In [3…

```
%load_ext notexbook
%texify
```

```
The notexbook extension is already loaded. To reload it, use:
  %reload_ext notexbook
```

Interaction stratégique

I. Jouer en simultané

On considère pour commencer les situations où les agents vont jouer leurs actions de manière simultanée. Pour la simplicité, les exemples impliquent à chaque fois deux agents, mais les notions restent valides pour plus de deux agents.

1. Un problème classique: le dilemme du prisonnier

Deux joueurs ont été appréhendés après un vol. Il est clair que au moins un des deux a commis ce délit. Ils sont interrogés en parallèle, et disposent des mêmes actions: garder le **silence**, ou **dénoncer** l'autre. Ils jouent donc en simultané, sans se concerter. Selon le choix de stratégies des joueurs, les gains sont les suivants:

R \ B	**silence**	**dénoncer**
silence	(20,20)	(0,30)
dénoncer	(30,0)	(10,10)

Analysons du point de vue du joueur R les **meilleures réponses** selon les choix de l'agent B:

- si B garde le silence, il vaut mieux que R dénonce l'autre (30>20)
- si B dénonce, il vaut mieux que R dénonce aussi (10>0)

Les meilleures réponses pour R sont donc:

meilleure réponse	**action de B**
dénonce	dénonce
dénonce	silence

Stratégie dominante

Lorsque, pour un joueur X, la table de meilleures réponses indique toujours la même stratégie, on parle de **stratégie dominante** pour le joueur X.

> La stratégie **dénoncer** est donc dominante pour R et pour B

Stratégies dominées

Il est rare qu'il existe des stratégies dominantes. Mais il est plus courant de pouvoir identifier des **stratégies dominées** par une autre stratégie. Il est en fait possible de distinguer plusieurs niveaux de dominance:

- une stratégie s **domine fortement** une autre stratégie s' si les gains obtenus pour s sont toujours (ie. quelque soit la stratégie de l'autre joueur) strictement supérieurs à ceux de la stratégie s'.

- une stratégie s **domine faiblement** une autre stratégie s' si les gains obtenus pour s sont toujours (ie. quelque soit la stratégie de l'autre joueur) supérieurs ou égaux à ceux de la stratégie s', et strictement supérieurs pour au moins une stratégie de l'autre agent.

Considérons cet autre exemple:

R \ B	**a**	**b**	**c**
d	(20,20)	(0,30)	(20,10)
e	(30,10)	(10,0)	(0,10)

On constate qu'ici, pour l'agent R, aucune stratégie ne domine l'autre. Pour l'agent B, la stratégie **c** est dominée par la stratégie **a**, mais aucune stratégie n'est dominante.

Equilibre de Nash

Lorsque les stratégies sont des **meilleures réponses mutuelles**, on se trouve dans un **équilibre de Nash**: aucun agent n'aurait intérêt, unilatéralement, a changer de stratégie.

> Ici la stratégie jointe **dénoncer/dénoncer** est un EN

Ici, on observe que cet équilibre de Nash est dominé par le résultat (20,20) qui serait atteint si les deux joueurs gardaient le silence.

Ce type de dilemme a été largement étudié, et même utilisé dans des jeux (par exemple dans l'émission de TV britannique Golden Balls, qui repose sur une situation proche).

On a pour coutume de parler d'action de **coopération**, ou de **trahison** dans ce cadre. Notons que l'on peut varier les valeurs dans de gains la matrice sans changer la nature du dilemme. Il y a en effet une valeur obtenue en cas de coopération mutuelle (cc), une valeur de piège obtenue en cas de trahison mutuelle (tt), et dans le cas (tc), une valeur T de traitrise et une de dupe (tc). Pour rester dans une situation de dilemme, il faut que tc > cc > tt > ct.

In [4…
```
def jouerDP(s1,s2):
    gains = {'cc':(20,20), 'ct':(0,30), 'tt':(10,10), 'tc': (30,0
    return gains[str(s1)+str(s2)]

print (jouerDP('c','c'))
```

```
(20, 20)
```

2. Dilemme du prisonnier itéré

Imaginons à présent que deux joueurs (toujours les mêmes) jouent plusieurs fois d'affilée un dilemme du prisonnier.

Définir une stratégie pour le dilemme du prisonnier itéré, c'est donc définir quel coup jouer, en fonction de :

- les coups que l'on a joués précédemment
- les coups que l'autre a joués précédemment

Notez que cela inclue également quel premier coup jouer. Les stratégies peuvent également faire intervenir des aspects stochastiques, etc.

Des stratégies classiques sont les suivantes:

- toujours coopérer/trahir
- jouer au hasard
- alterner trahison et coopération
- donnant-donnant (tit-for-tat): commencer par coopérer, puis jouer comme le dernier coup joué par l'adversaire

- rancunière: on coopère, et dès que l'autre trahit, on trahit pour toujours

In [1…

```
import random
def prochainCoup(mesCoups,adversCoups,nom):
    """selon la liste de mes coups et des coups de l'autre
    je choisis un coup """

    #
    # strategies de base
    #
    if nom=='trahison':
        return 't'

    if nom=='coopere':
        return 'c'

    if nom=='titfortat':
        if adversCoups == []:
            return 'c'
        else:
            return adversCoups[-1]

    if nom=='random':
        return random.choice(['t','c'])
    if nom=='alterne':
        if mesCoups == []:
            return 'c'
        elif mesCoups[-1]=='t':
            return 'c'
        else:
            return 't'
    #
    # quelques strategies proposees par les etudiants en cours
    #

    if nom=='titfortatBis':
        if adversCoups == []:
            return 't'
        else:
            return adversCoups[-1]
    if nom=='titfortatTer':
        if adversCoups == []:
            return 'c'
        else:
            if len(mesCoups) == 1:
                return adversCoups[-1]
            elif adversCoups[-1]=='t' or (mesCoups[-1]=='t' and m
                return 't'
            else:
                return 'c'
```

```
    if nom=='equipeABis':
        nbC = 0
        nbT = 0
        for i in adversCoups:
            if i =='c':
                nbC += 1
            else:
                nbT += 1
        if len(adversCoups)<4:
            return 'c'
        else:
            if nbT>nbC:
                return 't'
            else:
                return 'c'


    if nom=='equipeA':
        if (len(adversCoups) < 5):
            return 'c'
        return 't'
```

In [1…

```
print (prochainCoup([],[],'random'))
```

```
c
```

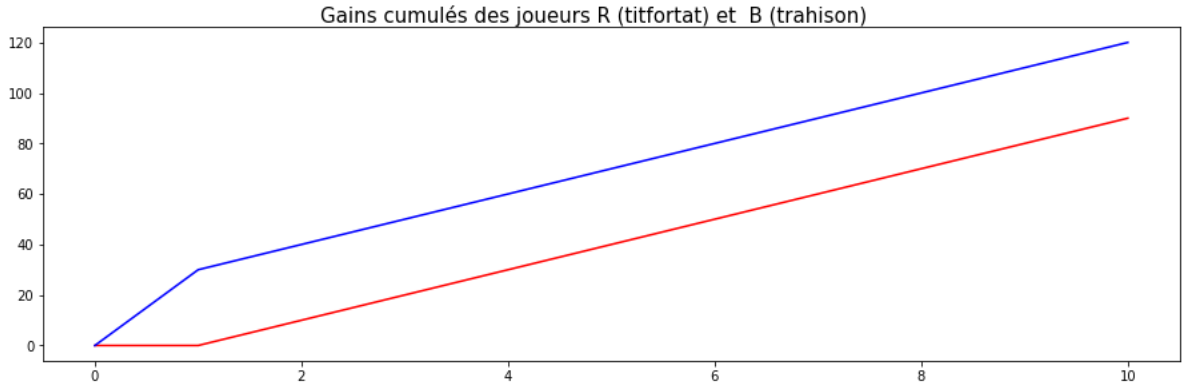
In [1…

```
nbIterations=10
toPlot = {'R':[0], 'B':[0]}
coupsJoues = {'R': [], 'B': []}
gainsCumules = {'R': 0, 'B': 0}
strategie = {'R': 'titfortat', 'B': 'trahison'}

for i in range(nbIterations):
    coupR = prochainCoup(coupsJoues['R'],coupsJoues['B'],strategi
    coupB = prochainCoup(coupsJoues['B'],coupsJoues['R'],strategi
    gainR,gainB = jouerDP(coupR,coupB)
    gainsCumules['R']+=gainR
    gainsCumules['B']+=gainB
    coupsJoues['R']+=coupR
    coupsJoues['B']+=coupB
    #print ("R joue ",coupR, ", B joue", coupB, "⟶ gains:", gai
    toPlot['R'].append(gainsCumules['R'])
    toPlot['B'].append(gainsCumules['B'])
```

In [1…

```
%matplotlib inline
import numpy as np
from matplotlib import pyplot as plt
from IPython.core.pylabtools import figsize
import networkx as nx
import pylab
figsize(12.5, 4)
#print (toPlot['R'],toPlot['B'])
p = np.linspace(0, nbIterations, nbIterations+1)
plt.plot(p, toPlot['R'], color='red')
plt.plot(p, toPlot['B'], color = 'blue')
legende = "Gains cumulés des joueurs R (" + strategie['R'] + ") e
plt.suptitle(legende, y=1.02, fontsize=15)
plt.tight_layout()
```



> En 1980, Robert Axelrod propose d'organiser des confrontations de stratégies lors d'un tournoi https://en.wikipedia.org/wiki/The_Evolution_of_Cooperation. C'est à cette occasion que la stratégie tit-fot-tat est proposée, par Anatol Rapoport.

Comment évaluer la qualité des stratégies proposées? Il faut définir précisément ce que l'on veut dire par *meilleure stratégie*. On peut penser au moins à deux définitions:

- une stratégie qui bat toutes (ou le maximum) d'autres stratégies lors des matchs un contre un
- une stratégie qui obtient un gain cumulé après tous les matchs le plus élevé possible.

Si l'objectif est simplement de battre les autres stratégies, alors la stratégie *toujours trahir* est la meilleure. Mais si l'objectif est de maximiser le gain sur l'ensemble des rencontres, alors les choses sont moins claires... En pratique, Tit-for-tat s'avère être une excellente stratégie. Pourtant elle est clairement battue par une stratégie de trahison constante, par exemple. Mais elle obtient de très bons scores contre toutes les stratégies.

Pour aller plus loin sur sur le dilemme du prisonnier itéré:

- le site et les travaux de B. Beaufils, J.-P. Delahaye et Ph. Mathieu à l'Université de Lille. http://www.lifl.fr/IPD/ipd.html

- un module Python axelrod (https://axelrod.readthedocs.io/en/stable/) permettant de rejouer les tournois d'Axelrod, avec plus de 200 stratégies pré-codées.
- et voir aussi la magnifique *explorable explanation* The Evolution of Trust (http://ncase.me/trust/), par Nicky Case.

Les stratégies stochastiques

De manière générale, une stratégie stochastique à mémoire 1 coup est une stratégie qui donne la probabilité de joueur c au coup suivant, selon les coups joués par les joueurs au tour précédent:

$P = \{Pcc, Pct, Ptc, Ptt\}$

Comment modéliser le tit-for-tat?

A quoi correspond une stratégie $P = (1, 0, 0, 1)$

Les ZD-stratégies

En 2012, Press et Dyson ont publié un article qui a fait sensation dans le domaine. Ils définissent une famille paramétrique de stratégies stochastiques, où les probabilités sont liées selon les équations suivantes (pour le cas où les gains sont 5/3/1):

$$Pcc = 3a + 3b + c$$
$$Pct = 1 - 5b + c$$
$$Ptc = 5a + c$$
$$Ptt = a + b + c$$

De manière remarquable, lorsqu'une stratégie $ZD(a, b, c)$ joue contre une autre statégie quelconque G_2, leurs gains sont liés de manière linéaire $aG1 + bG2 + c = 0$. En particulier, si $a = 0$ et $b \neq 0$, le gain de G2 est fixe (-b/c)!

Sur les statégies ZD, voir également l'article de J.-P. Delahaye Le dilemme du prisonnier et l'illusion de l'extorsion (http://cristal.univ-lille.fr/~jdelahay/pls/242.pdf)

II. Jouer séquentiellement

Considérons à présent un deuxième jeu très classique: **le jeu de la poule mouillée** (*the game of chicken*)

R \ B	**sauter**	**rester**
sauter	(5,5)	(2,7)
rester	(7,2)	(0,0)

Si l'on analyse ce jeu en tenant à présent compte de la séquence, c'est-à-dire du fait que un joueur jour après l'autre, que pouvons-nous en dire?

In [5…

```
#--- pour petits tests rapides ---#


# exemple Game of chicken
successeurs = {'init':['a-saute','a-reste'],'a-saute':['a-saute-b
valeur = {'a-saute-b-saute':(5,5), 'a-reste-b-saute':(7,2), 'a-sa


def feuille(state): # les feuilles n'apparaissent pas comme clés
    return state not in successeurs
```

1. Induction à rebours et algorithme Minimax

On représente le jeu comme un arbre, où chaque joueur joue alternativement à chaque niveau de profondeur. L'analyse peut alors se faire à rebours: en remontant depuis les feuilles, on fait suppose que le joueur, confronté à plusieurs coups possibles, choisira celui qui lui procure *à ce stade-là* le meilleur gain. Au niveau au-dessus on peut itérer ce même raisonnement, suppoant ce que fera le joueur suivant, et ainsi de suite jusqu'à la racine. C'est le principe de l'**induction à rebours**.

In [4…

```
from operator import itemgetter


def backward_induction(state,turn_taking):
    """ implementation de backward induction, turn_taking: sequen
        """
    v = maxValue(state,0,turn_taking)
    return v

def maxValue(state,depth,turn_taking):
    if feuille(state): # si feuille on renvoie la valeur
        print("feuille:", valeur[state])
        return valeur[state]

    v = inf # joue le role de infini

    for s in successeurs[state]:
        print ("étendu noeud ", s)
        player= turn_taking[depth]
        print("player:", player)
        # on remonte le vecteur de score pour lequel la valeur es
        v = max([v,maxValue(s,depth+1,turn_taking)],key=itemgette
    return v

#————————————————
# test
#————————————————
turn_taking=[0,1] # le joueur 0 jour, puis le joueur 1
inf = [-1000]*len(turn_taking) # joue le role de infini
print (backward_induction('init',turn_taking))
```

```
étendu noeud  a-saute
player: 0
étendu noeud  a-saute-b-saute
player: 1
feuille: (5, 5)
étendu noeud  a-saute-b-reste
player: 1
feuille: (2, 7)
étendu noeud  a-reste
player: 0
étendu noeud  a-reste-b-saute
player: 1
feuille: (7, 2)
étendu noeud  a-reste-b-reste
player: 1
feuille: (0, 0)
(7, 2)
```

Dans le cas où le jeu est un jeu à **somme nulle**, on peut se contenter de représenter la valeur au feuille de l'arbre par une valeur unique: un joueur cherche alors à minimiser le score tandis que l'autre cherche à le maximiser, on parle donc d'algorithme **minimax**. Notez qu'il s'agit d'un cas particulier d'induction à rebours. (Voir le code ci-dessous dans lequel les fonctions `minValue` et `maxValue` sont distinguées, par soucis de lisibilité).

La valeur retournée par l'algorithme de recherche minimax est appelée **la valeur théorique du jeu**, qui serait obtenue sous l'hypothèse que les agents jouent rationnellement selon cet algorithme.

In [5…

```
def minimax(state):
    """ implementation de minimax, version Russel & Norvig, Chapt
        """
    v = maxValue(state)
    return v

def maxValue(state):
    if feuille(state): # si feuille on renvoie la valeur
        return valeur[state]
    v = -inf
    for s in successeurs[state]:
        print ("étendu noeud ", s)
        v = max(v,minValue(s))
    return v

def minValue(state):
    if feuille(state): # si feuille on renvoie la valeur
        return valeur[state]
    v = inf
    for s in successeurs[state]:
        print ("étendu noeud ", s)
        v = min(v,maxValue(s))
    return v

#——————————————————
# test
#——————————————————

inf = 1000 # joue le role de infini
successeurs = {'a':['b','c'],'b':['d','e'],'c':['f','g'],'d':['h'
valeur = {'h':4, 'i':9, 'j': 8, 'k': 12, 'l':5, 'm':6, 'n':1, 'o'
print (minimax('a'))
```

```
étendu noeud  b
étendu noeud  d
étendu noeud  h
étendu noeud  i
étendu noeud  e
étendu noeud  j
étendu noeud  k
étendu noeud  c
étendu noeud  f
étendu noeud  l
étendu noeud  m
étendu noeud  g
étendu noeud  n
étendu noeud  o
9
```

3. Elagage Alpha-beta (pruning)

L'algorithme minimax étend l'intégralité des noeuds de l'arbre. Pour éviter cette exploration complète de l'arbre de jeu, on peut exploiter le fait que les joueurs sont des MINimisateurs et des MAXimisateurs. En chaque noeud, on va garder mémoire dans la recherche de deux valeurs, qui vont parfois permettre de couper certaines parties de l'arbre de recherche.

- la valeur α est une borne inf que la recherche maximise. C'est donc la valeur min que peut se garantir le joueur MAX.
- la valeur β est une borne sup que la recherche minimise. C'est donc la valeur max que peut se garantir le joueur MIN.

Donc le joueur MAX met à jour les valeurs α, et le joueur MIN met à jour les valeur β.

Il y a donc deux types de coupes dans l'arbre de recherche qui peuvent survenir:

- une **coupe** α survient lorsque, pour un noeud MIN, on a trouvé un noeud parmi les fils avec une valeur $\leq \alpha$
- une **coupe** β survient lorsque, pour un noeud MAX, on a trouvé un noeud parmi les fils avec une valeur $\geq \beta$

Pour bien comprendre le principe de la coupe, il faut raisonner sur deux niveaux (MAX et MIN). Par exemple, pour une coupe α, si on trouve pour un noeud MIN une valeur $\leq \alpha$ cela signifie que MIN remontera une valeur $\leq \alpha$ (il minimise), quelque soient les valeurs des autres fils explorés. Or (au dessus) MAX a une garantie d'obtenir au moins α. Poursuivre l'exploration est donc superflu.

In [5…

```
def alphabeta(state):
    """ implementation de alphabeta, version Russel & Norvig, Cha
        """
    v = maxValue(state,-inf,inf)
    return v

def maxValue(state,alpha,beta):
    if feuille(state): # si feuille on renvoie la valeur
        return valeur[state]
    v = -inf
    for s in successeurs[state]:
        print ("étendu noeud ", s)
        v = max(v,minValue(s,alpha,beta))
        if v ⩾ beta: # coupe beta, pas la peine d'étendre les au
            print ("coupe beta")
            return v
        alpha = max(alpha,v) # mise à jour de alpha par MAX
    return v

def minValue(state,alpha,beta):
    if feuille(state): # si feuille on renvoie la valeur
        return valeur[state]
    v = inf
    for s in successeurs[state]:
        print ("étendu noeud ", s)
        v = min(v,maxValue(s,alpha,beta))
        if v ⩽ alpha: # coupe alpha, pas la peine d'étendre les
            print ("coupe alpha")
            return v
        beta = min(beta,v)
    return v

#——————————————————
# test
#——————————————————

inf = 1000 # joue le role de infini
successeurs = {'a':['b','c'],'b':['d','e'],'c':['f','g'],'d':['h'
valeur = {'h':4, 'i':9, 'j': 8, 'k': 12, 'l':5, 'm':6, 'n':1, 'o'

print (alphabeta('a'))
```

```
étendu noeud  b
étendu noeud  d
étendu noeud  h
étendu noeud  i
étendu noeud  e
étendu noeud  j
étendu noeud  k
coupe beta
étendu noeud  c
étendu noeud  f
étendu noeud  l
étendu noeud  m
coupe alpha
9
```

3. Pour aller plus loin

Limites des concepts de solution proposés

On observe souvent que les humains ne se comportent pas de la manière prescrite par les solutions de théorie des jeux. En particulier, de nombreux jeux reposent sur une notion de **confiance réciproque** qui peut amener à des solutions plus favorables pour tout le monde. Un des plus connu est le **Trust Game**: un agent, doté d'un budget **B**, a le choix entre donner un certaine somme à un autre agent et ne rien donner. S'il donne une somme `x` à l'agent 2, cette somme est triplée et l'agent 2 a alors le choix entre ne rien donner, ou donner une somme en retour à l'agent 1. Par induction à rebours, l'agent 1 ne doit rien donner initialement, car il n'y a aucune incitation pour l'agent 2 a lui rendre de l'argent... Toutefois les expériences avec des sujets humains contredisent ces prédictions.

Des travaux tentent donc de capturer ces notions de confiance réciproque. Intuitivement, si l'on considère le jeu séquentiel suivant:

- 1: jouer A: (2,0)
- 1: jouer B:

- 2: jouer C: (0,2)
- 2: jouer D: (2,1)

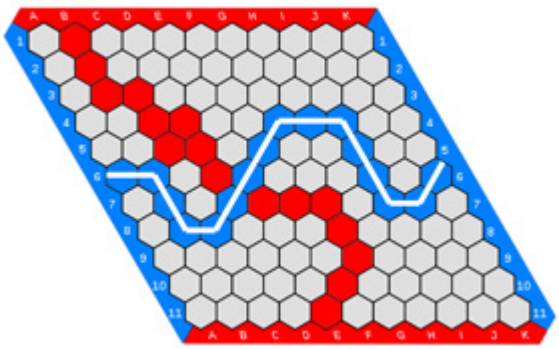
Dans ce cas-là, le raisonnement suivant est possible: si le joueur 2 observe que 1 n'a pas joué A qui lui aurait garanti un gain de 2, par réciprocité le joueur 2 doit offrir au moins au moins autant à l'agent 1. Sur cette exemple, l'agent 2 devrait donc jouer D selon ce principe. Mais ce principe peut être itéré, ce qui amène à des nouveaux concepts de solutions.

> Pour aller plus loin: Joshua Letchford, Vincent Conitzer, and Kamal Jain. An "Ethical" Game-Theoretic Solution Concept for Two-Player Perfect-Information Games. In Proceedings of the Fourth Workshop on Internet and Network Economics (WINE-08), pp. 696-707, Shanghai, China, 2008

Récréation finale: l'exemple du jeu de Hex

Le jeu de Hex est un jeu combinatoire dont les règles s'expriment très simplement, mais qui est d'une grande richesse. Les règles sont les suivantes:

- chaque joueur peut poser à son tour une tuile n'importe où sur le terrain
- le joueur qui parvient à relier ses deux bordures (de même couleur) est le gagnant



Ce jeu possède quelques propriétés remarquables:

- les matchs nuls ne peuvent pas se produire
- tout joueur jouant en premier dispose d'une **stratégie gagnante** (il peut donc gagner quelque soit les coups joués par les autres).

Toutefois ces stratégies gagnantes ne sont connues que pour les petites tailles de parties (jusqu'à 7x7 à ma connaissance).

> Version du 10 Mars 2021. Style *no*TEX*book* (voir ici) (https://pypi.org/project/notexbook-theme/). En cas de problème d'affichage, ouvrir l'URL dans le nbviewer.jupyter.org/ Pour profiter pleinement du notebook et pouvoir tester de manière interactive, vous devez l'ouvrir par `jupyter notebook`

In […