Hashiwokakero Solver

Par Samuel Vicart

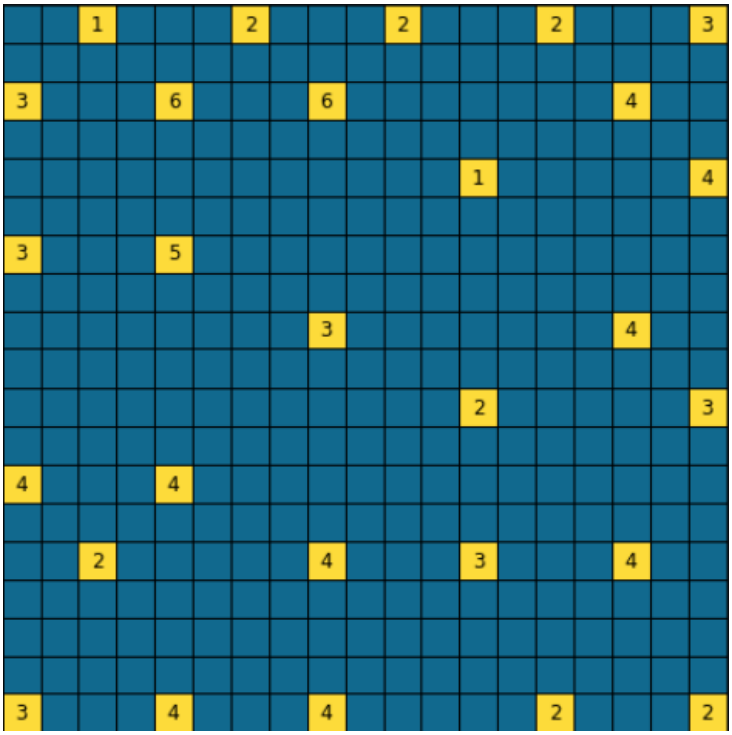
Table des matières

I/ Présentation du jeu

Commençons se rapport par une présentation du Hashiwokakero.

Il s'agit d'un jeu de logique créé par Nikoli, un éditeur de jeux Japonais. Les règles et le but de ce jeu sont très simples mais peuvent mener à des puzzles extrêmement complexes. Pour bien comprendre le fonctionnement, je vous propose de faire une partie de test en énonçant au fur et à mesure les règles du jeu.

Voici un plateau de jeux de Hashiwokakero :



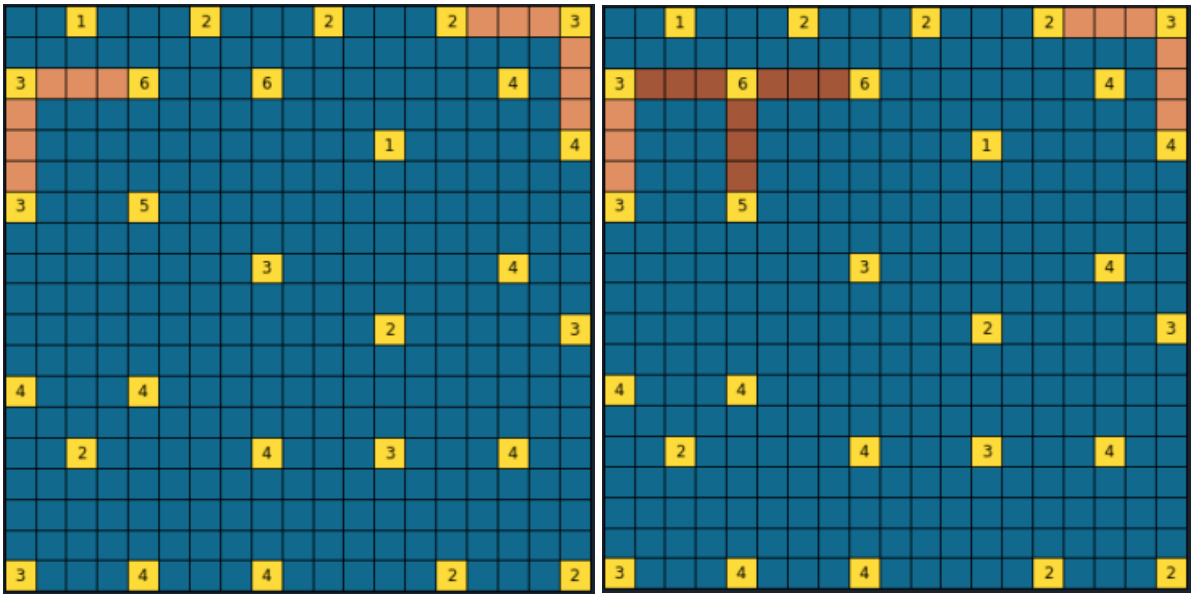
On y observe pour l'instant deux éléments :

- Les carrés bleu d'une part,  sont des espaces vides. Ils pourront par la suite être remplacer par des ponts. Nous verrons cela juste après.

- Les carrés jaunes d'autre part, 

sont appelés des îles. Ce sont les piêces maitresses du jeux. On trouve sur chaque île un nombre allant de 1 à 8. Il s'agit du nombre de voies qu'elle doivent acceuillir. Comme dit précédement, nous aborderons les ponts et les voies dans la prochaines partie. Les îles sont également statiques. C'est à dire que nous n'avons pas la possibilité de les bouger.

Voyons maintnenant les premières étapes de la résolution de ce puzzle :



On obserevs deux nouveaux éléments :

Des carrés marron clair,  et des carrés marron foncé  . Ce sont des ponts (en bois). Ils vont nous servir, comme vous pouvez l'observer, à relier des îles entre elles. Le ponts les plus claires sont des ponts à une vois. Les plus foncés sont plus résistant, ce sont des ponts à deux vois. Ces deux sortes de ponts doivent repecté les mêmes règles :

- Il ne doivent jamais se croiser ni se superposer.

- Ne doivent jamais passer au dessus d'une île.

- Doivent toujours relier deux îles entres elles.

- ne peuvent être construit qu'horizontalement ou verticallement .

- Une ligne de carrés correspont à un pont à 1 ou 2 voies celon sa couleur.

Pour finir, les ponts peuvent et doivent être placés et déplacés par le joueur.

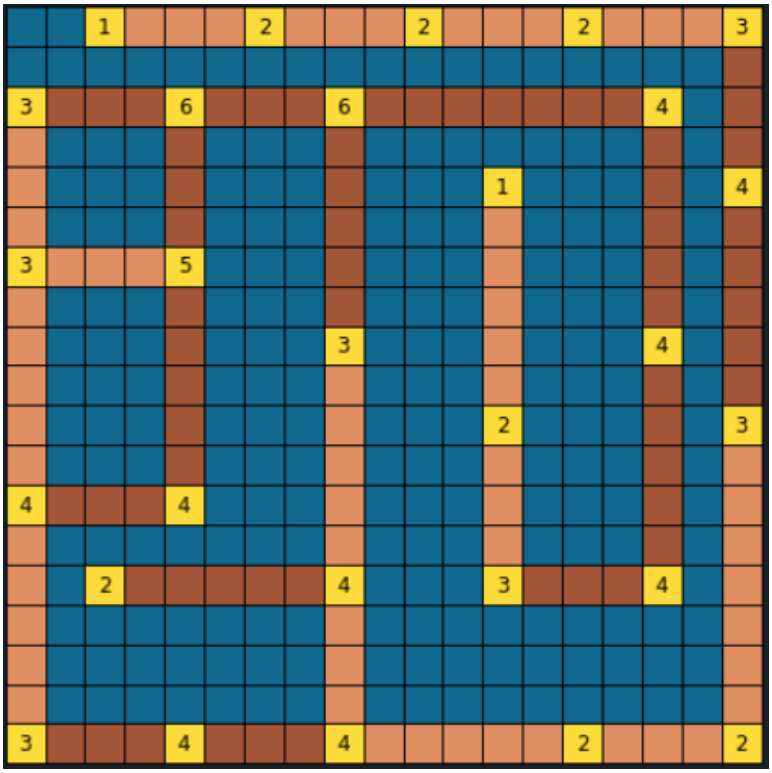
Parlons maintenant du but du jeu. Il est plus ou moins simple et se décompose en deux règles :

- Toutes les îles doivent être reliées entre elles grâce au pont en formant un groupement unique d'ile .

- Chaque île doit construire le nombre de voies indiqué en son centre. Pour prendre un exemple, l'île à 6 points et reliée par 3 ponts marrons foncé respecte déjà cette règle (car 3 ponts x 2 voies = 6)

Ces deux règles peuvent paraitre simples voir enfantines mais elles peuvent donner lieu à des casse-tête extrêmement complexes.

Voyons pour finir le puzzle complètement résolu :



On observe que toutes les règles du jeu sont bien respectées et que le but à bien été atteint.

A vous de jouer maintenant !

II/ Notice d'utilisation du programme « Hashiwokakero Solver »

A. Lancer l'application

Avant tout, commencez par dézipper le dossier « ressource » fourni dans un dossier préalablement créé.

Pour utiliser l'application vous disposez de plusieurs possibilités :

Tout d'abord un exécutable est fourni dans le dossier « ressources ». Il suffit alors de lancer

« Chashiwokakero_Solver.exe» se trouvant dans le dossier « dist » .

Ce dernier peut prendre un peut de temps pour charger les ressources. Il est donc normal d'attendre une a deux minute lors de son ouverture avant de voir du texte apparaître.

Cependant cet exécutable a été compilé pour fonctionner sur des machines ayant Windows10 comme système d'exploitation. Je ne suis donc pas dans la possibilité de vous assurer le bon fonctionnement de ce dernier sur des machines fonctionnant sur Linux ou autres systèmes d'exploitation.

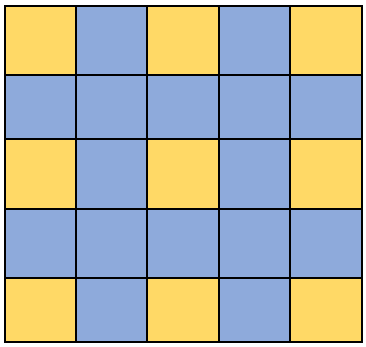
L'autre possibilité est de charger le code source dans un utilitaire tel que « Spyder ».

Pour être sur d'avoir les librairies nécessaires au bon fonctionnement de cette application, je préconise d'installer un environnement tel qu'« anaconda » proposant notamment le logiciel « spyder » .

B. Utiliser l'application

L'application que je vous propose possède une interface graphique. Vous pouvez choisir d'importer votre graphe ou d'utiliser des graphes pré-importés de difficulté différentes. Ces derniers seront résolus plus ou moins rapidement selon leur niveau de difficulté.

Veuillez respecter cette forme d'input :



 Est l'emplacement d'une île potentielle

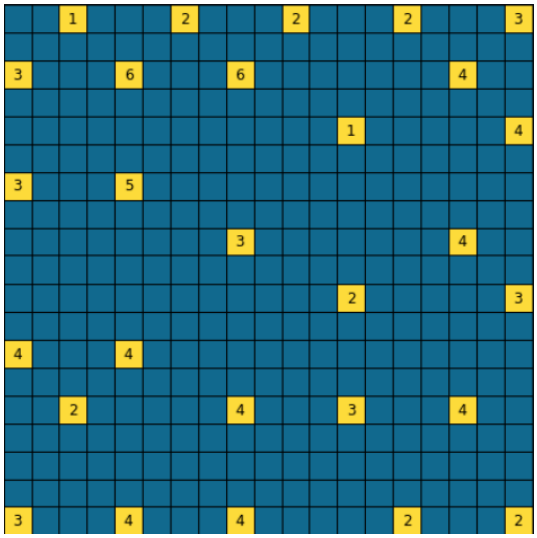
 Est l'emplacement d'un 0 obligatoire

Exemple :

1	0	2	0	0
0	0	0	0	0
0	0	2	0	2
0	0	0	0	0
0	0	2	0	3

Sachez par ailleurs que les exemples ont été choisis totalement aléatoirement et créés par le site : https://www.kakuro-online.com/hashi/

Au lancement de l'application, une fois le chargement terminé, un indice de commande devrait s'ouvrir. Après une courte présentation, il vous sera demandé de choisir entre 3 puzzles différents par le biais d'un input. Il vous suffira alors de taper 1, 2 ou 3 en fonction de la difficulté du puzzle que vous souhaitez résoudre. Une fois votre choix effectué veuillez taper sur entrer pour envoyer l'information au programme. Une phrase accompagnée d'un graphe tel que celui-ci-dessous devrait alors s'afficher.

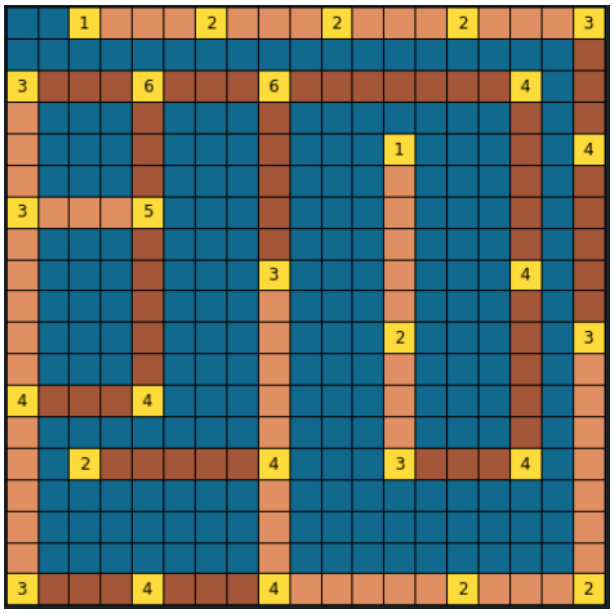


Il s'agit du puzzle que le programme va par la suite résoudre.

Pour lancer la résolution du puzzle, vous devez tout d'abord fermer le graphe qui est apparu puis appuyer sur entrer lorsqu'il vous sera indiqué de le faire.

Lorsque le calcul de la solution est effectué des informations peuvent apparaitre dans la console. Puis lorsque le programme aura trouvé une solution au puzzle, il l'affichera alors dans une nouvelle page.

Voila à quoi cela pourra alors ressembler :



Pour des raisons de rapidité de calcul j'ai préféré ne pas afficher les étapes intermédiaires de la résolution du puzzle. Elle vous seront-elles aussi détaillées lors de la soutenance.

Des informations complémentaires sur les calculs effectués seront également affiché dans la console.

Pour finir, vous n'avez plus qu'a appuyer sur entrer pour quitter l'application.

III/ Fonctionnement de l'application

Tout d'abord j'aimerais préciser que j'ai choisis d'utiliser python comme langage car il me semblait le plus adapter pour le calcul matriciel et la gestion de données.

A. Les structures de données

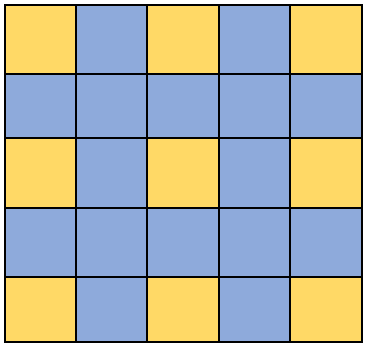
Pour bien aborder le fonctionnement de l'application, je vous propose de découvrir les différentes structures de données que l'application utilise.

Il y à deux structures de données principales utilisées dans mon application.

La première représente l'input et l'output des puzzles. Il s'agit d'une matrice numpy. Voici sa forme :

Array[int][int]

Voici la règle qu'elle respecte (elle est également énoncée plus haut)



Est l'emplacement d'une île potentielle (int >= 0)

Est l'emplacement de l'eau (int = 0) ou d'un pont (-3 > int < 0)

Voici un exemple d'un puzzle résolu sous la forme de cette matrice :

1	-1	2	0	0
0	0	-1	0	0
0	0	2	0	2
0	0	-1	0	-2
0	0	2	-1	3

La deuxième structure de donnée est une structure comprenant les informations des iles et des relations entre ces dernières. Il s'agit concrètement du graphe du puzzle sous une forme plus simple et pratique pour les calculs. Voici sa forme pour N îles :

îles	0	1	…	N - 1
Coordonnées	tuple(int,int)			
Points	int			
Voisins	array[int]			
Ponts	array[array[int,int]]			

Coordonnées :

Tuple d'entier positif comportant la coordonnée en x et la coordonnée en y de l'ile : (1, 6)

Points (ou poids) :

Entier représentant le nombre de voies que doit avoir l'ile. Ex : 3

Voisins :

Tableau répertoriant les index des iles voisines (une île est dite voisines si et seulement si elle se situe à coté de l'île en question ET qu'aucun pont est situé entre les deux îles).

Ponts :

Tableau comportant un maximum de 4 tableaux de ponts. Ex : [[5,2][6,1][1,3]

Tableau de ponts :

Un tableau comporte deux entiers. Le premier représente l'index de l'île vers lequel le pont a été créé. L'autre représente le nombre de voies sur ce pont. Ex : [5, 2]

B. Les règles établies

Pour créer mon algorithme j'ai décidé de créer des règles. Ces règles sont à appliquer à chaque boucle et à chacune des îles. Elles ont pour but de placer des ponts dont on est sûr qu'ils seront bien placés.

Commençons par quelques définitions :

- Le point d'une île est son poids. C'est-à-dire le nombre de pont qu'elle doit accueillir.

- Le point restant d'une île est le point auquel on soustrait le nombre de pont accueillit.

- Une île est fermée lorsqu'elle n'a plus de point restant.

- Une île est ouverte si elle n'est pas fermée

- Un pont est non lié si une île et son voisin peuvent mais n'ont pas encore lié ce pont. Cela implique que les deux îles ne doivent être ouverte.

- Un pont est restant si une île et son voisin peuvent mais n'ont pas encore lié ce pont. Cela n'implique PAS que les deux îles ne doivent être ouverte.

Attentions les définitions de pont restant et de pont non lié sont proches mais bien distinctes !

Voici maintenant les règles fondamentales que j'ai trouvé et ai intégré à mon programme :

1. Si une île possède :

N points restants

N ponts Non-liés

On peut effectuer toutes ses liaisons.

2. Si une île possède :

N points restants

N + 1 ponts restants

On peut lier tous ses voisins non liés avec 1 ponts.

3. Deux îles sont des voisins si :

Elles ont au moins une coordonnée en commun et qu'aucun pont ne se situe entre elles.

3.b Deux îles apriori voisines de point 1 ne le sont pas.

4. Si :

- Une ile a 1 ou 2 points restants

- Un seul voisin encore ouvert

On peut lier l'île a avec 1 ou 2 ponts. S'il reste un point à l'ile après cette action cela signifie que les chois aléatoire pris précédemment ne sont pas bons.

Ces quatre règles sont réellement le fondement de mon programme. Voyons maintenant comment elles ont été utilisées et crées.

C. Les algorithmes principaux

Comme dit précédemment, le programme réside sur des règles qui sont testés et appliquées à répétition. Si jamais l'application de ces règles n'ont plus aucun effet on ajoute alors un pont aléatoirement à deux voisins possédant un pont non lié. (Voir chapitre III partie b pour la définition de pont non lié). Voici cet algorithme principal :

```
    memory = []
    memM = np.zeros(np.shape(M))
    loops = 0
    guesses = 0
    finish = 0
    guessIndex = 0
    precedentGuesses = []

    tant que finish != 1:
        tant que hashcode(memM) != hashcode(M):
            tant que hashcode(memM) != hashcode(M):
                fourthRuleFlag = 0

                memM = M.copy()
                loops += 1
                draw(GtoM(G, M))
                pour chaque île:

                    faire: 1_ère_règle

                    pour chaque île:
                        faire: 3 ème règle

                    faire: 2_ème_règle

                    pour chaque île:
                        faire: 3_ème_règle

                    si 4 ème_B_règle   > 0:
                        fourthRuleFlag = 1
                        sortir de la boucle

                    pour chaque île:
                        faire: 3_ème_règle

                si fourthRuleFlag == 1:
                    sortir de la boucle

            si fourthRuleFlag == 1:
                sortir de la boucle

            ajouter_EnMemoire(Graphe) //Cette méthode ne met en mémoire que lors de son premier appel
            Ajouter_Pont_Aleatoire(G , guessIndex, precedentGuesses)

            pour chaque île:
                faire: 3_ème_règle


        si G_Est_Connexe(Graphe) == 1 and Toutes_iles_fermees == 1:

            finish = 1
        si non:
            guessIndex += 1
            G = recuper_en_memoire(memory)
            M = GtoM(G,M)
```

Il parait complexe à première vu mais ne l'est pas tant que ça lorsqu'on se penche réellement dessus.

Laissez-moi vous l'expliquer simplement :

On a premièrement une boucle infinie qui nous fait répéter l'action tant qu'une solution n'est pas trouvée.

Puis nous avons une boucle qui fait un retour en arrière si tous les ponts ont été placé mais que la solution n'a pas été trouvé.

Lorsqu'on retourne en arrière, on retourne au dernier état du graphe ou l'on est sûr que tout est correcte. C'est-à-dire avant qu'un quelconque choix hasardeux n'ai été pris.

Puis on arrive dans la boucle précédemment expliqué dans laquelle on exécute pour chaque île les règles. Vous remarquerez que la règle numéro 3 est exécutée après chaque autres règle et pour toutes île. C'est par ce que tout changement dans le graphe nécessite de recalculer tous les voisins.

Dans cette boucle, si une erreur est détectée, on sort des boucles pour directement effectuer un retour en arrière.

Enfin, si on a positionné tous les ponts et que les règles de fin sont respectées le programme s'arrête et l'on sort de la boucle infinie. La solution vient d'être trouvée

La complexité de cet algorithme n'est pas connue car une partie d'aléatoire réside pour sortir de la boucle infinie.

Etudions maintenant les 2 premières règles :

```python
def firstRule(G,ile):

    if getPointsRestant(G,ile)  == getNbPontsNonLie(G, ile) and getPointsRestant(G, ile) != 0 :
        for voisin in getVoisins(G, ile):
            if getPointsRestant(G, voisin) > 0:
                for i in range(min(getPointsRestant(G, voisin), 2, getPointsRestant(G, ile))):
                    addPont(G, ile, voisin)


#rule n°2
def secondRule(G,ile):

    nbPontAUneBranche = sum([ponts[1] for ponts in getPonts(G, ile) if ponts[1] == 1])
    if getPointsRestant(G,ile) + 1 == getNbPontsRestant(G, ile) and nbPontAUneBranche == 0:

        for voisin in getVoisinsNonLie(G, ile):
            addPont(G, ile, voisin)

```

Je me permets de les laisser en python car ce langage est proche d'un langage universel et les fonctions utilisées selon moi assez explicites. Si vous avez des questions, j'aurais le plaisir d'y répondre lors de la soutenance. Je fais également ce choix car cela permet de laisser la couleur au texte et donc une meilleure lisibilité.

On voit que ces algorithmes sont très simples, dans le meilleure des cas ils sont en temps constant. Dans le pire des cas ils peuvent avoir une complexité de :

O(8) pour N étant le nombre d'îles dans le graphe pour la première règle et O(4) pour la seconde.

La règle numéro 4 est similaire dans son fonctionnement et sa complexité. Je ne l'aborderai donc pas ici.

Abordons plutôt la règle numéro 3 maintenant :

```
def thirdRule(G,ile):
    """
    Parameters
    ----------
    G : Graphe
        Le graphe a modifier.
    ile : int
        l'ile sur laquelle tester les voisin.

    Returns
    -------
    None.
    """
    coordsIle = getCoords(G,ile)
    #Pour tous ses voisins:
    for voisin in getVoisins(G, ile):
        coordsVoisin = getCoords(G,voisin)
        flag = 0
        #on test pour toutes les iles..
        for k in getIles(G):
            test = getCoords(G,k)
            #et leurs voisins liés
            for l in getPonts(G, k):
                testV = getCoords(G,l[0])
                #si il y a un nouveau ponts qui séparre les iles  (if complexe mais fonctionnel)
                if (coordsIle[0] < test[0] and test[0] < coordsVoisin[0] and test[1] < coordsIle[1] and testV[1] > coordsIle[1] ):
                    flag = 1
                if (coordsIle[0] > test[0] and test[0] > coordsVoisin[0] and test[1] > coordsIle[1] and testV[1] < coordsIle[1]):
                    flag = 1

                if (coordsIle[1] < test[1] and test[1] < coordsVoisin[1] and test[0] < coordsIle[0] and testV[0] > coordsIle[0]):
                    flag = 1

                if (coordsIle[1] > test[1] and test[1] > coordsVoisin[1] and test[0] > coordsIle[0] and testV[0] < coordsIle[0]):
                    flag = 1

        #Si c'est le cas on retire le voisin
        if flag == 1:
            G[2][ile].remove(voisin)
```

Cette règle est plus complexe et nécessite 4n fois plus de calcul que les précédentes.

Regardons ensemble son fonctionnement :

Tout d'abord on cherche tous les voisins de l'île.

Puis on va faire une série de test sur cette île. Il s'agit de test comparant les coordonnées des autres îles avec celle de l'ile de base et d'un de ses voisins.

Si l'un de ces tests se révèle vrai alors cela signifie qu'un pont à été construit entre l'île et son voisin.

On retire alors pour l'île son voisin.

La complexité de cette méthode est bien plus élevée que celles des 3 précédentes. Au minimum elle est de O(1) (si elle n'a pas de voisin mais ce n'est pas possible puisque toutes les îles ont un voisin). Au maximum elle est de O(12n) ou n est le nombre d'île du graphe.

Pour finir j'aimerais ajouter que j'avais mis en place une mémoire sur les choix aléatoire effectuée pour ne pas répéter des choix précédemment effectués. Je l'ai d'ailleurs laissé dans mon code source. Après quelques tests, je me suis aperçu qu'une telle mémoire était inutile car les chances d'effectuer deux

mêmes choix entre deux retours en arrière sont infimes surtout lorsque l'ont travail avec des matrices complexes. Je l'ai donc désactivé.

Pour comprendre exactement comment fonctionne mon programme je vous invite à regarder le code source

IV/ Problèmes rencontrées

J'ai du relevé un bon nombre de défis pour mener à bien ce projet et réussir à créer un programme fonctionnel. J'ai également buté sur plusieurs problèmes et ai du à plusieurs reprises recommencer mon programme entièrement.

Le premier problème que j'ai rencontré fut sur le choix de la structure de donnée pour le graphe. J'ai premièrement décidé de n'utiliser que la matrice python. Malheureusement j'ai vite compris que cette solution était loin d'être idéale car elle ne permettait pas de garder en mémoire facilement les liaisons entre les différentes iles (les ponts et les voisins). J'ai cependant gardé cette solution pour procéder à mon affichage et également à l'input.

Le second, tout aussi important et problématique fut sur les règles. J'avais, avant d'implémenter ces règles-ci créé une multitude d'autre règle que je pensais vraie. Cependant après avoir bien avancé sur mon projet j'ai finalement remarqué que ces règles n'étaient finalement pas vraies dans tous les cas possibles. Cela a donc compromis entièrement la version deux de mon programme.

Voici l'exemple d'une règle fausse :

« **1.** Si une île possède au moins autant de points que de voisins on peut affecter à chaque voisin direct ayant également plus de points que de voisins un pont. »

J'ai donc dû repenser mon programme et me reposer pour déterminer de nouvelles règles.

Je pense maintenant que mon programme est capable de trouver une solution à n'importe quel graphe. Cependant il se peut que cette solution soit trouvée en un temps considérable selon la taille du graphe et sa difficulté. J'ai personnellement testé plusieurs graphes à une 30ènes d'îles et mon programme a su tous les résoudre.

V/ Conclusion

Finalement, ce projet m'a réellement poussé à me surpasser. J'ai appris de chacun dès problèmes que j'ai rencontrés et suis fier du résultat final.