Algorithmes de résolution du jeu du dollar

Table des matières

1 Introduction

Le jeu du dollar est un problème posé par N.L.Biggs [**1**] et dérivé du *chip-firing game*, introduit dans les années 90 par A. Björner, L. Lovász et P. W. Shor. [**2**]. Le jeu consiste en un graphe connexe (orienté ou non) où chaque sommet est pondéré par un entier relatif, représentant une somme d'argent en dollars, et plus particulièrement une dette si le nombre associé est négatif. Le jeu prend alors un tel graphe comme condition initiale, et tour à tour, autorise deux mouvements : un prêt est un coup lors duquel un sommet donne un dollar à chacun de ses sommets adjacents, quitte à être en dette ; et un emprunt est un coup, lors duquel un sommet prend un dollar à chacun de ses sommets adjacents. Le problème de ce jeu est de pouvoir trouver une suite de coups de longueur minimale, si elle existe, telle qu'à l'issue de ceux-ci, le graphe ne comporte plus de sommets présentant des dettes.

Dans cet exposé, je m'intéresserai uniquement à des graphes non-orientés.

Je définirai dans un premier temps quelques notions et un résultat restreignant le champ des graphes étudiés. Je présenterai ensuite différents algorithmes de résolution du jeu et en ferai l'étude théorique. Ces algorithmes ont pû être testés grâce à une implémentation en python utilisant l'environnement de développement Processing.py. Ces algorithmes permettent de trouver une séquence de coups qui résolve le jeu, mais qui n'est pas minimale. Je présente alors plusieurs moyens de minimiser cette séquence, indépendamment du graphe étudié.

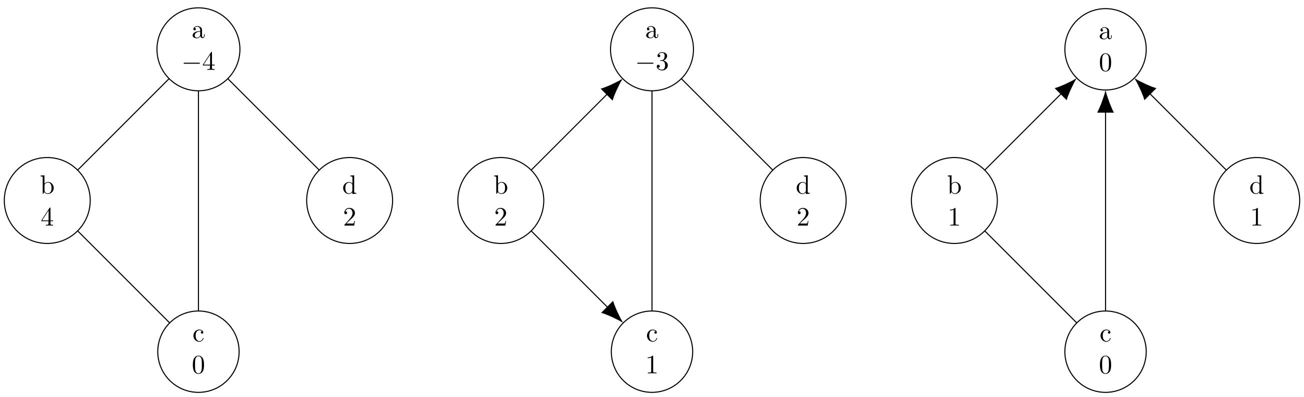

FIGURE 1 – Un exemple de séquence : un prêt est effectué par le sommet b, puis un emprunt par le sommet a. À l'issu de ces coups, aucun sommet n'a de dettes.

2 Résultats théoriques

J'introduis d'abord quelques notions.

Définition 1. *La valeur d'un sommet v, notée $\mathrm{val}(v) \in \mathbb{N}$, donne la pondération de ce sommet, c'est-à-dire le nombre de dollars que «possède» le sommet.*

Définition 2. *un sommet pauvre v est un sommet tel que $\mathrm{val}(v) < 0$. On dit aussi que c'est un sommet en dettes. Un sommet suffisant est un sommet tel que $\mathrm{val}(v) \in [\![0, \deg v - 1]\!]$. C'est un sommet qui n'a pas de dettes mais qui ne peut pas faire un prêt sans tomber en dette. Un sommet riche est un sommet tel que $\mathrm{val}(v) \geqslant \deg v$, c'est-à-dire qui peut effectuer un ou plusieurs prêts.*

Définition 3. *Une séquence de coups (S_n) est une suite d'éléments dans $V \times \{-1, 1\}$, finie ou non, qui décrit les coups effectués : Pour n un entier naturel, $S_n = (v, \varepsilon)$ indique que le $n^{\text{ième}}$ coup est effectué par le sommet v, avec $\varepsilon = 1$ si le coup est un prêt, et $\varepsilon = -1$ si le coup est un emprunt.*

La séquence de coups de la figure 1 est par exemple $((b, 1), (a, -1))$.

Définition 4. *Une partie $\mathcal{P}$ est un triplet $(G, \mathrm{val}_0, (S_n))$, où $G = (V, E)$ est le graphe sur lequel se déroule la partie, avec V l'ensemble de ses sommets et E l'ensemble de ses arêtes. $\mathrm{val}_0 : V \longrightarrow \mathbb{Z}$ est la pondération initiale en dollars des sommets du graphe. Par soucis de concision, j'appellerai par la suite un graphe un graphe pondéré par une telle fonction. Enfin, (S_n) est une séquence de coups.*

Définition 5. *On dit que la séquence est solution en n coups s'il existe un entier n tel qu'au $n^{\text{ième}}$ coup, aucun sommet n'est pauvre.*

Définition 6. *On dit que la séquence est optimale si elle est solution en n coups et s'il n'existe pas d'autres séquences qui sont solutions en m coups, avec $m < n$.*

Le théorème suivant, issu de [**3**], permet de donner une condition suffisante d'existence d'une séquence solution.

Théorème 1. *Si le nombre d'Euler du graphe* $g = |E| - |V| + 1$ *est inférieur ou égal au nombre total de dollars* N *sur le graphe, alors il existe toujours une séquence solution.*

Remarque. *Dans le cas où* g *est strictement supérieur à* N*, il n'existe pas toujours de séquences solution du jeu. En particulier, il est possible de montrer qu'il existe toujours une configuration initiale ayant* N *dollars au total pour laquelle il n'existe pas de séquence solution.*

Dans la suite, je me restreindrai uniquement aux graphes ayant au moins une séquence solution. Dans l'implémentation python notamment, je suppose que le graphe vérifie la relation $g \leqslant N$.

3 Algorithmes de résolutions

Je propose deux algorithmes permettant de trouver une séquence solution d'un graphe $G = (V, E)$ avec une pondération initiale.

J'utilise les notations suivantes :

— ENDETTE() la fonction vérifiant si le graphe contient encore des sommets endettés ou non.

— SORT_BY_VALUE(L) la fonction triant de manière croissante une liste de sommet L par leur quantité de dollars.

— BORROW(v) la procédure effectuant un emprunt par le sommet v.

— GIVE(v) la procédure effectuant un prêt par le sommet v.

Algorithm 1 Algorithme utilisant seulement des emprunts

```
while ENDETTE() do
  POOR ← []
  for all v ∈ 𝒱 do
    if val(v) < 0 then
      POOR ← {v} ∪ POOR
    end if
  end for
  POOR_SORTED ← SORT_BY_VALUE(POOR)
  v ← POOR_SORTED[0]
  BORROW(v)
end while
```

Ces deux algorithmes servent de point de départ : ils permettent de trouver une séquence solution en n coups, à partir de laquelle il est possible d'en tirer une séquence solution en m coups, avec $m < n$.

Algorithm 2 Algorithme naïf

```
while ENDETTE() do
  POOR = []
  WEALTHIES = []
  for all v ∈ 𝒱 do
    if VALUE(v) < 0 then
      POOR ← {v} ∪ POOR
    else if val(v) > deg v then
      WEALTHIES ← {v} ∪ WEALTHIES
    end if
  end for
  if |WEALTHIES| > 0 then
    WEALTHIES_SORTED = SORT_BY_VALUE(WEALTHIES)
    v ← WEALTHIES_SORTED[0]
    GIVE(v)
  else
    POOR_SORTED = SORT_BY_VALUE(POOR)
    v ← POOR_SORTED[0]
    BORROW(v)
  end if
end while
```

Le premier algorithme a l'avantage de fonctionner le plus souvent sur des graphes aléatoires, tandis que le second effectue la plupart du temps une séquence solution plus courte que le premier.

4 Minimisation des séquences

La partie précédente fournie une séquence de coups qui résout le jeu. Je m'intéresse à présent à plusieurs moyens de minimiser cette séquence solution. Trois résultats permettent ceci.

Théorème 2. *L'ordre des coups n'importe pas.*

Démonstration. Chaque coup consiste en effet à ajouter ou à soustraire une certaine quantité (éventuellement nulle) à la valeur des sommets du graphe. L'addition étant commutative, il est possible de réordonner la séquence à souhait. ■

Ce résultat permet alors de réduire une séquence solution à un sous-ensemble de $V \times \mathbb{N}* \times \{-1, 1\}$, où pour chaque sommet est attribué un nombre d'emprunt et de prêts. Dans le cas de la figure 1, la séquence est représentée ainsi : $\big\{(b, 1, 1), (a, 1, -1)\big\}$: b effectue un seul prêt, et a effectue un seul emprunt. Si b effectuait deux prêts, la séquence serait représentée ainsi : $\big\{(b, 2, 1), (a, 1, -1)\big\}$. J'omets ici les sommets n'ayant fait aucun coup pour plus de clarté.

Théorème 3. *Un prêt et un emprunt effectué par le même sommet équivaut à ne rien faire.*

Ce dernier résultat permet d'annuler les emprunts et les prêts effectués par un même sommet, ce qui permet donc de réduire à nouveau la représentation d'une séquence solution, cette fois à un sous-ensemble de $V \times \mathbb{Z}$: à chaque sommet est associé un entier relatif z, négatif s'il effectue $|z|$ emprunts, et positif s'il effectue z prêts.

Enfin, voici un dernier résultat permettant d'éventuellement réduire une séquence solution indépendamment du graphe.

Théorème 4. *Un prêt est équivalent à une série d'emprunts.*

Démonstration. Soit v un sommet. En considérant le coup $(v, +1)$, la séquence de coups $((v', -1))_{v' \in V \smallsetminus \{v\}}$ est équivalente. En effet, chaque arête du graphe sert à faire transiter un même dollar dans les deux sens, excepté les arêtes adjacentes à v, qui ne servent qu'à faire transiter des dollars de v vers ses voisins. ■

Corollaire 4.1. *Réciproquement, un emprunt est équivalent à une série de prêts.*

Il est possible de généraliser les résultats précédents par le théorème suivant.

Théorème 5. *Soit $W \subset V$ un sous-ensemble des sommets d'un graphe effectuant chacun un prêt. Alors la séquence correspondante est équivalente à celle des sommets de $V \smallsetminus W$ effectuant chacun un emprunt.*

La démonstration est identique.

Les résultats précédents permettent alors de réduire assez facilement une séquence. En effet, en notant W les sommets effectuant au moins un prêt, si $|W| > |V|/2$, il suffit de remplacer un prêt de chaque sommet de W en emprunts effectués par les sommets de $V \smallsetminus W$. Il est ainsi possible de procéder ainsi jusqu'à ce que le nombre de sommets effectuant des prêts soit inférieur à $|V|/2$ et que le nombre de sommets effectuant des emprunts soit aussi inférieur à $|V|/2$. Ceci est possible car à chaque itération de ce processus, le nombre de coup diminue toujours d'au moins un.

On obtient alors une séquence solution plus courte, parfois optimale pour de petits graphes, de l'ordre de 5 à 10 sommets et de degré maximal 5 ou moins. Obtenir des séquences optimales sur des graphes plus grand est cependant beaucoup plus rare avec cette méthode et nécessite de s'intéresser à la structure même du graphe pour en tirer avantage et diminuer encore la séquence obtenue, ou en trouver une autre.

Bibliographie

[1] N.L.Biggs. *Chip-Firing and the Critical Group of a Graph : Journal of Algebraic Combinatorics 9* (1999), p25-45

[2] A. Björner, L. Lovász et P. W. Shor. *Chip-firing games on graphs : European Journal of Combinatorics 12* (1991), p283-291

[3] M. Baker et S. Norine *Riemann-Roch and Abel-Jacobi theory on a finite graph, Advances in Mathematics 215* (2007)

5 Annexe

5.1 Implémentation complète du code python

J'utilise ici l'environnement de développement Processing.py. Le programme ci-dessous permet de générer un graphe aléatoire ou de charger un graphe pondéré à partir d'un fichier, puis de le manipuler : il est possible de déplacer les sommets, de faire des prêts avec le clic gauche et des emprunts avec le clic droit sur un sommet choisi, de sauvegarder un graphe, la position de ses nuds et sa pondération, et enfin d'appliquer un algorithme de résolution au graphe. La séquence générée s'affiche à gauche, avec le nombre total de coups effectués en bas à gauche. La partie est une solution si le carré en bas à droite est vert. L'algorithme peut être choisi à l'aide des touches B et N du clavier. Deux autres algorithmes ont été fournis, l'un consistant seulement à donner et l'autre cherchant à alterner entre un emprunt et un prêt, ceux-ci sont cependant infructueux.

dollar_game.py

```python
import actions
import algorithm
import display
import globals
import graph

def setup():
    size(display.WindowSizeX, display.WindowSizeY)
    frameRate(25)

    # globals.graph_name = "graph_numberphile"
    # graph.load()

    graph.generateRandom(15, 4)
    display.setup_buttons()

def draw():
    clear()
    background(20)

    display.draw_actions()
    display.draw_transfers()
    display.draw_edges()
    display.draw_balanced()
    display.increment_transfer()
    for b in display.buttons:
```

```
        b.display()
    display.draw_selected_algorithm()

    if not globals.manual_mode:
        algorithm.selected_algorithm()()

def keyPressed():
    if key == 'a' or key == 'A':
        display.mode.actionLeft()
    if key == 'b':
        algorithm.select_prec()
    if key == 'n':
        algorithm.select_next()

def mouseClicked():
    if not globals.manual_mode:
        return
    for button in display.buttons:
        if button.is_mouse_inside():
            button.clicked = 5
            if mouseButton == LEFT:
                button.actionLeft()
            if mouseButton == RIGHT:
                button.actionRight()

def mouseWheel(event):
    display.actions_pos += event.getCount()
    if display.actions_pos >= len(globals.actions):
        display.actions_pos = len(globals.actions) - 1
    if display.actions_pos < 0:
        display.actions_pos = 0

targetButton = None
def mouseDragged():
    global targetButton
    if targetButton:
        if targetButton.is_mouse_inside():
            targetButton.actionDrag(targetButton)
        else:
            targetButton = None
    else:
        for button in display.buttons:
            if button.is_mouse_inside():
                targetButton = button
                button.actionDrag(button)
                break
```

actions.py

```python
from copy import copy
import display
from   copy import deepcopy
import globals
import json
import os

def nullAction(*args, **kwargs):
    pass

def changeMode():
    globals.manual_mode = not globals.manual_mode
    if globals.manual_mode:
        display.mode_description[0] = 'manual'
    else:
        display.mode_description[0] = 'automatic'

def reset():
    globals.actions = []
    for v in globals.graph['vertices']:
        v['value'] = copy(globals.start['vertices'][v['index']]['value'])

def saveGraph():
    number = 0
    while( os.path.isfile("saved_graph_" + str(number) + '.json')):
        number += 1
    temp_graph = deepcopy(globals.graph)
    for v in temp_graph['vertices']:
        del v['action_give']
        del v['action_take']
    with open("saved_graph_" + str(number) + '.json', 'w') as stream:
        json.dump( temp_graph, stream )

def drag(button):
    if isinstance(button, display.vertexButton):
        button.vertex['pos'] = [mouseX, mouseY]
    elif isinstance(button, display.rectButton):
        button_width = abs( button.rdCornerX - button.luCornerX )
        button_height = abs( button.rdCornerY - button.luCornerY )
        button.luCornerX = mouseX - button_width
```

```
        button.luCornerY = mouseY - button_height
        button.rdCornerX = mouseX + button_width
        button.rdCornerY = mouseY + button_height
```

algorithm.py

```python
# -*- coding: utf-8 -*-
import actions

import algo_alternate
import algo_only_borrow
import algo_only_give
import algo_naive

algorithms = {
    "": (lambda: None),
    "naive": algo_naive.algo,
    "only borrow": algo_only_borrow.algo,
    "only_give": algo_only_give.algo,
    "alternate": algo_alternate.algo
}

selected = 0

def select_prec():
    global selected
    selected -= 1
    selected = max(selected, 0)

def select_next():
    global selected
    selected += 1
    selected = min(selected, len(algorithms.keys()) - 1)

def selected_key():
    return list(algorithms.keys())[selected]

def selected_algorithm():
    return algorithms[selected_key()]
```

minimization.py

```python
import globals

def minimize():
    sz = len(globals.graph['vertices'])
    seq = [0] * sz
    for v, move in globals.actions:
        seq[v] += move
    print(seq, sum([abs(e) for e in seq]))

    while len([e for e in seq if e > 0]) > sz / 2:
        seq = [e - 1 for e in seq]
        print(seq, sum([abs(e) for e in seq]))

    while len([e for e in seq if e < 0]) > sz / 2:
        seq = [e + 1 for e in seq]
        print(seq, sum([abs(e) for e in seq]))

    actions = []
    for i, v in enumerate(seq):
        for _ in range(abs(v)):
            actions.append( (i, (-1 if v < 0 else 1)) )
    globals.actions = actions
```

display.py

```python
import actions
import algorithm
import events
import globals
import graph
import minimization

# Window parameters

Border = 50
WindowSizeX = 2100
WindowSizeY = 1300
LeftBorder = Border
RightBorder = WindowSizeX - Border
TopBorder = Border
BottomBorder = WindowSizeY - Border


# Drawing context
actions_pos = 0
transfers = [] # time between 0 and 10
buttons = []
mode = None
mode_description = ['manual']
reset = None

# classes

class rectButton:
    def __init__(self, luCornerX, luCornerY, rdCornerX, rdCornerY,
                 actionLeft = None, actionRight = None, actionDrag = None,
                 linkedData = None):
        self.luCornerX = luCornerX
        self.luCornerY = luCornerY
        self.rdCornerX = rdCornerX
        self.rdCornerY = rdCornerY
        self.actionLeft = actionLeft if actionLeft else actions.nullAction
        self.actionRight = actionRight if actionRight else actions.nullAction
        self.actionDrag = actionDrag if actionDrag else actions.nullAction
```

```
        self.linkedData = linkedData
        self.clicked = False

    def display(self):
        strokeWeight(4)
        stroke(128)
        fill(128 if self.clicked else 25)
        rectMode(CORNERS)
        rect(self.luCornerX, self.luCornerY, self.rdCornerX, self.rdCornerY)
        fill(200)
        textAlign(CENTER, CENTER)
        textSize(30)
        text(self.linkedData[0], (self.luCornerX + self.rdCornerX) / 2, (self.luCornerY + self.rdCornerY) / 2)
        self.clicked = max(self.clicked - 1, 0)

    def is_mouse_inside(self):
        return self.luCornerX < mouseX and self.luCornerY < mouseY and mouseX < self.rdCornerX and mouseY < self.rdCornerY

class vertexButton:
    def __init__(self, vertex, radius,
                 actionLeft = None, actionRight = None, actionDrag = None):
        self.vertex = vertex
        self.radius = radius
        self.actionLeft = actionLeft if actionLeft else actions.nullAction
        self.actionRight = actionRight if actionRight else actions.nullAction
        self.actionDrag = actionDrag if actionDrag else actions.nullAction

    def display(self):
        fill(128)
        ellipse(self.vertex['pos'][0], self.vertex['pos'][1], 30, 30)
        fill(200)
        textSize(20)
        textAlign(CENTER, CENTER)
        text(str(self.vertex['value']), self.vertex['pos'][0], self.vertex['pos'][1])
        fill(200, 20, 20)
```

```python
        textSize(17)
        textAlign(CENTER, TOP)
        text(str(self.vertex['index']), self.vertex['pos'][0],
        ↪  self.vertex['pos'][1] - 30)

    def is_mouse_inside(self):
        return sqrt((pmouseX - self.vertex['pos'][0])**2 +
        ↪  (pmouseY - self.vertex['pos'][1])**2) < self.radius

# setup

def setup_buttons():
    global mode, reset
    mode = rectButton(0, 0, WindowSizeX / 4, 50,
                      actionLeft = actions.changeMode,
                      linkedData = mode_description)
    buttons.append(mode)
    reset = rectButton(WindowSizeX / 4, 0, 2 * WindowSizeX / 4,
    ↪  50,
                       actionLeft = actions.reset,
                       linkedData = ["reset"])
    buttons.append(reset)
    saveB = rectButton(2 * WindowSizeX / 4, 0, 3 * WindowSizeX /
    ↪  4, 50,
                       actionLeft = actions.saveGraph,
                       linkedData = ["save"])
    buttons.append(saveB)
    minim = rectButton(3 * WindowSizeX / 4, 0, 4 * WindowSizeX /
    ↪  4, 50,
                       actionLeft = minimization.minimize,
                       linkedData = ["minimize"])
    buttons.append(minim)

    for i, v in enumerate(globals.graph['vertices']):
        left, right = graph.nodeActionGen(i)
        button = vertexButton(v, 30, actionLeft = left,
        ↪  actionRight = right, actionDrag = actions.drag)
        buttons.append(button)

# draw methods

def increment_transfer():
    global transfers
    new_transfers = []
    for c in transfers:
        i, j, t = c
```

```
            if t < 10:
                new_transfers.append((i, j, t + 1))
        transfers = new_transfers

def draw_selected_algorithm():
    fill(200)
    textAlign(CENTER, CENTER)
    textSize(20)
    text(algorithm.selected_key(), WindowSizeX / 16, 50 / 2)

# each action is a button of height 40 and width 100
def draw_actions():
    rectMode(CORNERS)
    strokeWeight(4)
    stroke(128)
    textAlign(CENTER, CENTER)
    try:
        actions_to_draw = globals.actions[actions_pos:
         ↪  actions_pos + 31]
    except:
        actions_to_draw = globals.actions[actions_pos:]
    for i, a in enumerate(actions_to_draw):
        v, move = a
        move_string = u'<->' if move > 0 else u'>-<'
        fill(25)
        rect(0, i * 40 + 60, 100, i * 40 + 100)
        textSize(20)
        fill(200)
        text(move_string, 30, i * 40 + 80)
        textSize(30)
        fill(200, 20, 20)
        text(str(v), 70, i * 40 + 80)

    fill(128)
    rect(0, 30 * 40 + 60, 100, 30 * 40 + 100)
    fill(255)
    text(str(len(globals.actions)), 50, 30 * 40 + 80)


def draw_edges():
    strokeWeight(4)
    stroke(128)
    for e in globals.graph['edges']:
        i, j = tuple(e)
        ix, iy = tuple(globals.graph['vertices'][i]['pos'])
        jx, jy = tuple(globals.graph['vertices'][j]['pos'])
```

```
        line(ix, iy, jx, jy)

def draw_transfers():
    fill(255, 128, 0)
    ellipseMode(RADIUS)
    for (i, j, t) in transfers:
        ix, iy = tuple(globals.graph['vertices'][i]['pos'])
        jx, jy = tuple(globals.graph['vertices'][j]['pos'])
        x = (jx - ix) * t / 10 + ix
        y = (jy - iy) * t / 10 + iy
        ellipse(x, y, 10, 10)

def draw_balanced():
    if graph.balanced():
        fill(0, 255, 0)
    else:
        fill(255, 0, 0)

    strokeWeight(0)
    rectMode(CORNERS)
    rect(WindowSizeX - 50, WindowSizeY - 50, WindowSizeX,
    ↪  WindowSizeY)
```

globals.py

```
# General parameters

graph_suffix = 'peterson'
graph_name = 'graph_' + graph_suffix
manual_mode = True

# Data

graph = None
start = None
actions = []
```

graph.py

```python
import actions
from   copy import deepcopy
import display
import globals
from   math import exp, sqrt
import random

import json

# actions generator for graph nodes

def nodeActionGen(i):
    def funcLeft(): # gives to neighbors of i one dollar
        neighbors = neighbors_of(i)
        globals.graph['vertices'][i]['value'] -= len(neighbors)
        for n in neighbors:
            globals.graph['vertices'][n]['value'] += 1
            display.transfers.append((i, n, 0))
        globals.actions.append((i, 1))

    def funcRight(): # takes from all neighbors of i one dollar
        neighbors = neighbors_of(i)
        globals.graph['vertices'][i]['value'] += len(neighbors)
        for n in neighbors:
            globals.graph['vertices'][n]['value'] -= 1
            display.transfers.append((n, i, 0))
        globals.actions.append((i, -1))

    globals.graph['vertices'][i]['action_give'] = funcLeft
    globals.graph['vertices'][i]['action_take'] = funcRight
    return funcLeft, funcRight

def optimizeGraphDisplay(graph):
    minX = min([v['pos'][0] for v in graph['vertices']])
    minY = min([v['pos'][1] for v in graph['vertices']])
    maxX = max([v['pos'][0] for v in graph['vertices']])
    maxY = max([v['pos'][1] for v in graph['vertices']])
    graph_width = maxX - minX if maxX - minX else 2
    graph_height = maxY - minY if maxY - minY else 2
    for v in graph['vertices']:
        x = int( (v['pos'][0] - minX) * (2050 - 150) / graph_width + 150 )
        y = int( (v['pos'][1] - minY) * (110 - 1250) / graph_height + 1250 )
```

```python
        v['pos'] = [x,y]

def load():
    #global graph
    with open(globals.graph_name + '.json', 'r') as stream:
        graph = json.loads( stream.read() )
    for i, v in enumerate(graph['vertices']):
        v['index'] = i
    optimizeGraphDisplay(graph)
    globals.start = graph
    globals.graph = deepcopy(graph)

def generateRandom(nodes, maxEdges, max_debt = -10, strongly_winnable = True):
    graph = dict()
    graph['vertices'] = []
    graph['edges'] = []
    for i in range(nodes):
        node = dict()
        node['index'] = i
        node['pos'] = [random.randint(0, 100), random.randint(0, 100)]
        node['value'] = max_debt
        graph['vertices'].append(node)

    for v in graph['vertices']:
        # each edge can be choosed two times (from two nodes)
        edges = random.randint(1, maxEdges // 2)
        other_nodes = list(range(nodes))
        other_nodes.remove(v['index'])
        neighbors = [ random.choice(other_nodes) for _ in range(edges) ]
        graph['edges'].extend([(v['index'], n) for n in neighbors])

    optimizeGraphDisplay(graph)

    if strongly_winnable:
        g = euler_number(graph)
        N = g - nodes * max_debt
        for _ in range(N):
            random.choice(graph['vertices'])['value'] += 1
    else:
        for v in graph['vertices']:
            v['value'] = random.randint(-5, 5)
    globals.start = graph
```

```
    globals.graph = deepcopy(graph)

###################
### Graph utils ###
###################

def euler_number(graph):
    return len(graph['edges']) - len(graph['vertices']) + 1

def balanced():
    return all([v['value'] >= 0 for v in globals.graph['vertices']])

def neighbors_of(i):
    return [ a if b == i else b for [a,b] in globals.graph['edges'] if a == i or b == i ]

def degree_of(i):
    deg = 0
    for [a, b] in globals.graph['edges']:
        if a == i or b == i:
            deg += 1
    return deg
```

algo_naif.py

```python
import actions
import display
import globals
import graph

def debts():
    debt = False
    for v in globals.graph['vertices']:
        if v['value'] < 0:
            debt = True
            break
    return debt

def rich_vertices():
    vertices = []
    for i, v in enumerate(globals.graph['vertices']):
        if v['value'] >= graph.degree_of(i):
            vertices.append(i)
    return vertices

def poor_vertices():
    vertices = []
    for i, v in enumerate(globals.graph['vertices']):
        if v['value'] < 0:
            vertices.append(i)
    return vertices

def algo():
    if not debts():
        display.mode.actionLeft()
        print("Success !")
        print(len(globals.actions))
        return

    # spread wealth

    wealthies = rich_vertices()
    if len(wealthies):
        wealthies = sorted(wealthies, key=lambda i:
        ↪  globals.graph['vertices'][i]['value'])
        i = wealthies[-1]
        globals.graph['vertices'][i]['action_give']()
    else:
        poor = poor_vertices()
```

```
        poor = sorted(poor, key=lambda i:
        ↪  globals.graph['vertices'][i]['value'])
        i = poor[0]
        globals.graph['vertices'][i]['action_take']()
```

algo_only_borrow.py

```python
import actions
import display
import globals
import graph

def debts():
    debt = False
    for v in globals.graph['vertices']:
        if v['value'] < 0:
            debt = True
            break
    return debt

def poor_vertices():
    vertices = []
    for i, v in enumerate(globals.graph['vertices']):
        if v['value'] < 0:
            vertices.append(i)
    return vertices

def algo():
    if not debts():
        display.mode.actionLeft()
        print("Success !")
        print(len(globals.actions))
        return

    poor = poor_vertices()
    poor = sorted(poor, key=lambda i: globals.graph['vertices'][i]['value'])
    i = poor[0]
    globals.graph['vertices'][i]['action_take']()
```

algo_only_give.py

```python
import actions
import display
import globals
import graph

def debts():
    debt = False
    for v in globals.graph['vertices']:
        if v['value'] < 0:
            debt = True
            break
    return debt

def rich_vertices():
    vertices = []
    for i, v in enumerate(globals.graph['vertices']):
        if v['value'] >= 0:
            vertices.append(i)
    return vertices

def algo():
    if not debts():
        display.mode.actionLeft()
        print("Success !")
        print(len(globals.actions))
        return

    # spread wealth

    wealthies = rich_vertices()
    if len(wealthies):
        wealthies = sorted(wealthies, key=lambda i: globals.graph['vertices'][i]['value'])
        i = wealthies[0]
        globals.graph['vertices'][i]['action_give']()
```

algo_alternate.py

```python
import actions
import display
import globals
import graph

from random import choice

wealthy_turn = True

def debts():
    debt = False
    for v in globals.graph['vertices']:
        if v['value'] < 0:
            debt = True
            break
    return debt

def rich_vertices():
    vertices = []
    for i, v in enumerate(globals.graph['vertices']):
        if v['value'] >= 0:
            vertices.append(i)
    return vertices

def poor_vertices():
    vertices = []
    for i, v in enumerate(globals.graph['vertices']):
        if v['value'] < 0:
            vertices.append(i)
    return vertices

def algo():
    global wealthy_turn
    if not debts():
        display.mode.actionLeft()
        print("Success !")
        print(len(globals.actions))
        return

    # spread dollars

    if wealthy_turn:
        wealthies = rich_vertices()
        if len(wealthies) == 0:
```

```
                return
            wealthies = sorted(wealthies, key=lambda x: globals.graph['vertices'][x]['value'])
            i = wealthies[-1]
            globals.graph['vertices'][i]['action_give']()
        else:
            poor = poor_vertices()
            i = choice(poor)
            globals.graph['vertices'][i]['action_take']()

        wealthy_turn = not wealthy_turn
```