Projet Prep'ISIMA 2 : Exploration dans un réseau

Théo Peuchlestrade
Loris Van Katwijk

2020-2021

Intitulé du projet : Algorithme d'exploration en profondeur d'un réseau en présence de conflits entre plusieurs arêtes.

Table des matières

1 Introduction **2**

Courte mise en contexte **2**

Explications des algorithmes réalisés **2**

Résultats et explication de notre interface **2**

Conclusion sur notre projet **2**

2 Notre programme **5**

2.1 La fonction *dfs* 6
2.2 La fonction *randomGraph* 7
2.3 La fonction *randomInterdits* 8
2.4 La procédure *afficheGraph* 8
2.5 La procédure *afficheDfs* 10
2.6 La procédure *graphFromJson* 12
2.7 La procédure *loadRandomFunctions* 13
2.8 La procédure *fileOpenerInterface* 14
2.9 La procédure *afficheInterditsInterface* 15
2.10 La procédure *info* 15

3 Le résultat **16**

3.1 La zone de saisie 18
3.2 Le bouton "Générer aléatoirement un graphe" 18
3.3 Le bouton "Utiliser un graphe existant" 19
3.4 Le bouton "Afficher le graphe" 20
3.5 Le bouton "Afficher les interdits" 21
3.6 Le bouton "Afficher le DFS" 22
3.7 Le bouton "I" 23

4 Conclusion **23**

1 Introduction

Les graphes sont un sujet qui passionne bon nombre de chercheurs dans le domaine des mathématiques et de l'informatique. Un graphe est une structure composée d'objets dans laquelle certaines paires d'objets sont en relation. Les objets correspondent à des abstractions mathématiques et sont appelés noeuds, et les relations entre noeuds sont des arêtes. [1] C'est un sujet qui est étudié depuis le 18ème siècle et dont beaucoup de questions restent toujours en suspens. Les domaînes de la résolution de graphes ou de leur visualisation sont toujours en plein essort. Nous avons choisi de travailler sur ce sujet passionnant, et d'implémenter une méthode de résolution et un moyen de visualiser les graphes.

Le plan de notre rapport débutera tout d'abord par une courte mise en contexte de notre projet en partie 1. Ensuite, nous détaillerons nos choix de développement dans la partie 2. Puis, dans la partie 3 nous montrerons les différentes fonctionnalités de notre interface. Enfin, dans la partie 4 nous conclurons notre projet.

Voici le plan de notre rapport point par point :
- Courte mise en contexte
- Explications des algorithmes réalisés
- Résultats et explication de notre interface
- Conclusion sur notre projet

Notre projet est basé sur de la théorie que nous mettons en pratique à travers différents algorithmes.

Pour la partie théorie, nous travaillons sur des réseaux qui sont des ensembles interconnectés, faits de composants et de leurs inter-relations, autorisant la circulation en mode continu ou discontinu de flux. [2] Dans notre cas, un réseau est un ensemble de noeuds (aussi appelés sommets ou points) inter-connectés par des arêtes (aussi appelés liens ou lignes) les reliants entre eux. Certaines arêtes peuvent être définies comme incompatibles entre elles. C'est à dire que lorsque l'on utilise l'une on ne peut pas utiliser l'autre. Deux arêtes incompatibles peuvent être comparées dans la vie courante à un ensemble de routes reliant un point A à un point B. Il est possible que certaines

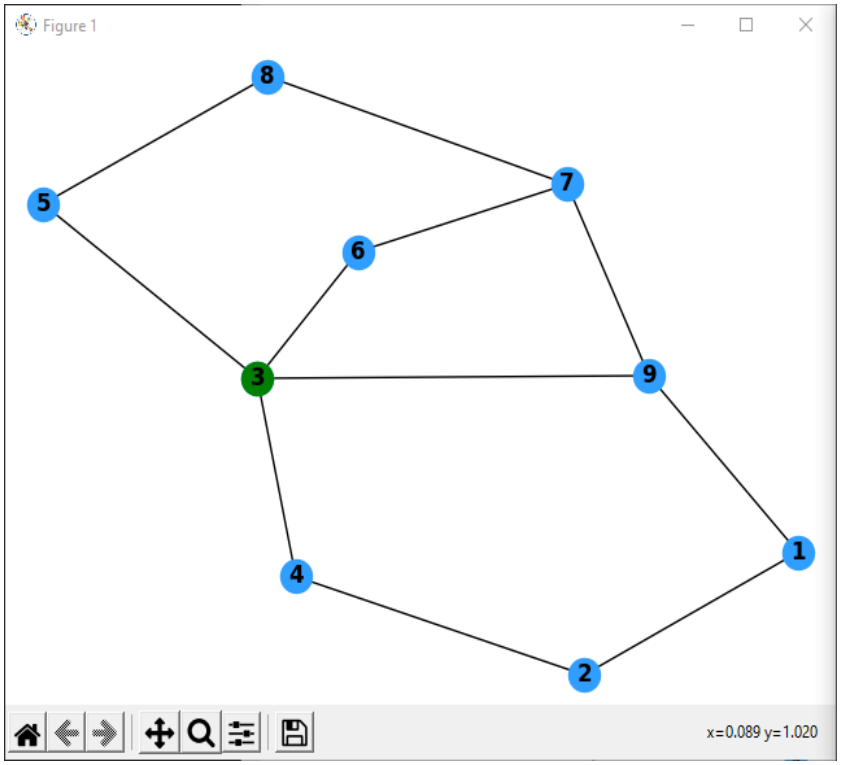
d'entre elles soient à sens unique et donc qu'on ne puisse pas les prendre en sens inverse. Il ne faut pas oublier de compter ces routes et dire qu'elles sont indisponibles à l'utilisation. Chaque noeud possède une liste de noeuds dans son entourage direct, les noeuds liés directement à lui par des arêtes. On appelle cela le voisinage du noeud, les noeuds autour étant des voisins. [3]

Pour la partie technique, nous basons notre résolution de graphe sur un algorithme de *Parcours en profondeur* de graphes, ou *Depth First Search* (DFS) en anglais. Son application la plus simple consiste à déterminer s'il existe un chemin d'un noeud à un autre. [4] Cet algorithme n'est pas à confondre avec un algorithme de *Parcours en largeur*, ou *Breadth First Search* (BFS) en anglais, permettant de calculer les distances de tous les noeuds depuis un noeud source. [5] Son mode de fonctionnement est le suivant : on commence par explorer un nœud source, puis ses successeurs, puis les successeurs non explorés des successeurs, etc.

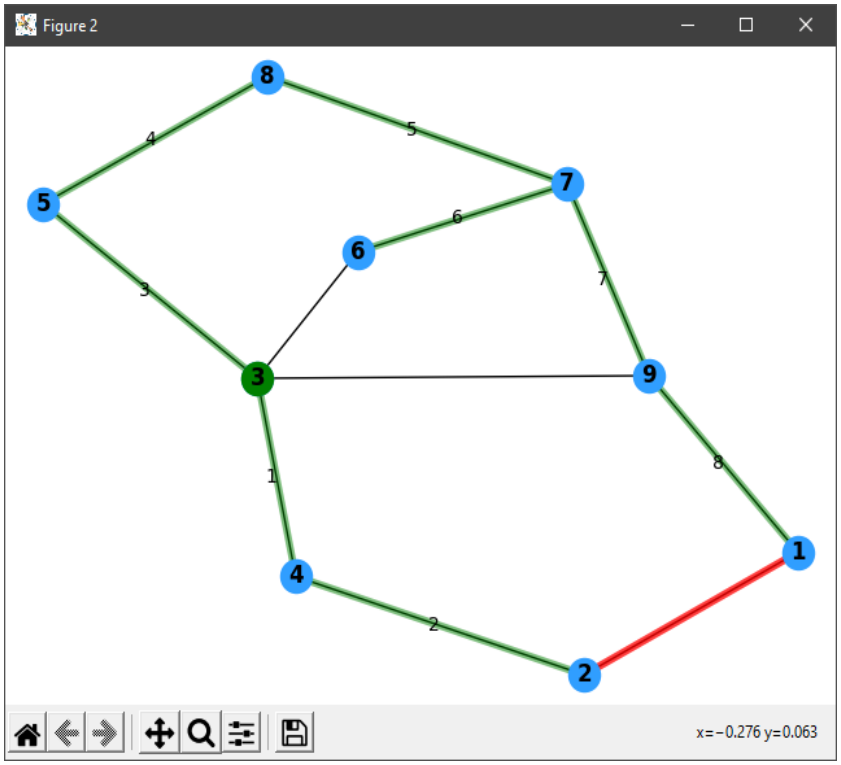
Ensuite, nous notons toutes les informations relatives à un graphe dans un fichier JSON. Le JSON ou "JavaScript Object Notation" est un format de données textuelles dérivé de la notation des objets du langage JavaScript. Il permet de représenter de l'information structurée comme le permet XML par exemple. [6] Toutes les variables que nous appelons *graph* dans nos algorithmes sont en fait des fichiers en JSON (ou leur contenu) dans lesquels nous avons spécifié les caractéristiques de notre graphe. Dans ce fichier JSON, nous définissons tout d'abord le voisinage de chaque noeud dans un dictionnaire portant le nom "graph". Ensuite, nous écrivons les incompatibiltés entre deux arêtes, au sein de la liste "interdits". Ces incompatibilités sont en fait le résultat de deux arêtes qui, lorsque l'on passe sur une, rend le passage sur l'autre interdit. Enfin, nous pouvons définir un point de départ pour la résolution du graphe par notre algorithme DFS, en spécifiant un entier à "depart".

Exemple de l'écriture d'un graphe dans un fichier JSON possédant neuf noeuds, deux incompatibilités et un point de départ, et sa visualisation dans notre programme :


```
{"graph": {"1": ["2", "9"], "2": ["1", "4"], "3": ["4", "5", "6", "9"], "4": ["2", "3"], "5": ["3", "8"], "6": ["3", "7"], "7": ["6", "8", "9"], "8": ["5", "7"], "9": ["1", "3", "7"]},
"interdits": [[["1", "2"], ["2", "4"]], [["1", "2"], ["3", "4"]]],
"depart": "3"}
```



Voici ci-dessous un schema explicitant le parcours du DFS sur l'exemple de graphe en JSON présenté précédemment. Un graphe suit un principe d'économie. C'est à dire que lorsqu'un graphe possède des incompatibilités entre arêtes, on peut les enlever (si on le souhaite) de notre visualisation et notre graphe devient un arbre (qui est en fait un graphe acyclique et connexe). Les arêtes vertes et les chiffres associés dans l'ordre croissant explicitent l'ordre de passage du DFS sur les différentes arêtes du graphe. Les arêtes rouges sont celles qui sont interdites. Le point de départ est le noeud représenté par la couleur verte. Les autres points sont de couleur bleue.



2 Notre programme

Dès le début de la gestation du projet, nous avions pour idée de réaliser une interface graphique. En effet, nous avions déjà pour but de créer un programme interactif et ergonomique. Nous voulions que notre outil puisse être utilisé par n'importe quelle personne intéressée dans le cadre de son loisir ou de son activité professionnelle. Nous avons donc réparti nos algorithmes en fonction des différents boutons fonctionnels que nous voulions mettre en avant sur notre interface.

Nous allons donc vous présenter nos différents algorithmes permettant le bon fonctionnement de notre programme, dans les différentes sous-parties ci-après. Les seuls "packages", bibliothèques python nécessaires au fonctionnement de notre programme sont les suivantes : matplotlib, networkx et tkinter. Nous avons choisi de restreindre le nombre de packages utilisés afin que l'utilisateur n'ait à installer que le minimum possible de bibliothèques externes

et ainsi rendre l'utilisation de notre programme plus fluide et instinctive.

Nous avons également créé un site web, disponible à l'adresse suivante projetisima.000webhostapp.com, regroupant les différentes étapes de l'avancement de notre projet.

2.1 La fonction *dfs*

Nous avons choisi un DFS de type récursif car il se décrit naturellement de manière récursive et la récursivité est ce que nous préférons employer pour nous approprier ce genre d'algorithme (nous préférons la récursivité à l'itérativité). Dans l'approche itérative, on insère d'abord tous les éléments dans une pile, la tête de pile étant le dernier noeud inséré et le premier noeud que l'on manipule étant le dernier enfant. Dans l'approche récursive on manipule chaque noeud quand on le voit, le premier noeud que l'on manipule est le premier enfant. Nous avons donc créé un algorithme de DFS récursif, efficace et explicite, nommé "dfs" et basé sur l'algorithme disponible sur le site askpython.com, que nous avons modifié, sans en perdre son essence même. Notre DFS retourne la liste des arêtes visitées et la liste des arêtes fermées incompatibles.

```
# Algorithme "Deep First Search"
#-------------- Code Récupéré, ou légerement modifié --------------
def dfs(graph, interdit, node, visited=[], closed=[], visited_edges=[]):
    visited.append(node) # On commence par rajouter le point actuel à la liste des points visités
    for k in sorted(graph[node]): # Pour chaque voisin du point

        #-------------- Code Personnel --------------
        if [node,k] not in closed and [k,node] not in closed: # si le lien avec ce voisin n'est pas fermé
            if k not in visited:            # Si le voisin n'a pas deja été visité
                for (lien1,lien2) in interdit:
                    if node in lien1 and k in lien1:# Si le lien à une règle d'interdiction
                        closed.append(lien2)            # On ferme les liens qui deviennent incompatibles
                    elif node in lien2 and k in lien2:# Même chose pour que les interdictions soient symétriques
                        closed.append(lien1)
                visited_edges.append((node,k)) # On ajoute le lien aux liens visités

                #-------------- Code Récupéré, ou légerement modifié --------------
                dfs(graph, interdit, k, visited, closed, visited_edges) # On effectue le même algorithme pour le voisin.
    return visited_edges, closed
```

Ce que retourne l'algorithme :

{} ex1.json > ...

```
1  {"graph": {"1": ["2", "9"], "2": ["1", "4"], "3": ["4", "5", "6", "9"], "4": ["2", "3"], "5": ["3", "8"], "6": ["3", "7"], "7": ["6", "8", "9"], "8": ["5", "7"], "9": ["1", "3", "7"]},
2   "interdits": [[["1", "2"], ["3", "9"]], [["1", "2"], ["3", "6"]]]}
```

{} result.json X

{} result.json > ...

```
1  {"visited_edges": [["1", "2"], ["2", "4"], ["4", "3"], ["3", "5"], ["5", "8"], ["8", "7"], ["7", "6"], ["7", "9"]],
2   "closed": [["3", "9"], ["3", "6"]]}
```

2.2 La fonction *randomGraph*

Cet algorithme permet de générer des graphes de manière aléatoire en prenant en paramètre le nombre de noeuds choisis, c'est à dire un entier, pour le graphe à générer. L'algorithme retourne un dictionnaire contenant les voisins attribués à chacun des noeuds, afin de pouvoir constituer un graphe. Ce dictionnaire est utilisé au sein de *loadRandomFunctions*.

```python
# Algorithme de génération aléatoire de graphes :
def randomGraph(nbNodes = 9):
    graph = {}
    for i in range(1, nbNodes+1):   # On initialise le graphe
        graph[str(i)] = []
    for i in range(1, nbNodes+1):   # On génère alétoirement le voisinage de chaque noeud
        rge = list(range(1, nbNodes+1))
        rge.remove(i)
        r = random.sample(rge, random.randint(1, nbNodes//2))
        for j in r:
            if (str(j) not in graph[str(i)]):
                graph[str(i)].append(str(j))
        for j in r:
            if (str(i) not in graph[str(j)]):
                graph[str(j)].append(str(i))
    for i in range(1, nbNodes+1):   # On trie les voisins
        graph[str(i)].sort()
    return graph
```

Ce que retourne l'algorithme :

{"graph": {"1": ["2", "9"], "2": ["1", "4"], "3": ["4", "5", "6", "9"], "4": ["2", "3"], "5": ["3", "8"], "6": ["3", "7"], "7": ["6", "8", "9"], "8": ["5", "7"], "9": ["1", "3", "7"]},

2.3 La fonction *randomInterdits*

Cet algorithme a pour but de générer une liste aléatoire d'incompatibilités entre arêtes. Il est utilisé au sein de *loadRandomFunctions*. Il prend en paramètre un graphe, c'est à dire un dictionnaire contenant les spécificités du graphe, c'est à dire les voisins de chaque noeud. Il retourne la liste des incompatibilités entre arêtes associées à certains noeuds.

Ce que prend en entrée l'algorithme :


```
{"graph":{"1":["2","9"],"2":["1","4"],"3":["4","5","6","9"],"4":["2","3"],"5":["3","8"],"6":["3","7"],"7":["6","8","9"],"8":["5","7"],"9":["1","3","7"]},
```


```python
# Algorithme de génération aléatoire d'interdictions entre les noeuds :
def randomInterdits(graph, interditMin=1, interditMax=4):
    interdit = []
    for i in range(random.randint(interditMin,interditMax)):    # On génére une liste d'interdits de manière aléatoire
        r=str(random.randint(1,len(graph)))
        (a,b) = (r,graph[r][random.randint(0,len(graph[r])-1)])
        (c,d) = (a,b)
        while (c,d)==(a,b) or (c,d)==(b,a):
            r=str(random.randint(1,len(graph)))
            (c,d) = (r,graph[r][random.randint(0,len(graph[r])-1)])
        interdit.append( (sorted((str(a),str(b))), sorted((str(c),str(d))) ))
    return interdit
```

Ce que retourne l'algorithme :

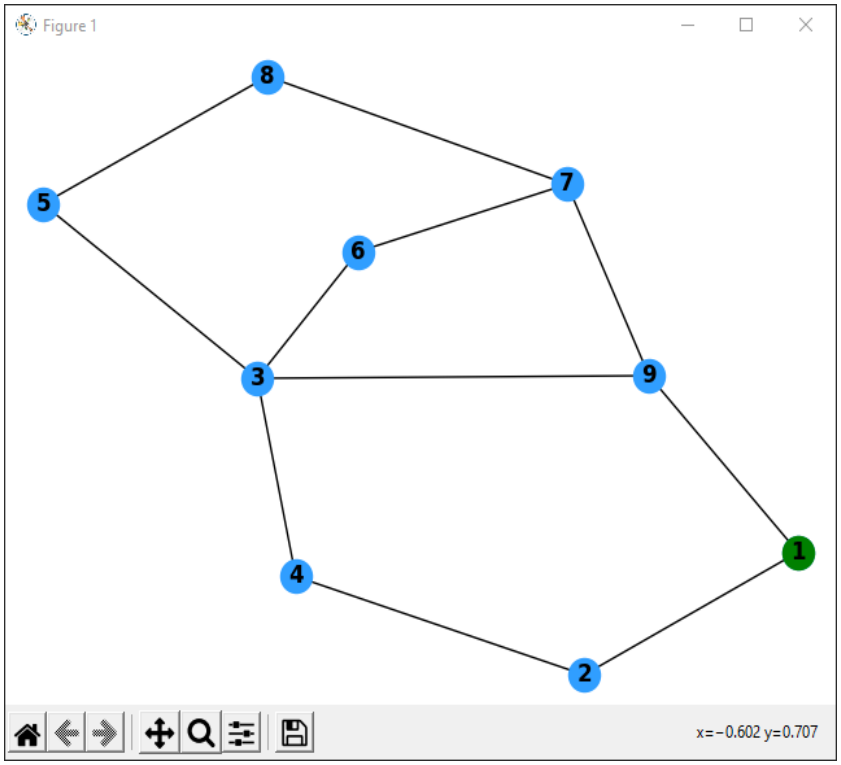

```
"interdits": [[["1", "2"], ["3", "9"]], [["1", "2"], ["3", "6"]]]}
```


2.4 La procédure *afficheGraph*

Cet algorithme prend en paramètre un dictionnaire et son point de départ (noeud). Ensuite, il utilise le module **NetworkX** afin d'afficher le graphe dans une fenêtre d'affichage à part. Les propriétés de coloration des différents noeuds et du noeud de départ sont définies à l'intérieur.

```python
# Algorithme affichant un graphe pourvu de ses spécificités :
def afficheGraph(graph, debut=1):
    plt.figure()

    G = nx.Graph()  # Intégration du graphe dans NetworkX
    for i in graph:
        G.add_node(i)
    for i in graph:
        for j in graph[i]:
            G.add_edge(i,j)

    pos = nx.spring_layout(G, seed=6)

    nx.draw(G, with_labels=True, pos=pos, font_weight='bold', node_color='#309eff') # Affiche le graphe
    nx.draw_networkx_nodes( # Affiche le noeud de debut en vert
        G,
        pos,
        nodelist=[debut],
        node_color='g'
    )
    plt.show()
```

Ce qu'affiche l'algorithme :

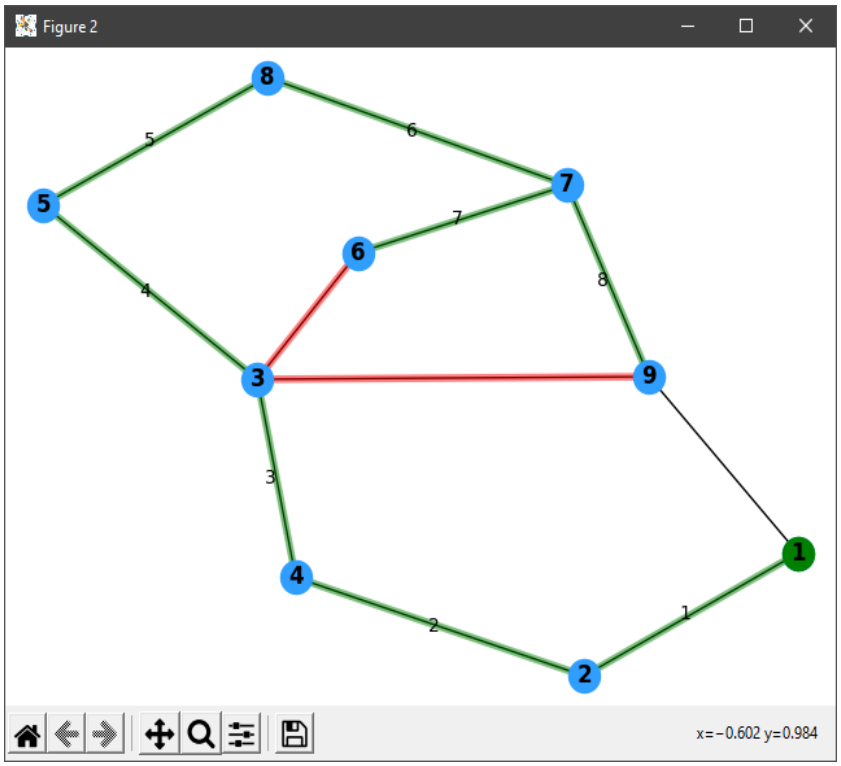


2.5 La procédure *afficheDfs*

Cet algorithme prend en paramètre un dictionnaire contenant les voisins de chacun des noeuds de notre graphe, une liste contenant les incompatibilités entre arêtes associées à certains noeuds, un entier représentant le noeud de départ et ce que retourne la fonction *dfs*. Ensuite, il utilise le module **NetworkX** afin d'afficher le graphe résolu dans une fenêtre d'affichage à part. Les propriétés de coloration des différents noeuds, du noeud de départ, des arêtes incompatibles et de numérotation du déroulement du DFS sont définies à l'intérieur.

```python
# Affiche le graph avec le chemin emprunté par l'algorithme DFS
def afficheDfs(graph, interdictions, debut, dfs):
    #plt.text(-1,1.2, "Incompatibilités : "+str(interdictions))  # On affiche les incompatibilités
    (visited_edges, closed) = dfs
    plt.figure()

    G = nx.Graph()  # Intégration du graphe dans NetworkX
    for i in graph:
        G.add_node(i)
    for i in graph:
        for j in graph[i]:
            G.add_edge(i,j)

    pos = nx.spring_layout(G, seed=6)

    nx.draw(G, with_labels=True, pos=pos, font_weight='bold', node_color='#309eff') # Affiche le graphe

    nx.draw_networkx_edges( # Affiche les arrêtes interdites en rouge
        G,
        pos,
        edgelist=closed,
        width=4,
        alpha=0.5,
        edge_color="r",
    )

    nx.draw_networkx_edges( # Affiche les arrêtes visitées en vert
        G,
        pos,
        edgelist=visited_edges,
        width=4,
        alpha=0.5,
        edge_color="g",
    )

    nx.draw_networkx_nodes( # Affiche les noeuds non visités en rouge
        G,
        pos,
        nodelist=[x for x in graph if (not x in [y for (a,y) in visited_edges] and x != debut)],
        node_color='r'
    )

    nx.draw_networkx_nodes( # Affiche le noeud de debut en vert
        G,
        pos,
        nodelist=[debut],
        node_color='g'
    )

    nx.draw_networkx_edge_labels( # Affiche les numero des liens
        G,
        pos,
        edge_labels={visited_edges[i]:i+1 for i in range(len(visited_edges))},
        rotate = False,
        horizontalalignment='right',
        bbox = {"alpha":0}
    )

    with open("result.json", 'w') as f:
        f.write(json.dumps({"visited_edges":visited_edges,"closed":closed}))
        f.close()
    plt.show()
```

Ce qu'affiche l'algorithme :



2.6 La procédure *graphFromJson*

Cet algorithme récupère les données d'un graphe écrites dans un fichier JSON. Les différentes données du graphe sont : un dictionnaire "graph" regroupant les noeuds avec leur voisinage, une liste "interdits" regroupant les incompatibilités entre arêtes et l'entier "depart" représentant le noeud de départ. Cet algorithme va charger ces différentes informations dans les paramètres / variables des algorithmes les utilisant.

```python
# Algorithme récupérant les différentes informations d'un graphe écrites dans un fichier .json :
def graphFromJson(path):
    global graphSelect
    global interditSelect
    global departSelect
    with open(path) as f:
        x = json.loads(f.read().replace('\n', ''))
    if ("depart" in x):
        (graphSelect,interditSelect, departSelect) = (x["graph"], x["interdits"], x["depart"]) #
    else:
        (graphSelect,interditSelect, departSelect) = (x["graph"], x["interdits"], '1')
```

2.7 La procédure *loadRandomFunctions*

Cet algorithme utilise les fonctions *randomGraph* et *randomInterdits* afin de récupérer les différentes informations d'un graphe et les écrire dans un fichier JSON.

```python
# Algorithme utilisant un graphe généré alétoirement en convertissant ses spécificités en texte inscrit dans un fichier .json et le charge :
def loadRandomFunctions(nbNoeuds=9, interditMin=1, interditMax=4):
    with open("random.json", 'w') as f:
        rgraph = randomGraph(nbNoeuds)
        f.write(json.dumps({"graph":rgraph,"interdits":randomInterdits(rgraph,interditMin,interditMax),"depart":str(random.randint(1,nbNoeuds))}))
        f.close()
    graphFromJson("random.json")
    buttonInterdits["state"] = "normal"
    buttonDfs["state"] = "normal"
    buttonGraph["state"] = "normal"
    plt.show()
```

Format de ce qu'écrit l'algorithme dans le fichier *random.json* (les spécificités du graphe généré aléatoirement) :

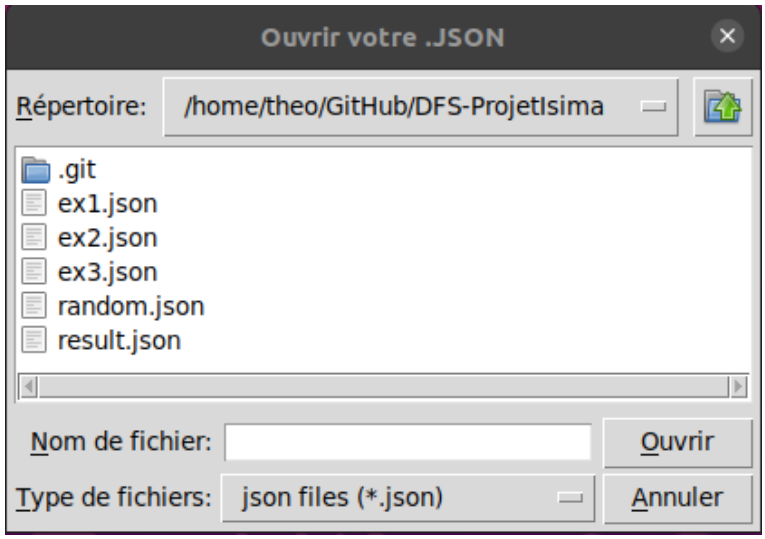

```
{"graph": {"1": ["2", "3", "4", "5", "7"], "2": ["1", "7"], "3": ["1", "5", "6", "7"], "4": ["1", "5"], "5": ["1", "3", "4", "6", "7"], "6": ["3", "5"], "7": ["1", "2", "3", "5"]},
"interdits": [[["1", "3"], ["4", "5"]], [["5", "6"], ["4", "5"]]],
"depart": "1"}
```

2.8 La procédure *fileOpenerInterface*

Cet algorithme permet d'ouvrir le gestionnaire de fichier afin que l'utilisateur puisse sélectionner le fichier JSON contenant les informations de son graphe. Ce dernier sera utilisé en tant que paramètre / variable dans les algorithmes ayant besoin des informations contenues dans ce fichier.

```
# Algorithme permettant l'ouvrture de fichier afin de les utiliser dans les différents algorithmes précédents
def fileOpenerInterface():
    filename = askopenfilename(title="Ouvrir votre .JSON", filetypes=[('json files','.json'),('all files','.*')])
    graphFromJson(filename)
    buttonInterdits["state"] = "normal"
    buttonDfs["state"] = "normal"
    buttonGraph["state"] = "normal"

    plt.show()
```

Ce qu'affiche l'algorithme (par exemple lorsque l'on clique sur le bouton qui active la fonction dans l'interface, détaillée dans la prochaine partie) :

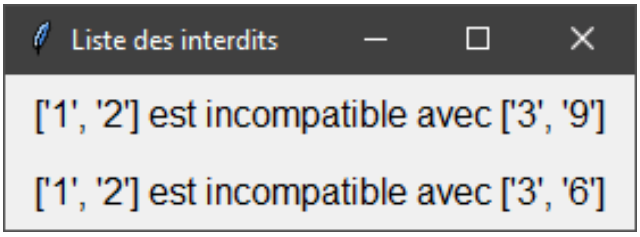


2.9 La procédure *afficheInterditsInterface*

Cet algorithme utilise les procédures *loadRandomFunctions* ou *graphFromJson* afin d'afficher la liste "interdits" dans une nouvelle fenêtre.

```
# Ouvre une fenetre sur laquel on donne la liste des incompatibilités
def afficheInterditsInterface(interdit):
    fenetreInterdit = Tk()
    fenetreInterdit.title("Liste des interdits")


    for (a,b) in interdit:
        Label(fenetreInterdit, text=f"{a} est incompatible avec {b}", font="Arial 12").pack(padx=10,pady=5)

    fenetreInterdit.mainloop()
```

Ce qu'affiche l'algorithme :

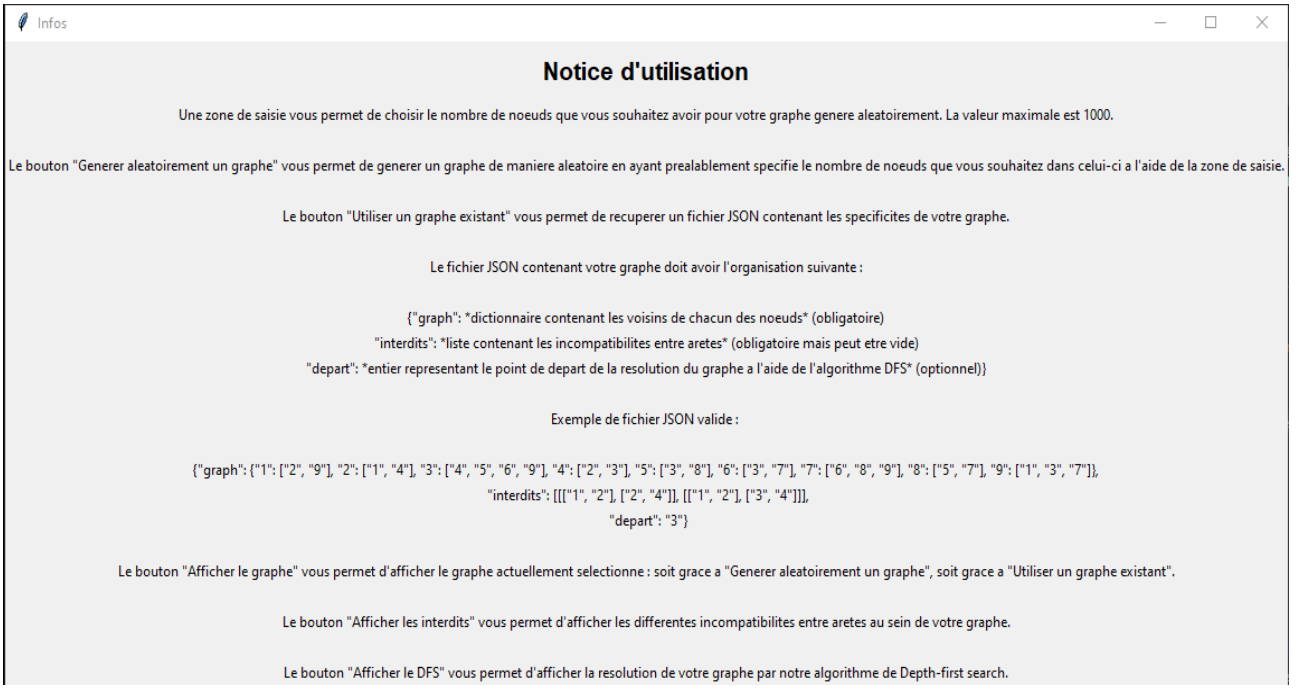


2.10 La procédure *info*

Cette procédure permet d'ouvrir un fichier de notice permettant à l'utilisateur de mieux comprendre le fonctionnement de l'interface graphique.

```
# Notice
def info():
    fenetreInfo = Tk()
    fenetreInfo.title("Infos")
    Label(fenetreInfo, text="Notice d'utilisation", font="Arial 16 bold").pack(padx=20, pady=10)
    with open("README.txt","r") as f:
        lines = str(f.read()).split("\n")
    for l in lines:
        Label(fenetreInfo, text=l).pack()
```

Ce qu'affiche l'algorithme :



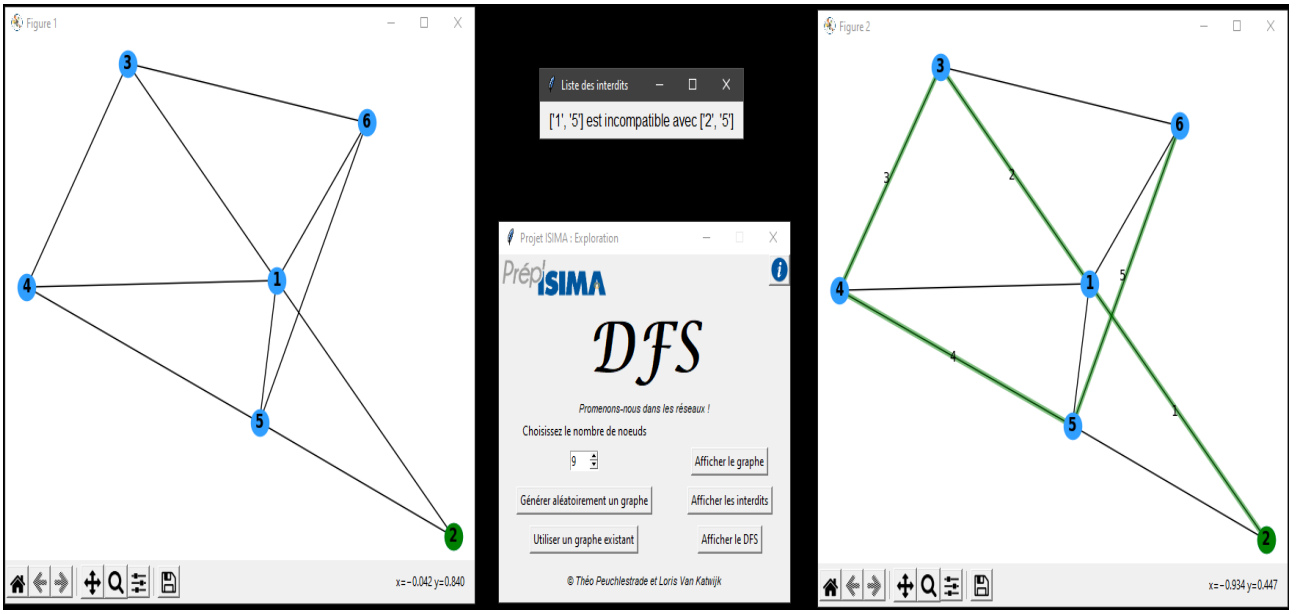
3 Le résultat

Nous avons donc créé une interface simple d'utilisation, fonctionnelle et efficace, afin que nos utilisateurs puissent naviguer avec aisance à travers les différentes fonctionnalités. Notre interface graphique se décompose en deux parties : la partie supérieure et la partie inférieure de l'interface.

Dans la partie supérieure gauche est disposé le logo de notre classe préparatoire intégrée au sein de l'ISIMA, dans le cadre de laquelle nous réalisons ce projet. Ensuite, dans la partie supérieure centrale, nous pouvons distinguer le titre de notre programme ainsi qu'un sous-titre accrocheur. Enfin, dans la partie supérieure droite, nous avons implémenté un bouton "I" permettant d'ouvrir une notice d'utilisation de notre interface afin d'aider l'utilisateur à appréhender notre programme.

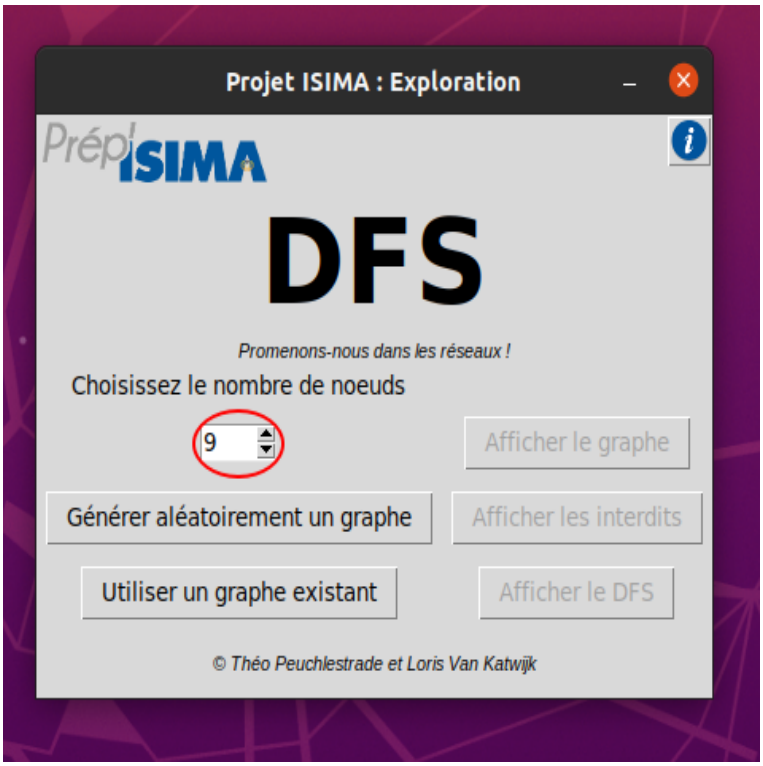
Dans la partie inférieure gauche, nous avons disposé les différents boutons "d'entrée", permettant d'amener du contenu dans notre programme (générer un graphe ou l'importer). Ensuite, dans la partie inférieure droite, nous avons disposé les différents boutons "de sortie", permettant le traitement des graphes et leur affichage. Enfin, dans le bas de page, nous avons mis nos noms et prénoms afin de déposer notre marque sur le programme. Maintenant, nous allons vous présenter les différents boutons/fonctionnalités disponibles sur notre interface graphique et leur utilité.

Vous avez la possibilité sous *Windows* d'appuyer sur plusieurs boutons de visionnage simultanément et voir apparaître les différentes fenêtres qui s'affichent les unes à côté des autres. Cependant, sur certaines versions de **NetworkX** ou **Matplotlib** sous *Linux*, lorsque l'on appuie sur le bouton *Afficher le graphe* ou le bouton *Afficher le DFS* il est impossible d'appuyer sur d'autres boutons de visionnages tant que les fenêtres de *Afficher le graphe* ou *Afficher le DFS* sont ouvertes. Ceci est un problème indépendant de notre volonté, nous ne pouvons pas le résoudre malheureusement. Si cela vous arrive, essayez d'installer une autre version de **NetworkX** ou **Matplotlib**, ou essayez de lancer notre programme sous un autre système d'exploitation.



3.1 La zone de saisie

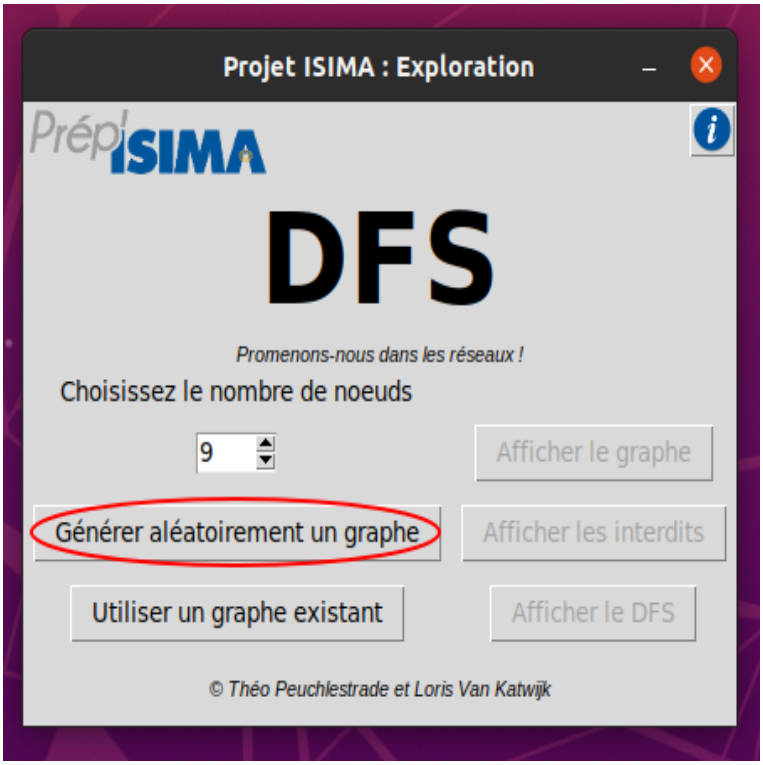
La zone de saisie permet de choisir le nombre de noeuds que l'on souhaite avoir pour notre graphe généré aléatoirement. A partir d'environ 20 noeuds, la visualisation sous **NetworkX** commence à être assez confuse. Malgré tout nous avons choisi de limiter la valeur maximale à 1000 noeuds pour que l'utilisateur puisse essayer un grand nombre de possibilités. Cependant, cette limitation est totalement liée à l'interface graphique, nos algorithmes eux n'ont pas de limitations, vous pouvez y insérer la valeur que vous souhaitez.



3.2 Le bouton "Générer aléatoirement un graphe"

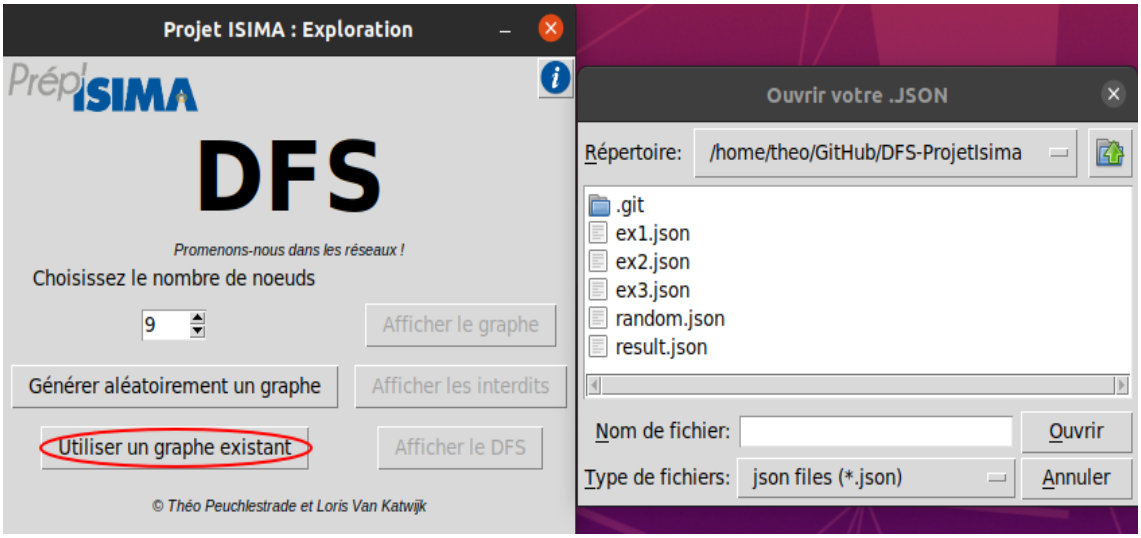
Ce bouton permet d'activer la procédure *LoadRandomFunctions* afin de générer un graphe aléatoirement. Le nombre de noeuds est défini par l'utilisateur dans la zone de saisie. Il y a un appel à la fonction *randomGraph*, afin d'insérer dans un JSON ("random.json") un dictionnaire contenant un graphe aléatoire. Il y a également un appel à la fonction *randomInterdits*, afin d'insérer dans le fichier "random.json" la liste des incompatibilités entre

arêtes. Un entier généré aléatoirement (par un randint) est utilisé pour définir le point de départ de la résolution du graphe.



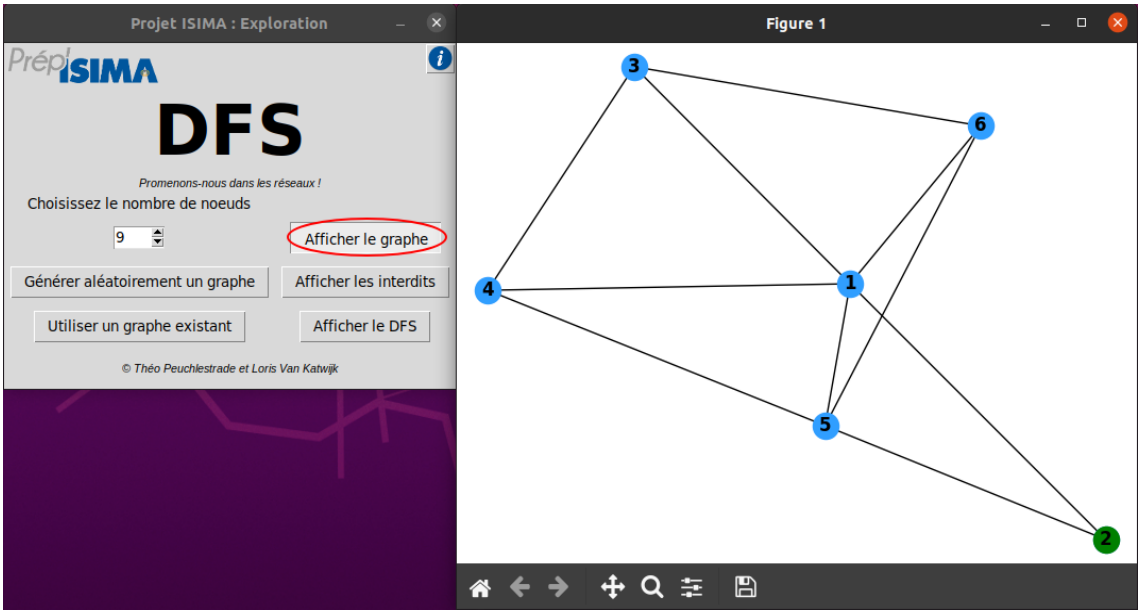
3.3 Le bouton "Utiliser un graphe existant"

Ce bouton permet d'activer la procédure *fileOpenerInterface*, qui permet à l'utilisateur de choisir un graphe qu'il a écrit dans un fichier JSON.



3.4 Le bouton "Afficher le graphe"

Ce bouton permet d'activer la procédure *afficheGraph*. Elle n'est disponible que lorsque l'utilisateur à déjà préalablement cliqué sur "Générer un graphe aléatoirement", ou bien s'il a chargé son propre graphe en JSON en ayant cliqué sur "Charger un graphe". A partir de ce moment, le graphe courant (soit aléatoire, soit chargé en fonction du dernier choix de l'utilisateur) est affiché.



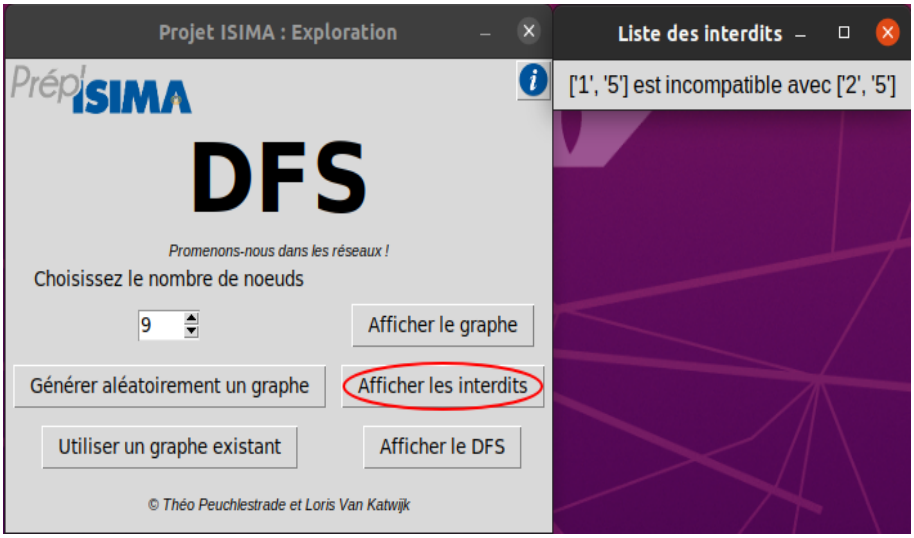
3.5 Le bouton "Afficher les interdits"

Ce bouton permet d'activer la procédure *afficheInterditsInterface* qui affiche :

- soit les interdits du graphe aléatoire, si l'utilisateur a cliqué sur le bouton "Générer aléatoirement un graphe",

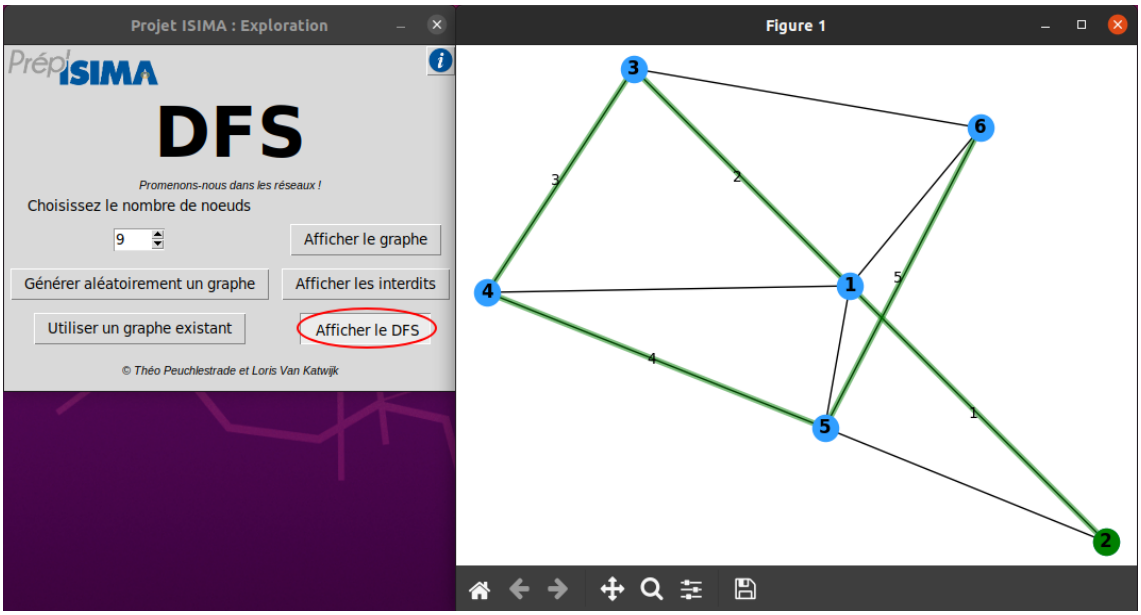
- soit les interdits du graphe que l'utilisateur a chargé, s'il a appuyé sur le bouton "Utiliser un graphe existant".

Ces interdits sont affichés dans une nouvelle fenêtre.



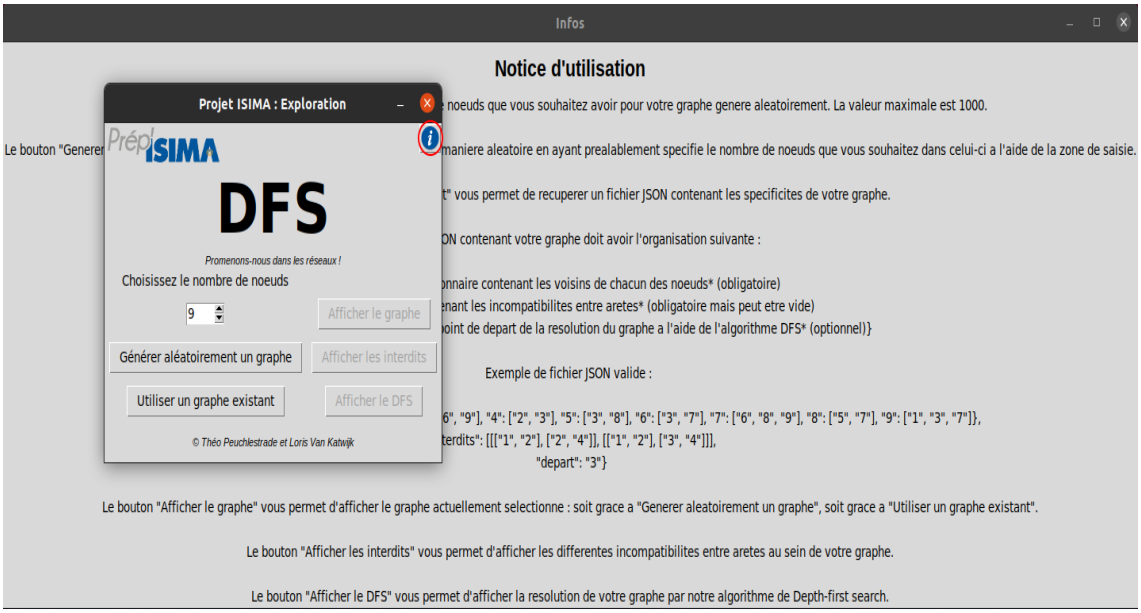
3.6 Le bouton "Afficher le DFS"

Ce bouton permet d'activer la fonction *dfs* ainsi que la procédure *afficheDfs*. Il y a affichage dans une nouvelle fenêtre le graphe résolu ainsi qu'écriture du résultat de la fonction *dfs* dans un fichier JSON s'intitulant "result.json".



3.7 Le bouton "I"

Ce bouton permet d'activer la procédure *info* et affiche la notice.



4 Conclusion

Ce projet nous a permis de développer une interface graphique afin que tout utilisateur, avec ou sans grande connaissance en informatique, puisse générer des graphes (de façon aléatoire ou manuelle dans un fichier JSON). L'utilisateur peut appliquer, sur ces graphes, des fonctionnalités de visionnage, parcours de type DFS et mise en valeur des incompatibilités.

Notre but était surtout de créer une solution alliant ergonomie, praticité (en utilisant le moins de bibliothèques python externes possibles) et efficacité des différents algorithmes. Nous voulions vraiment que l'utilisateur soit guidé pas à pas, nous avons donc écrit une notice courte et complète sur son utilisation. Nous avons mis tout en oeuvre pour proposer un programme complet et utilisable par tous.

Que ce soit pour comprendre les graphes, les parcourir ou les visualiser, ce programme permet à tout type de personnes de s'épanouir dans ce domaine. Bien sûr, un programme n'est jamais parfait et il y a toujours de la place pour l'optimisation, l'amélioration du design et des fonctionnalités.

Références

[1] Wikipedia, "Graphe (mathématiques discrètes)." https://fr.wikipedia.org/wiki/Graphe_(math%C3%A9matiques_discr%C3%A8tes).

[2] Wikipedia, "Théorie des réseaux." https://fr.wikipedia.org/wiki/Th%C3%A9orie_des_r%C3%A9seaux.

[3] T. Cormen, "Introduction à l'algorithmique." https://www.amazon.fr/Introduction-%C3%A0-lalgorithmique-Thomas-Cormen/dp/2100031287.

[4] Wikipedia, "Algorithme de parcours en profondeur." https://fr.wikipedia.org/wiki/Algorithme_de_parcours_en_profondeur.

[5] Wikipedia, "Algorithme de parcours en largeur." https://fr.wikipedia.org/wiki/Algorithme_de_parcours_en_largeur.

[6] Wikipedia, "Javascript object notation." https://fr.wikipedia.org/wiki/JavaScript_Object_Notation.