ordil[1],":")
else:
    a=input("Entrez le n° de la colonne sur laquelle vous voulez jouer:
    while a not in L:
       a=input("Il y a erreur. Entrez le n° de la colonne sur laquelle
    while l[int(a)-1][0]!="[ ]":
       a=input("Cette colonne est remplie, veuillez en choisir une autr
       while a not in L:
          a=input("Il y a erreur. Entrez le n° de la colonne sur laquel
    a=int(a)
    l=ajouter(l,a,"[O]")
x=x+1
```

  - vérification du nombre maximum de jetons alignés

b) Fonctions de vérification

-> Fonctions à 2 arguments (tableau : L ; jeton : t) faites dans le but de calculer la longueur de la plus grande ligne du tableau.

- Lignées verticales :

```
def vérifcolonne(l,t):
    p=1
    for j in range(0,7):
        sp=1
        pc=1
        for i in range(0,5):
            if l[j][i]==l[j][i+1]==t:
                sp=sp+1
                if sp>pc:
                    pc=sp
            else:
                sp=1
        if pc>p:
            p=pc
    return(p)
```

- Lignées horizontales :

```
def vérifligne(l,t):
    p=1
    for i in range(0,6):
        sp=1
        pl=1
        for j in range(0,6):
            if l[j][i]==l[j+1][i]==t:
                sp=sp+1
                if sp>pl:
                    pl=sp
            else:
                sp=1
        if pl>p:
            p=pl
    return(p)
```

Lignées diagonales:

Diagonales descendantes :

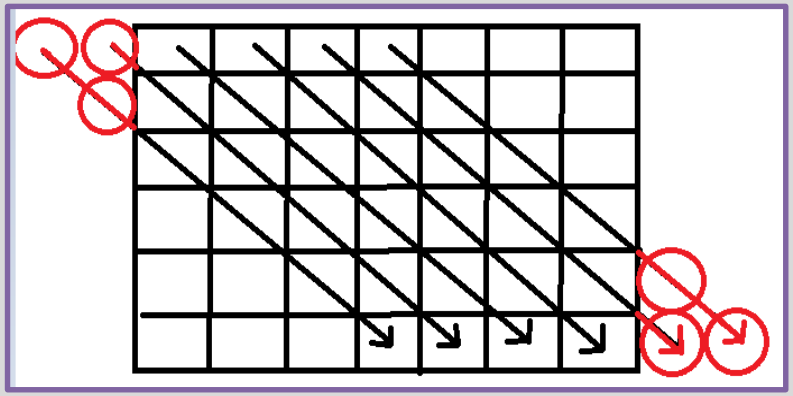
```
def vérifdiagonale_desc(l,t):
    p=1
    for k in range(0,6):
        sp=1
        pd=1
        i=0
        for j in range(k-2,k+3):
            if j in [0,1,2,3,4,5]:
                if l[j][i]==l[j+1][i+1]==t:
                    sp=sp+1
                    if sp>pd:
                        pd=sp
                else:
                    sp=1
            i=i+1
        if pd>p:
            p=pd
    return(p)
```

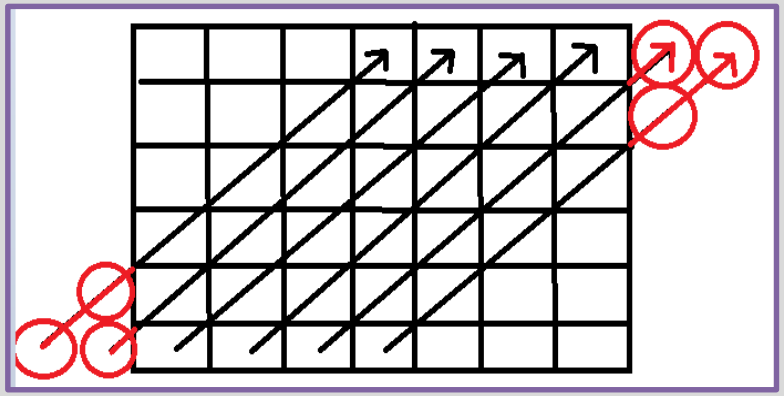
Diagonales montantes :

```
def vérifdiagonale_mont(l,t):
    p=1
    for k in range(0,6):
        sp=1
        pd=1
        i=5
        for j in range(k-2,k+3):
            if j in [0,1,2,3,4,5]:
                if l[j][i]==l[j+1][i-1]==t:
                    sp=sp+1
                    if sp>pd:
                        pd=sp
                else:
                    sp=1
            i=i-1
        if pd>p:
            p=pd
    return(p)
```

```
def maxi(l,t):
    return(max(vérifcolonne(l,t),vérifligne(l,t),vérifdiagonale_desc(l,t),vérifdiagonale_mont(l,t)))
```

Parcours schématisé de la vérification des cases (coins non vérifiés) :





- Calcul du maximum de chaque direction :

```
def maxi(l,t):
    return(max(vérifcolonne(l,t),vérifligne(l,t),vérifdiagonale_desc(l,t),vérifdiagonale_mont(l,t)))
```

III/ Résolution

Objectif : Parer les potentielles menaces que va faire apparaître l'adversaire (1) et en imposer le plus possible à l'autre (2)

1) Méthode Rétrograde

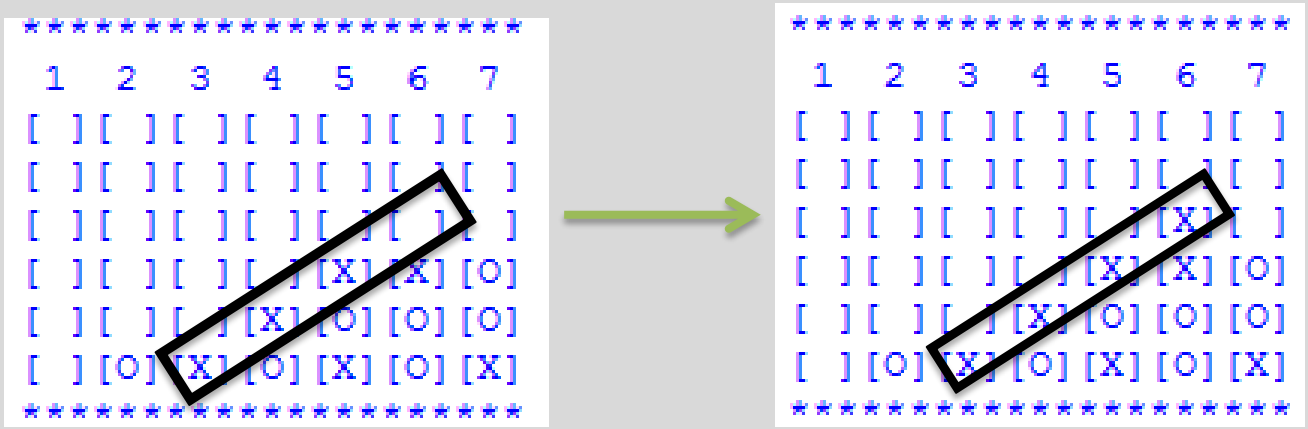
a) Réactions élémentaires par ordre de priorité (tours n-1)

- Compléter toutes les potentielles lignes de 4 :

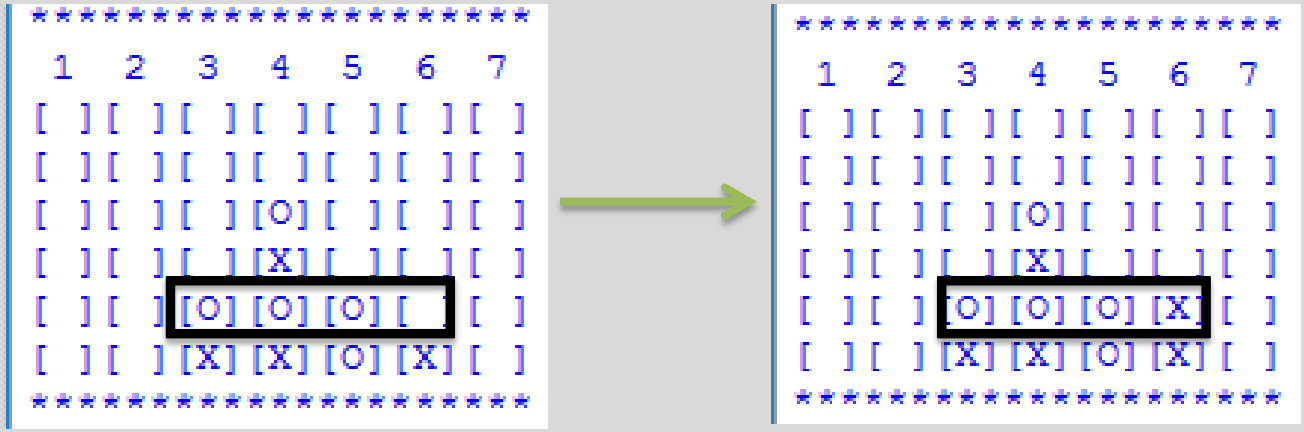
→ Pour savoir où compléter la ligne, utilisation de la fonction maxi sur le tableau auquel on ajoute un jeton sur chaque colonne, séparément :

```
for c in listcolnonpleines(l):
    if maxi(ajouter(l,c,t),t)>=4:
        listecoljouables=[c]
```

- Compléter toutes les potentielles lignes de 4 :



- Bloquer toutes les potentielles lignes de 4 de l'adversaire au tour suivant :



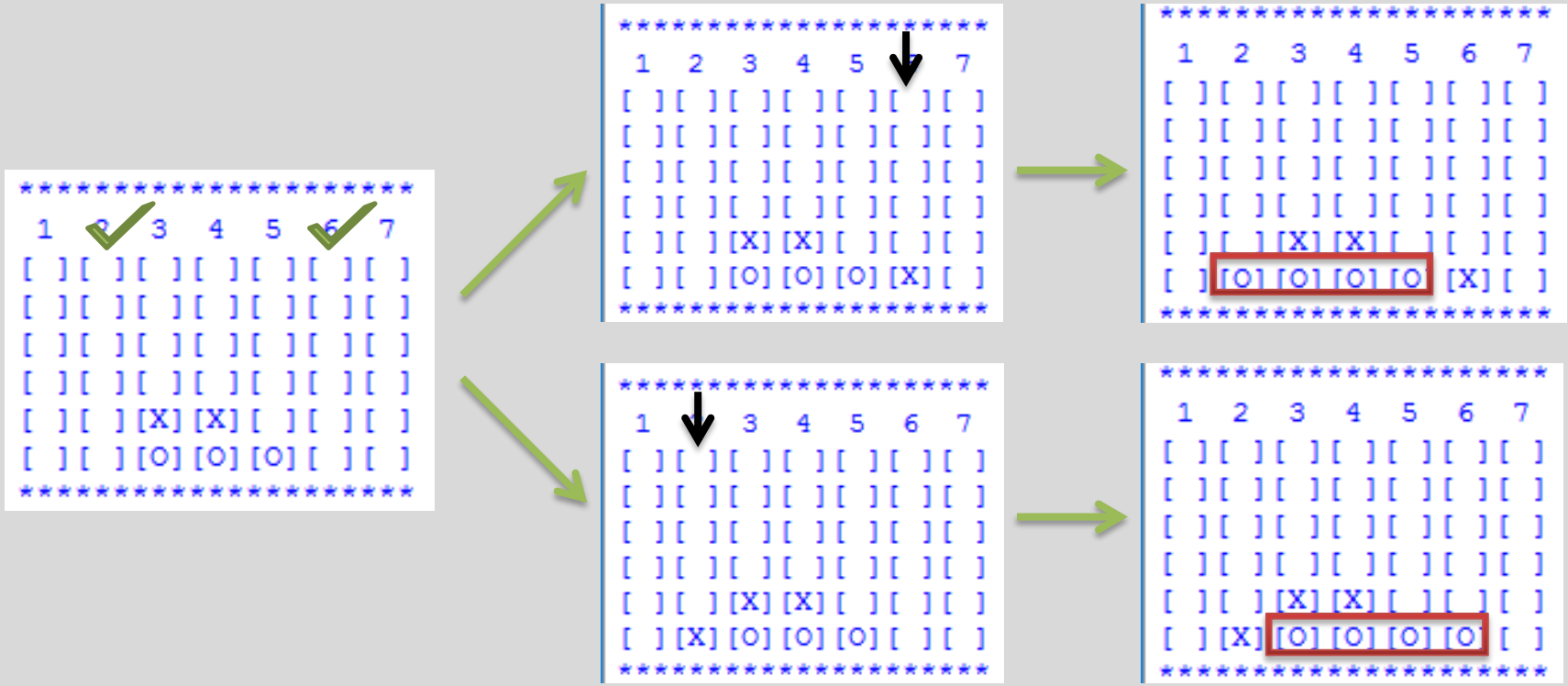
- Exclusion des colonnes rendant accessible une ligne de 4 pour l'adversaire :


```
def listcolok(l,lcj,t):
    listnew=[]
    for c in lcj:                 # on vérifie pour chaque colonne,
        lt=ajouter(l,c,t)         # après avoir ajouté notre jeton
        lti=ajouter(lt,c,inv(t))# puis celui de l'adversaire,
        if maxi(lti,inv(t))<4:   # que le tableau ne comporte pas de lignes de 4 pour l'adversaire
            listnew.append(c)
    return(listnew)
```

b) Réactions aux menaces multiples (tours n-2)

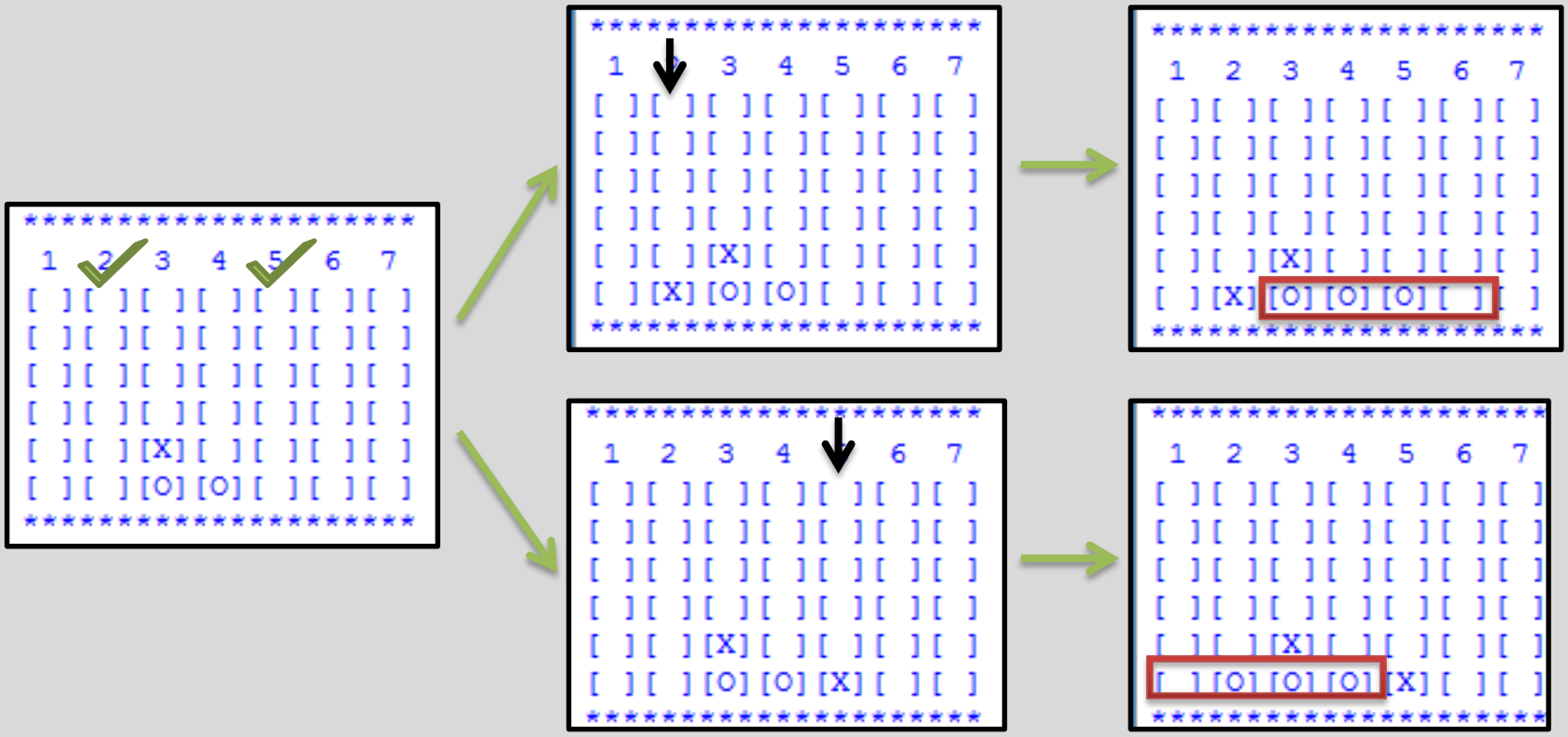
- Apparition de « pièges » , nécessité de réagir un tour plus tôt.

Ex de piège sur une seule ligne :



→ On ne peut donc pas contrer un piège au tour n-1

Réaction au tour n-2 :

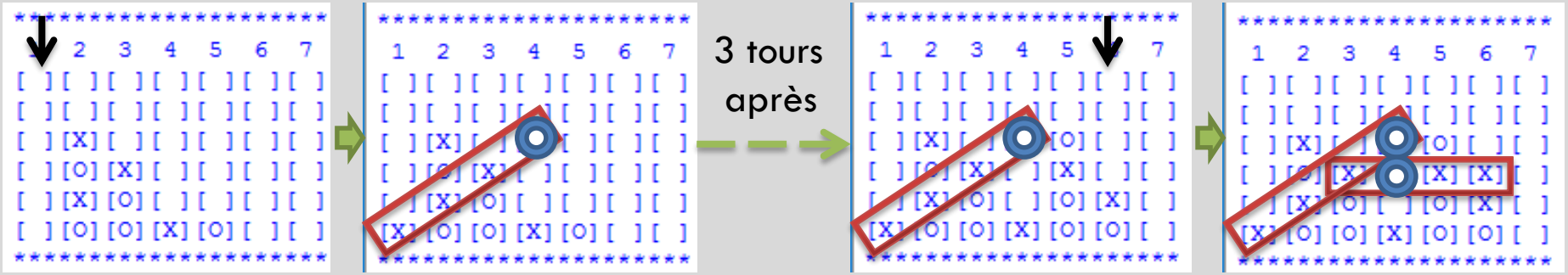


→ On arrive donc au tour n-1 dans une situation avec 1 seule menace.

Les pièges sont soit :

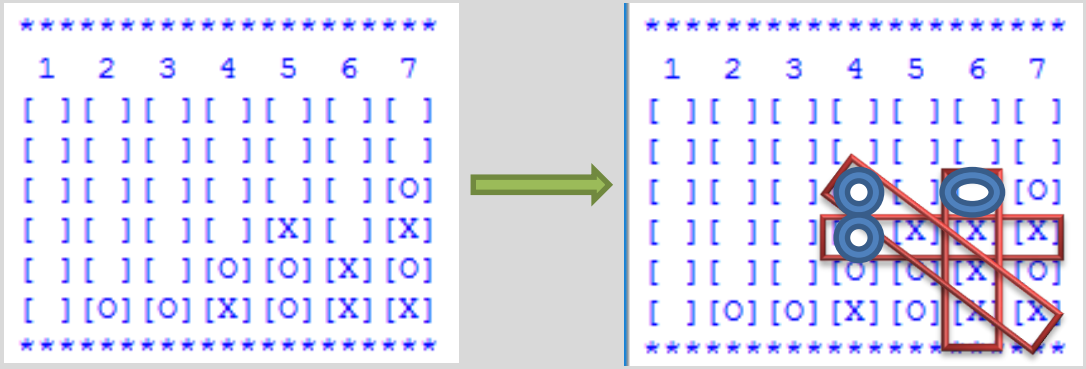
- 2 ou + menaces placées sur plusieurs tours à des emplacements consécutifs encore inaccessibles sur une même colonne :

Exemple de 2 menaces consécutives inaccessibles placées à 3 tours d'intervalles



- 2 ou + menaces placées en un seul tour à des emplacements accessibles :

Exemple ci-dessous d'une triple menace formée par des lignes croisées



- Fonction piege qui localise les pièges accessibles :

```
def piegecol(l,c,t,txt,g):
    jm=len(l)
    lt=ajouter(l,c,t)
    lti=ajouter(lt,c,inv(t))
    li=ajouter(l,c,inv(t))
    for c1 in range(1,jm+1):
        lii=ajouter(li,c1,inv(t))
        if maxi(lii,inv(t),g)>=g and (c1!=c or ("col" not in maxidirec(lii,inv(t),g)[1])):
            lit=ajouter(li,c1,t)
            for c2 in range(1,jm+1):
                liti=ajouter(lit,c2,inv(t))
                if maxi(liti,inv(t),g)>=g:  #on regarde s'il existe un couple de colonnes (c1,c2)
                    return(True,c1,c2)      #telles que l'adversaire gagne dans les 2 colonnes à la
    return(False,0,0)                       #fois en ayant joué dans la colonne c le tour d'avant.
```

```
def piege(l,t,txt,g):
    listecoljouables=[]
    for c in listcolnonpleines(l):   #Pour chaque colonne,
        pgc=piegecol(l,c,t,txt,g)
        li=ajouter(l,c,inv(t))
        lt=ajouter(l,c,t)
        lti=ajouter(lt,c,inv(t))
        if pgc[0]==True and maxi(lti,inv(t),g)<g and maxi(li,inv(t),g)<g: #On vérifie s'il y a un piege
            (c1,c2)=(pgc[1],pgc[2])
            if txt==1:
                print("piege en",c,c1,c2)
            if c!=c1 or c!=c2:
                if maxixaccessible(ajouter(l,c,t),inv(t),g)<g-1:
                    listecoljouables=[c]
                elif maxixaccessible(ajouter(l,c1,t),inv(t),g)<g-1:
                    listecoljouables=[c1]
                else:
                    listecoljouables=[c2] #dans ce cas, la colonne c est la colonne à jouer (absolument)
    return(listecoljouables)
```

- Fonction piege2 pour les pieges non accessibles :

```
def piege2(l,t,txt,g):
    im=len(l[0])
    listecoljouables=[]
    for c in listcolnonpleines(l):        #Pour chaque colonne potientellement jouée par l'adversaire,
        li=ajouter(l,c,inv(t))
        for cf in listcolnonpleines(li): #On cherche une colonne où figure au moins 2 menaces consécutives
            nbmenaces = 0
            nbmax = 0
            i=0
            while support(li,[cf-1],[i])==False and i<im:
                if maxixloc(li,i,cf-1,inv(t),g)>=g-1:
                    nbmenaces = nbmenaces + 1
                    if nbmax < nbmenaces:
                        nbmax = nbmenaces
                else:
                    nbmenaces = 0
                i=i+1
            if nbmax>=2:                  #Et dans ce cas on ajoute c aux colonnes à jouer (absolument)
                print("piege2 en",c)
                listecoljouables.append(c)
                break
    return(listecoljouables)
```

- Problèmes rencontrés

A partir de l'étape n-3 :

- Vérifications très nombreuses et rend les temps d'exécution sur plusieurs parties très longs.

- Difficultés à cerner l'ensemble des situations nécessitant 3 tours pour pouvoir contrer une potentielle ligne de l'adversaire.

→ Méthode rétrograde temporairement laissée de côté pour chercher une autre manière de construire une IA efficace

2) Méthode par sélection « offensive »

But : Compléter l'IA en cherchant une manière efficace de jouer lorsqu'il n'y a pas de menaces pour faire pencher la partie en sa faveur.

a) En choisissant les colonnes donnant la lignée la plus grande

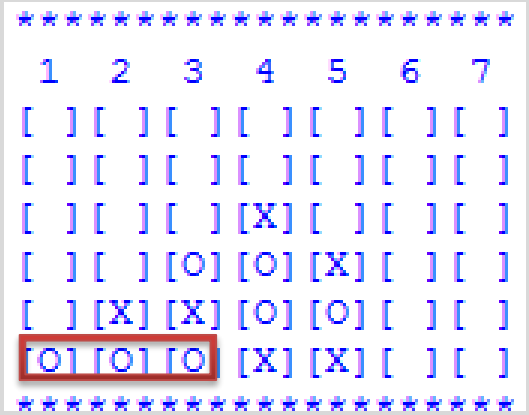
```
def slc1(l,lcj,t,txt,g):
    m=0
    lcjnew=list(lcj)
    for c in lcj:
        lt=ajouter(l,c,t)
        lti=ajouter(lt,c,inv(t))
        maxit=maxi(lt,t,g)
        maxixt=maxix(lt,t,g)
        maxiti=maxi(lti,inv(t),g)
        if txt==1:
            print(c,maxit,maxixt)
        if maxit>m and maxiti<g:
            lcjnew=[c]
            m=maxit
        elif maxit==m and maxiti<g and c not in lcjnew:
            lcjnew.append(c)
    return(lcjnew)
```

Calcul de la longueur maximale **maxit** des lignes du tableau auquel on a rajouté un jeton.

- m est la longueur maximale de ligne que l'on peut former au tour suivant.

Problème : slc1 utilise la fonction maxi qui prend en compte **toutes** les lignes.

Exemple de situtation où slc1 est inefficace :



Ici, **maxit** = 3 pour toutes les colonnes donc slc1 les renverra toutes dans **lcjnew** sans avoir rien selectionné.

c) En choisissant les colonnes donnant des lignes potentielles plus longues

- Une ligne peut être entrecoupée par des cases vides, mais pas par des cases adverses.
- Au contact d'au moins une cases vide.

- Création dans un 1er temps de la fonction nbjetons :

```
def nbjetons(l,i,j,direc,t,g):
    jm=len(l);im=len(l[0])
    LL=L(im)
    LC=L(jm)
    sl=["[ ]"]
    stop1=False
    stop2=False
    for sup in range(1,g):
        if direc=="ligne":  #construire sl en fonction d'un argument direction
            if j+sup in LC and stop1==False:
                if l[j+sup][i]!=inv(t):
                    sl.append(l[j+sup][i])
                else:
                    stop1=True
            if j-sup in LC and stop2==False:
                if l[j-sup][i]!=inv(t):
                    sl.insert(0,l[j-sup][i])
                else:
                    stop2=True
```

La fonction crée une liste **sl**, qui pour une case donnée, sera composée des cases adjacentes à celles-ci, sans jetons adverses.

Idem pour les autres direction, puis :

```
    nbjmax=0
    if len(sl)>=g:
        for k in range(0,len(sl)-(g-1)):
            nbj=0
            ssl=[]
            for p in range(0,g):
                ssl.append(sl[k+p])
            for i in range(0,g):
                if ssl[i]==t:
                    nbj=nbj+1
            if nbjmax<nbj:
                nbjmax=nbj
    return(nbjmax)
```

On compte le nombre maximal de jetons (pas forcément consécutifs) dans chaque sous liste **ssl** (composées de 4 cases) de **sl.**

- Création ensuite de la fonction maxix :

```
def maxixloc(l,i,j,t,g):
    ligne=nbjetons(l,i,j,"ligne",t,g)
    col=nbjetons(l,i,j,"col",t,g)
    diagm=nbjetons(l,i,j,"diagm",t,g)
    diagd=nbjetons(l,i,j,"diagd",t,g)
    return(max(ligne,col,diagm,diagd))
```

maxixloc renvoie le maximum des nombres obtenus dans les **différentes directions.**

```
def maxix(l,t,g):
    jm=len(l);im=len(l[0])
    maxix=0
    for i in range(0,im):
        for j in range(0,jm):
            if l[j][i]=="[ ]":
                maxx=maxixloc(l,i,j,t,g)
                if maxix<maxx:
                    maxix=maxx
    return(maxix)    #compris entre 0 et 3
```

Puis maxix renvoie le maximum de ces nombres sur chacune des **cases vides** du tableau.

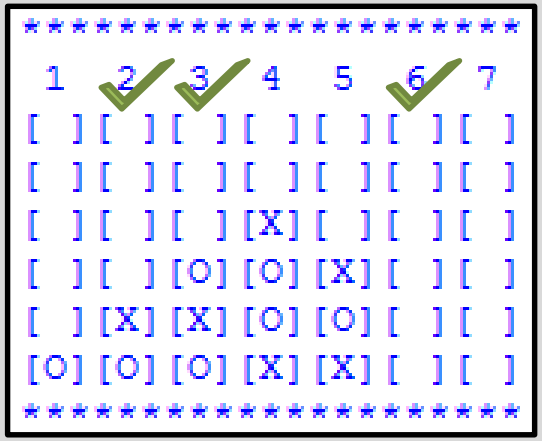
- Implémentation de maxix dans une nouvelle sélection :

```
def slc2(l,lcj,t,txt,g):
    m=0
    lcjnew=list(lcj)
    for c in lcj:
        lt=ajouter(l,c,t)
        lti=ajouter(lt,c,inv(t))
        maxit=maxi(lt,t,g)
        maxixt=maxix(lt,t,g)
        maxiti=maxi(lti,inv(t),g)
        if txt==1:
            print(c,maxit,maxixt)
        if maxixt>m and maxiti<g:
            lcjnew=[c]
            m=maxixt
        elif maxixt==m and maxiti<g and c not in lcjnew:
            lcjnew.append(c)
    return(lcjnew)
```

- Calcul de la longueur maximale **maxixt** des lignes du tableau auquel on a rajouté un jeton.

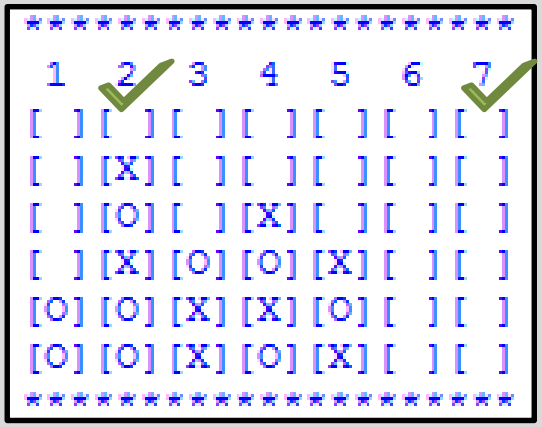
- **m** est la longueur maximale de lignes éventuellement entrecoupées de vide que l'on peut former au tour suivant.

- Reprise de l'exemple précédent avec la nouvelle sélection utilisant maxix :



-Sur l'exemple précédent (gauche), slc2 renverra [2,3,6], ce qui correspond à ce que l'on veut pour [O].

- slc2 sera plus efficace dans d'autres situations, comme celle de droite pour [X] où il renverra [2,7].



Problème : les fonctions de sélection ne vérifient pas que les cases soient valides comme pour la méthode rétrograde.

- Vérification de la possibilité de jouer dans les colonnes sélectionnées par slc2 :

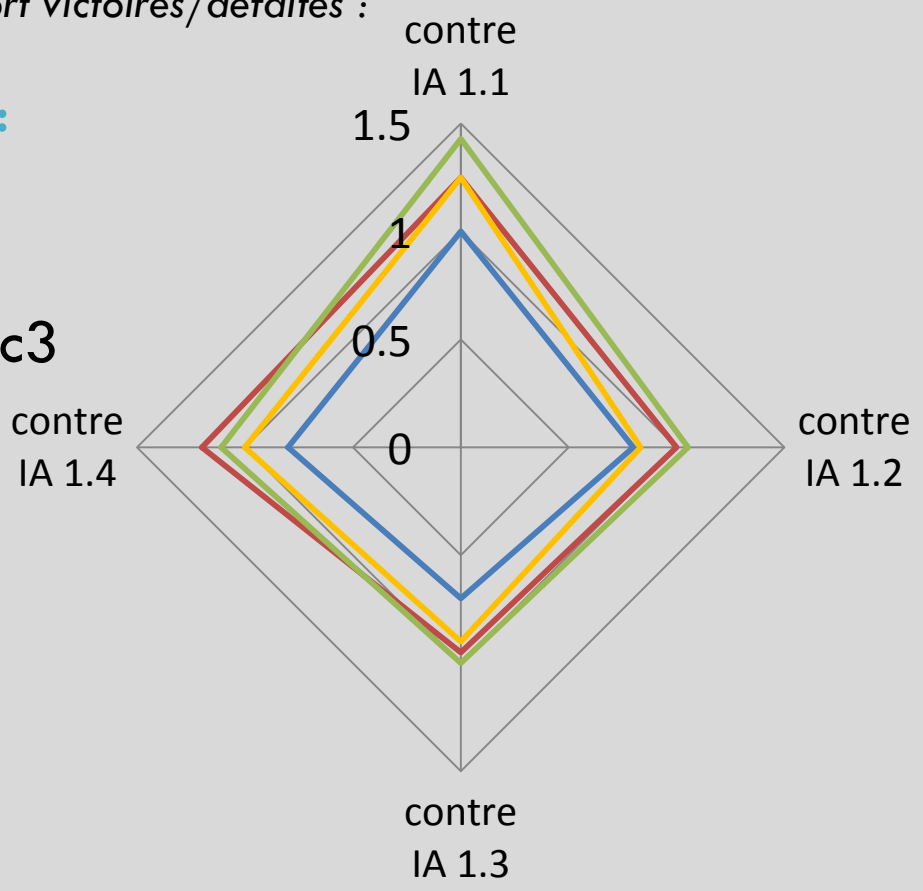
```
def slc3(l,lcj,t,txt,g):#enlever les colonnes de lcj ou l'adv gagne au tour suivant
    lcjnew=listcolok(l,slc2(l,lcj,t,txt,g),t,g)
    if txt==1:
        print(lcjnew,"slc3")
    if lcjnew!=[]:
        return(lcjnew)
    else:
        return(lcj)
```

- Elimination des colonnes de slc3 donnant des menaces multiples si l'adversaire joue au-dessus :

```
def slc4(l,lcj,t,txt,g):#enlever les colonnes a menaces multiples
    lcjnew=list(slc3(l,lcj,t,txt,g))
    if txt==1:
        print(lcjnew,"slc4")
    for c in lcjnew:
        lt=ajouter(l,c,t)
        lti=ajouter(lt,c,inv(t))
        if maxix(lti,inv(t),g)==3 and len(maxixplaces(lti,inv(t),g)[1])>=2:
            lcjnew.remove(c)
    if lcjnew!=[]:
        return(lcjnew)
    else:
        return(lcj)
```

Résultats intermédiaires (1) :

Représentation graphique des tests de comparaison entre les différentes IA (sur 10 000 parties) par leur rapport victoires/défaites :



Observations :

- slc2 > slc1
- slc3 ~ slc2
- slc4 < slc2,slc3

Légende des IA :

1.1 = slc1
1.2 = slc2
1.3 = slc3
1.4 = slc4
2 = intérêt décroissant

L'ajout de sélections supplémentaires semblent bloquer la progression de l'IA et n'est pas efficace.

→ Donc possibilité de repartir sur un chemin différent.

3) Méthode par « intérêt décroissant »

N'utilise que le tableau à l'étape présente et non les tableaux des tours suivants.

- Ordre de priorité :

Lignes de 4 possibles ?
Lignes de 4 à bloquer ?
Lignes de 3 possibles ?

⋮

Lignes de 2 à bloquer ?
Sinon : au hasard

➔ Prend donc comme ordre de priorité la longueur des lignes, ce qui semble cohérent.

```
def ia2(l,t,txt,g):
    for c in listcolnonpleines(l):
        if maxi(ajouter(l,c,t),t,g)>=g:
            if txt==1:
                print("action directe atk")
            return([ajouter(l,c,t),c])
    for c in listcolnonpleines(l):
        if maxi(ajouter(l,c,inv(t)),inv(t),g)>=g:
            if txt==1:
                print("action directe def")
            return([ajouter(l,c,t),c])
    maxixplc=maxixplaces(l,t,g)
    maxixplcinv=maxixplaces(l,inv(t),g)
    listmax=filtreplc(l,t,g,maxixplc[1])
    listmaxinv=filtreplc(l,t,g,maxixplcinv[1])
    if txt==1:
        print(maxixplc[0],listmax)
        print(maxixplcinv[0],listmaxinv)
    if maxixplc[0]>=2 and listmax!=[]:
        cf=random.choice(listmax)[0]
    elif maxixplcinv[0]>=2 and listmaxinv!=[]:
        cf=random.choice(listmaxinv)[0]
    elif maxixplc[0]>=1 and listmax!=[]:
        cf=random.choice(listmax)[0]
    elif maxixplcinv[0]>=1 and listmaxinv!=[]:
        cf=random.choice(listmaxinv)[0]
    else:
        lok=listcolok(l,listcolnonpleines(l),t,g)
        if lok!=[]:
            cf=random.choice(lok)
        else:
            cf=random.choice(listcolnonpleines(l))
    return([ajouter(l,cf,t),cf])
```

Résultats intermédiaires (2) :

Malgré une logique apparente : reste en pratique moins efficace que slc2 et slc3 (0.75 victoires pour 1 défaite , voir courbe bleu ciel sur les sommets **IA 1.2** et **IA 1.3** :

Observations : IA 1.1 < IA 2 < IA 1.4

Légende des IA :

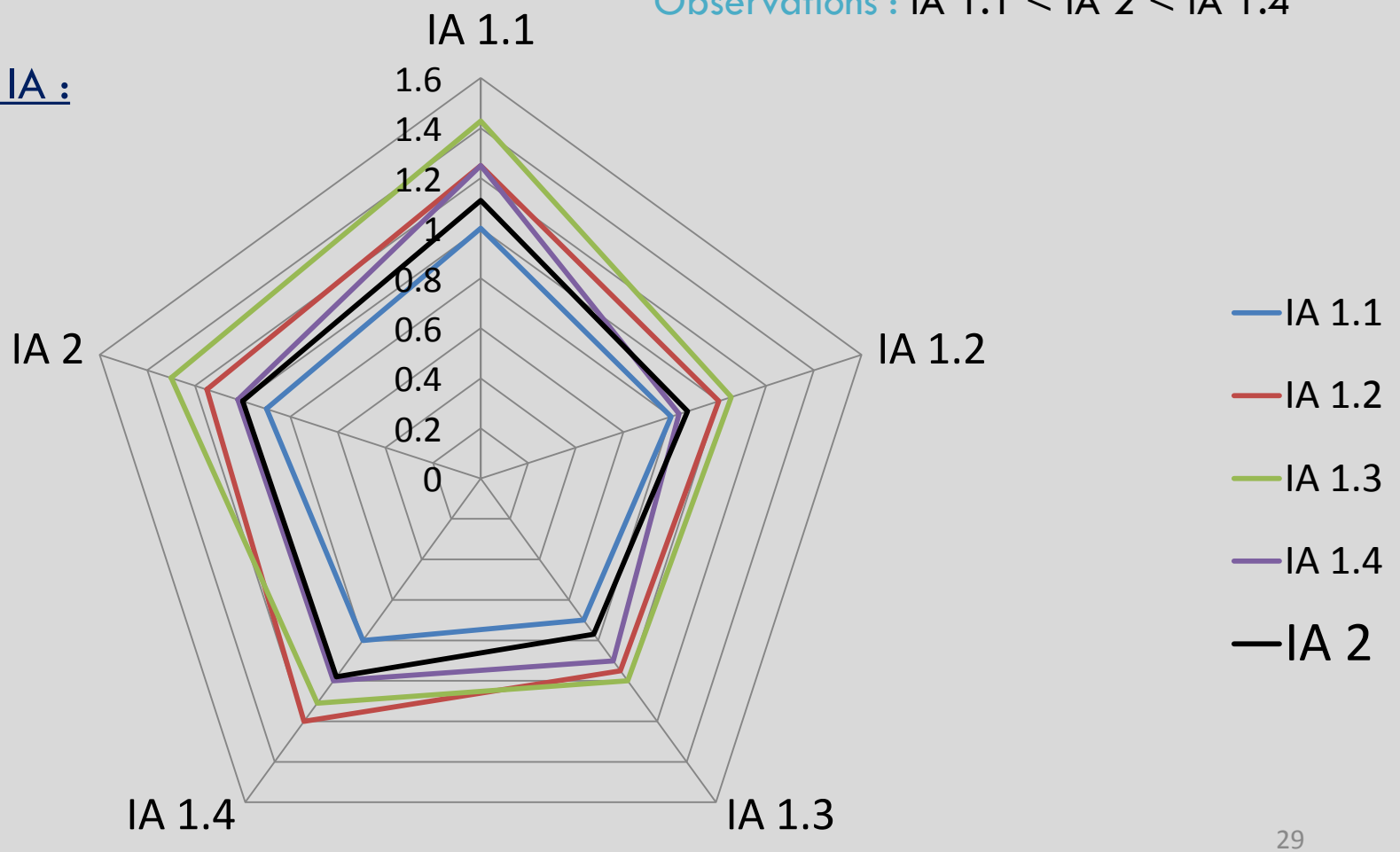
1.1 = slc1

1.2 = slc2

1.3 = slc3

1.4 = slc4

2 = intérêt décroissant



Autre piste de recherche :

- Idée d'utiliser les données du tour précédent :

→ Constamment gêner l'adversaire en jouant au-dessus de ce dernier seulement lorsqu'il n'y a pas d'autre priorités.

4) Association de la méthode rétrograde, sélective et imitative

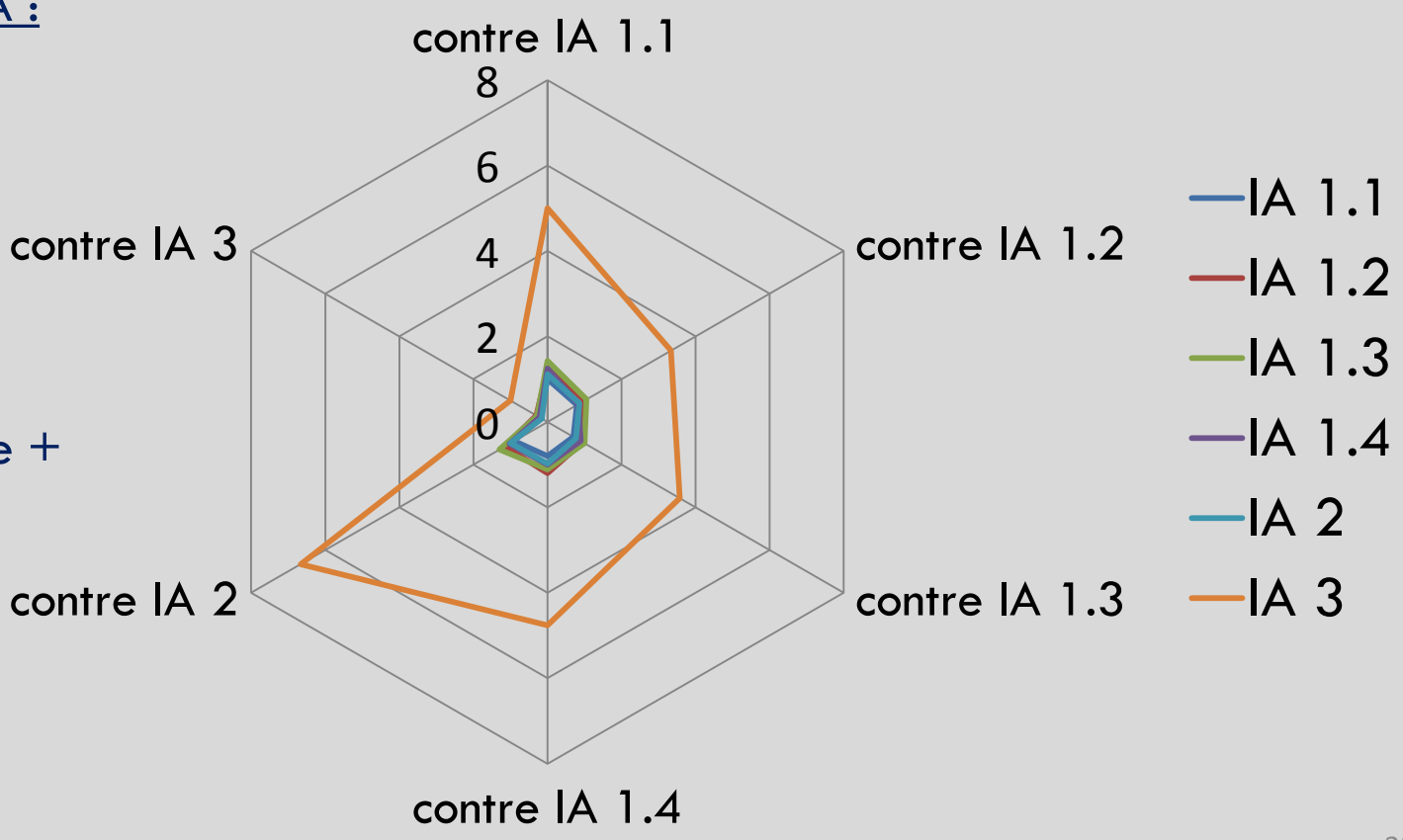
Si on fait attention de contrer l'adversaire par ses menaces simples et multiples, que l'on ne lui donne pas accès à d'autres et que le reste du temps on joue au-dessus de lui, on obtient :

```
def ia3(l,t,txt,g,T,C):
    for c in listcolnonpleines(l):
        if maxi(ajouter(l,c,t),t,g)>=g:          # On regarde si l'on peut gagner
            if txt==1:
                print("action directe atk")
            return([ajouter(l,c,t),c])
    for c in listcolnonpleines(l):
        if maxi(ajouter(l,c,inv(t)),inv(t),g)>=g:# On regarde si l'adversaire peut gagner
            if txt==1:
                print("action directe def")
            return([ajouter(l,c,t),c])
    cf=0
    if txt==1:
        print(cf,"ini")
    if cf==0:
        pg=piege(l,t,txt,g)+pg2(l,t,txt,g)
        if pg!=[]:
            cf=pg[0] #piege                      # On vérifie les menaces multiples
    if txt==1:
        print(cf,"postpg")
    if C!=[] and cf==0:
        c=C[len(C)-1]                                    #On vérifie que l'on peut jouer sans
        if c in listcolok(l,listcolnonpleines(l),t,g): #perdre si l'adversaire joue au-dessus
            cf=c                                         #et dans ce cas on joue la même
    if cf==0:                                            #colonne que lui au tour précédent
        lokmax=listcolok(l,maxixcol(l,t,g),t,g)
        if lokmax!=[]:
            cf=random.choice(lokmax)
        else:
            lok=listcolok(l,listcolnonpleines(l),t,g)
            if lok!=[]:
                cf=random.choice(lok)
            else:
                cf=random.choice(listcolnonpleines(l))
    return([ajouter(l,cf,t),cf])
```

IV/ Bilan et conclusion

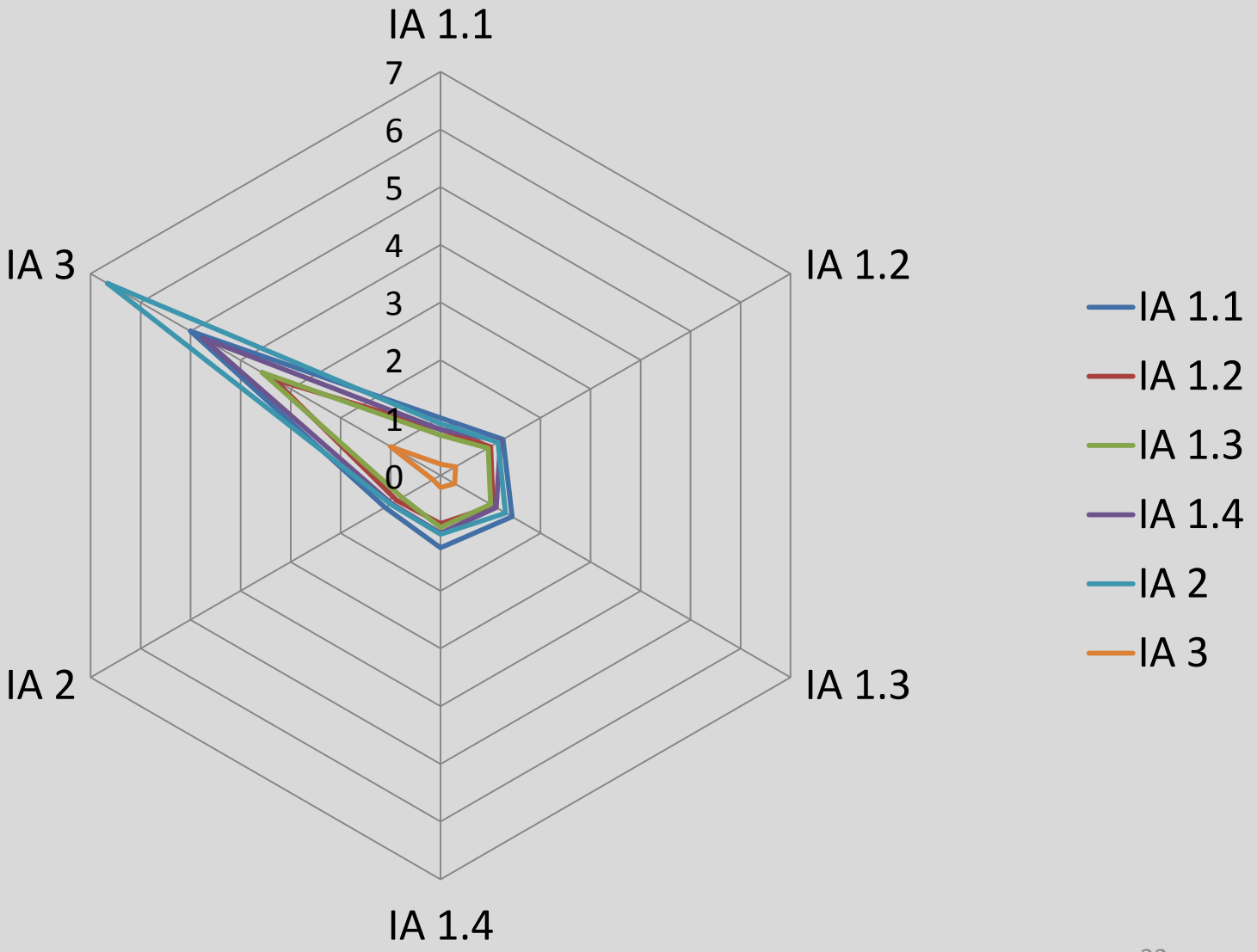
Représentation graphique des tests de comparaison entre les différentes IA (sur 10 000 parties) par leur rapport victoires/défaites :

Légende des IA :
1.1 = slc1
1.2 = slc2
1.3 = slc3
1.4 = slc4
2 = intérêt décroissant
3 = rétrograde + imitation + slc



Annexe 1

Légende des IA :
1.1 = slc1
1.2 = slc2
1.3 = slc3
1.4 = slc4
2 = intérêt décroissant
3 = rétrograde + imitation + slc



Annexe 2

Légende des IA :

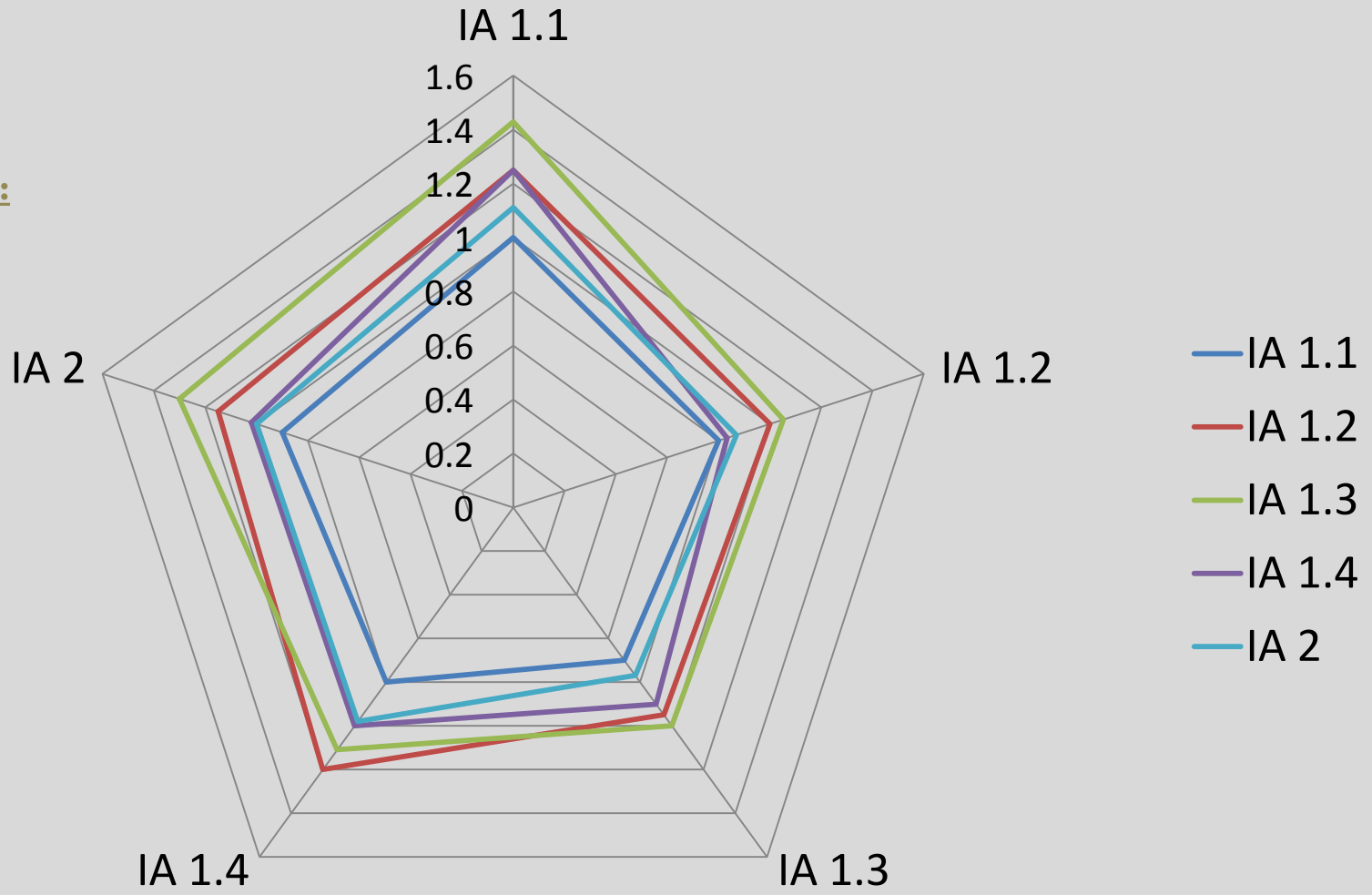
1.1 = slc1

1.2 = slc2

1.3 = slc3

1.4 = slc4

2 = intérêt décroissant



Annexe 3

Légende des IA :

1.1 = slc1

1.2 = slc2

1.3 = slc3

1.4 = slc4

2 = intérêt décroissant