MIKOU Abla
NDIAYE Maïrame

Rapport Projet Intelligence Artificielle

Lancement du projet

Afin de pouvoir utiliser notre code, il suffit d'ouvrir un terminal et de mettre la commande suivante : "python3 MIKOU_NDIAYE.py". Le programme vous demandera de renseigner différents paramètres nécessaires à son exécution. Premièrement, il faut renseigner le nombre de nœuds dans le graphe initial, par la suite inscrire la probabilité de présence d'un arc entre deux nœuds. Finalement, le programme demande la taille de la population initiale pour l'algorithme génétique.

I – Démarches pour la réalisation du projet

A - Prise en main du projet

Avant de commencer à implémenter les algorithmes, nous sommes passées par différentes étapes. Premièrement, nous devions comprendre le sujet et savoir ce qu'était exactement le problème de coupe-cycle minimum. Pour réaliser cela, nous nous sommes aidés de la définition fournie dans le sujet et également d'internet qui nous a permis de voir des exemples de ce que c'est. Nous nous sommes également remis à niveau sur les algorithmes qu'il fallait implémenter. Pour cela nous avons utilisé des slides du cours d'Intelligence Artificielle et des TPs de cours.

B - Formalisation du problème

Après cette phase de recherche et de documentation, nous nous sommes focalisés sur la formalisation du problème, étape très cruciale. Nous avons ainsi défini de manière formelle chaque notion importante du projet que nous allons présenter dans cette section. Les notions sont définies comme suit :

La problème : Soit G = (V,A) un graphe orienté, nous cherchons le plus petit ensemble de sommets $S \subseteq V$ tel que G' = (V\S, A) soit acyclique. Donc S est un ensemble de nœuds ayant une intersection non vide avec chacun des cycles du graphe initial.

La fonction de coût : Comme proposé dans le sujet, nous définissons le score d'un graphe G comme étant le nombre de sommets dans G qui sont inclus dans au moins un cycle de G. Soit $C_1,...,C_p$ les ensembles de sommets de chaque cycle de G, alors

$$f(G) = |\bigcup_i Ci\ |$$

Ainsi, cette fonction est à minimiser, une solution est valide lorsque f(G) = 0.

La notion de voisinage : Soit G = (V,A) un graphe orienté, l'ensemble des voisins de G correspond à tous les graphes qui sont identiques à G à l'exception d'un seul nœuds (et de ses arêtes). Soit s un sommet et X l'ensemble des arêtes pour lesquels s est une extrémité alors

$$Neighbors(G) = \{ G'(V',A') \text{ tq } G = G'(V'\backslash\{s\} ,A'\backslash X) \text{ ou } G= G'(V' \cup \{s\} ,A' \cup X)\}.$$

Formalisation pour l'algorithme génétique :

Pour cet algorithme, nous devions définir les notions suivantes :

- ***La fonction fitness :*** Nous avons défini la fitness comme étant le complémentaire de la fonction de coût. Soit G(V,A) un graphe orienté, la fitness de G correspond au nombre de sommets qui ne sont inclus dans aucun des cycles de G.

  Soit $C_1,...,C_p$ les ensembles de sommets de chaque cycle de G, alors

  $$fitness(G) = |V| - f(G)$$

  avec f la fonction de coût. Ainsi, plus la valeur de cette fonction est élevée pour un individu de la population, plus l'individu aura de chance d'être choisie lors de la sélection et ainsi de se reproduire. Lorsque la fitness d'un graphe est égale à son nombre de nœuds, alors nous considérons que le graphe est une solution valide.
  Pour choisir cette fitness nous nous sommes inspirés de la démarche utilisée dans le cours pour résoudre le problème des Q reines. En effet, alors que la fonction de coût était le nombre de paires de reines qui s'attaquent, la fitness correspondait au nombre de paires de reines qui ne s'attaquaient pas. Nous avons considéré que cette formalisation s'adapte bien au problème de coupe cycle.

- ***Le crossover :***

  Il s'agit de générer des enfants en réalisant un croisement entre les deux parents. Pour l'implémentation, nous avons choisies de représenter les états sous forme de graphe (avec la bibliothèque Networkx) et non pas sous forme de chaîne de caractères comme proposé dans le cours. En effet, nous avons trouvé que le format de chaîne de caractère n'est pas le plus simple pour faire les reproductions et mutations avec ce problème.

  Pour le principe et l'implémentation du crossover, nous nous sommes inspirés du problème des règnes expliqué en cours.

  Soit p1 le parent ayant le petit nombre de nœuds et p2 le parent ayant le plus grand nombre de nœuds. La taille d'un graphe représente son nombre de nœuds. Nous notons n1 la taille de p1 et n2 la taille de p2 avec n1 < n2.

  Au début du crossover, nous choisissons un nombre aléatoire entre 1 et la taille minimum des deux parents moins 1. Soit ***n*** ce nombre aléatoire. Ainsi, nous avons défini le premier enfant comme étant la combinaison des **n** premiers nœuds de p1 et des ***n2-n*** derniers nœuds de p2. Pour le deuxième enfant, nous prenons le

complémentaire des éléments restants, c'est-à-dire qu'il sera composé des ***n*** premiers nœuds de ***p2*** et des ***n1-n*** derniers nœuds de **p1**.

- ***La mutation :***

  Nous avons défini la mutation d'un graphe G comme la substitution d'un nœud de G choisi aléatoirement parmi les nœuds inclus dans des cycles par un nœud du graphe initial qui n'est pas impliqué dans un cycle, choisi de manière aléatoire aussi. Nous avons choisi de la caractériser comme ceci car, pour nous, la mutation doit améliorer notre potentielle solution pour optimiser l'algorithme.

C - Choix de représentation

Après la phase de recherche qui nous a permis de bien comprendre le sujet, nous avons formalisé le problème pour bien définir les termes de nos algorithmes de recherche comme décrit dans la partie précédente. Par la suite, nous avons choisi la représentation des graphes avec la bibliothèque Networkx en Python. En effet, après avoir suivi les recommandations du sujet et effectué quelques recherches sur internet, notre choix s'est porté sur cette bibliothèque qui fournit toutes les fonctions nécessaires à l'implémentation de nos algorithmes.

II – Différentes stratégies pour optimiser les algorithmes

Nous allons présenter dans cette partie l'ensemble des stratégies que nous avons utilisées pour améliorer nos algorithmes et les rendre optimaux. Nous présenterons également à la fin de cette partie des stratégies que nous avons imaginées et que nous pensons utiles pour optimiser les algorithmes mais que nous n'avons malheureusement pas eu le temps d'implémenter et de tester.

Stratégie 1 : Optimisation de l'algorithme génétique

Pour optimiser notre algorithme génétique, nous nous sommes concentrés sur l'amélioration de la fonction mutation. En effet, comme nous l'avons décrit plus haut, nous avons défini la mutation comme la substitution d'un nœud inclus dans un cycle avec un nœud non inclus dans un cycle et initialement non présent dans le graphe à muter. Avec cette stratégie, au lieu de se baser sur le hasard pour le choix du nœud à substituer, nous choisissons le nœud qui est inclus dans le plus grand nombre de cycles. Cette stratégie présente l'avantage de faire tendre l'algorithme génétique très rapidement vers l'optimum car le nœud choisi pour la substitution permet d'éclater le plus grand nombre de cycles. Vous retrouverez dans la partie expérimentation certains résultats obtenus avec cette stratégie.

Stratégie 2 : Lancer plusieurs fois l'algorithme sur le même graphe et prendre la meilleure solution parmi toutes

La deuxième stratégie d'optimisation que nous avons utilisée a été appliquée sur les trois algorithmes. Nous nous sommes inspirées de celle proposée dans le sujet pour la réaliser. Ainsi, pour améliorer les résultats de nos algorithmes de recherche locale, nous avons décidé, au lieu de lancer une seule fois l'algorithme, de le lancer un nombre d'itérations **I** fixé. Nous posons également au départ une limite **L**, cette limite correspond au nombre de nœuds maximum que doivent contenir les solutions de nos algorithmes. A chaque itération, nous lançons l'algorithme, nous regardons si la taille de la solution est plus petite que L, si c'est le cas alors nous remplaçons L par la taille de cette solution sinon nous laissons L tel qu'il était. Ainsi, au fur et à mesure des itérations la limite évolue et s'améliore, à la dernière itération nous renvoyons la solution qui a donné la plus petite limite L. Ainsi, les enjeux dans cette stratégie sont le choix du nombre d'itérations et de la limite de départ. Si I est trop petit, cela risque de ne pas être suffisant pour avoir la solution optimale. De la même façon si la limite L choisit au départ est trop petite, on prend le risque qu'elle soit plus petite que la taille de l'optimum, ce qui n'est pas du tout bon car on ne trouvera dans ce cas jamais la solution optimale. Pour éviter ce problème, nous avons fixé L au départ à la taille du graphe pour être sûr de trouver la bonne solution si elle existe. Dans la partie expérimentation, nous montrons différents résultats obtenus avec cette stratégie.

Stratégie 3 : Utiliser une sorte de Backtracking pour améliorer les performances

Cette stratégie n'a pas été implémentée par manque temps mais elle reste néanmoins une solution à laquelle nous avons pensé pour optimiser les algorithmes. En effet, nous nous sommes inspirées de la notion de *Backtracking Search* pour cette stratégie. L'idée est de lancer qu'une seule fois le graphe et à chaque solution trouvée de stocker cette dernière avant de retourner en arrière dans la recherche et choisir un autre voisin (de façon aléatoire) pour voir si une meilleure solution ne se trouve pas ailleurs. A la fin, la solution ayant le plus petit nombre de nœuds est renvoyée. Ainsi, l'avantage de cette technique est que nous sommes sûrs de trouver la solution optimale car on cherche toutes les solutions possibles. Cependant, cette stratégie présente le désavantage de la complexité. En effet, tester tous les chemins possibles pour avoir la bonne solution peut s'avérer très coûteux en termes de complexité, d'autant plus si le graphe à un grand nombre de nœuds.

III – Protocole expérimental

Pour l'expérimentation, nous avons décidé de générer des graphes à partir du modèle probabiliste d'**Erdos-Rényi**. Pour cela nous avons utilisé la fonction *fast_gnp_random_graph* de la bibliothèque *Networkx*.

Notre expérimentation se divise en deux parties :

1) D'une part, nous comparons nos différents algorithmes en utilisant les stratégies d'optimisation que nous avons évoqué plus haut. Ainsi, pour cela, nous générons une

instance de graphe et appliquons sur ce même graphe les trois algorithmes. Nous appliquons sur les trois algorithmes la stratégie numéro deux que nous avons présenté plus haut. C'est-à-dire nous fixons un nombre d'itérations à effectuer et une limite initiale qui évolue au cours du temps. Pour l'algorithme génétique, en plus, d'y appliquer la stratégie que nous venons d'évoquer, nous utilisons la version améliorée de l'algorithme où la fonction de mutation est implémenté comme décrite dans la stratégie 1 plus haut dans le rapport. Ainsi, l'objectif de cette partie d'expérimentation est d'évaluer et comparer les différents algorithmes dans leur meilleure version possible et voir quels paramètres influencent leurs performances.

Dans cette partie, nous avons défini différentes métriques pour permettre l'évaluation de nos méthodes :

- La moyenne des limites des solutions retournées par les algorithmes. Avec limite qui représente la taille du coupe-cycle.
- La moyenne temporelle des exécutions des algorithmes.

A chaque expérimentation dans cette partie, nous faisons varier différents paramètres pour voir leur impact sur les performances des différents algorithmes. Dans la partie expérimentations nous détaillerons les paramètres que nous avons fait varier et les résultats obtenus.

2) D'autres part, au lieu de faire des tests sur une seule instance du graphe, nous générons plusieurs instances différentes de graphes et testons les performances temporelles des algorithmes dessus. Cela permet de voir quels algorithmes sont vraiment bon en termes de complexité temporelle par rapport à d'autres. Ainsi les métriques que nous utilisons sont :

- La moyenne temporelle des exécutions des algorithmes.
- L'écart-type temporelle des exécutions des algorithmes.

Ces métriques temporelles n'ont pas la même vocation que celles présentés dans la partie 1 de nos expérimentations. Alors que dans la partie 1) elles permettent de voir pour quel type de graphe un algorithme est plus rapide que l'autre, ici cela permet de voir les performances temporelles générales.

IV – Expérimentations, résultats et conclusions

Partie 1 – Évaluation et comparaison des algorithmes améliorés par les différentes stratégies.

Pour comparer les algorithmes en termes de ***complexité temporelle***, nous stockons pour chaque algorithme, à chaque itération, le temps que l'algorithme met pour trouver une solution pour le graphe donné. Nous effectuons ensuite une moyenne de tous les temps d'exécutions, pour chaque algorithme.

Notre hypothèse de départ est que le *Hill Climbing* devrait être en général celui dont la complexité temporelle est la plus grande des trois. En effet, parmi les trois algorithmes, le *Hill*

Climbing est le seul, pour un état courant donné, à parcourir tous les voisins de l'état courant pour trouver celui qui minimise la fonction de coût. Cette étape de parcours et de tri de l'ensemble des voisins est très coûteuse en termes de complexité temporelle pour le *Hill Climbing*. De son côté, le *Simulated Annealing* ne faisant pas cette étape de tri et choisissant le voisin de manière aléatoire devrait avoir en général les meilleurs résultats en termes de complexité temporelle. Pour l'algorithme Génétique, nous nous doutions que ça complexité allait être très dépendante de la taille de la population initiale choisit. En effet, plus la taille de cette dernière est petite plus l'algorithme sera rapide voir très rapide. Cependant si la taille de la population est grande, l'algorithme pourra devenir très lent, car il aura une population plus grande, ce qui nécessitera un tri de fitness très grand, ce qui rendra l'algorithme très lent.

Les résultats ci-dessous confirment bien nos hypothèses sur la complexité temporelles. Nous avons testé les algorithmes sur un graphe de 30 nœuds avec 10 itérations :

Résultat 1 :

```
Saisir le nombre de noeuds dans le graphe : 30
Saisir la proba de mise en place d'arc : 0.2
Saisir la taille de la population initiale pour l'aglo génètique : 24
Saisir le nombre d'itérations  : 10
Temps moyen pour l'algorithme Hill Climbing =  1.2292333602905274  ms
Temps moyen pour l'algorithme Simulated Annealing =  0.4599935531616211  ms
Temps moyen pour l'algorithme Génètique =  0.5219576835632325  ms
```

Résultat 2 :

```
Saisir le nombre de noeuds dans le graphe : 30
Saisir la proba de mise en place d'arc : 0.2
Saisir la taille de la population initiale pour l'aglo génètique : 8
Saisir le nombre d'itérations  : 10
Temps moyen pour l'algorithme Hill Climbing =  1.1563944101333619  ms
Temps moyen pour l'algorithme Simulated Annealing =  0.37482926845550535  ms
Temps moyen pour l'algorithme Génètique =  0.14693665504455566  ms
```

Nous voyons dans les deux résultats que l'algorithme *Hill Climbing* est très lent par rapport aux autres méthodes de recherches, près de 3 fois plus lent que le *Simulated Annealing* et 2.5 fois plus lent que *l'algorithme Génétique*. De même, le *Simulated Annealing* répond bien à nos hypothèses de départ, il est plutôt bon dans les deux résultats. La seule différence entre l'expérience 1 et l'expérience 2 est la taille de la population initiale de *l'algorithme génétique*. En effet, comme nous l'avions prédit, la taille de la population a une grande influence sur les performances temporelles de *l'algorithme génétique*. En divisant par 3 la taille de la population initiale, la complexité temporelle a été divisée par quasiment 4. Ainsi, sur l'aspect temporelle nos hypothèses de départ sont bien vérifiées.

Nous avons ensuite décidé de comparer par rapport à l'***optimalité*** des solutions qu'ils retournent. En effet, cela faisait partie des aspects importants du projet, le fait d'optimiser les solutions trouvées. Notre hypothèse au départ était que *l'algorithme génétique* devait être en général le plus optimal. En effet, c'est un algorithme qui utilise à la fois la diversification, la qualité des individus et la mutation pour retourner l'individu qui répond le mieux au problème. De la même façon, nous considérions que le *Hill Climbing* devait donner de bons résultats

aussi, car choisissant le meilleur voisin à chaque étape, l'algorithme est censé à la fin donner une solution très optimale. Cependant, effectuant une recherche 100% locale, il n'y a aucune diversification qui est réalisée, raison pour laquelle nous pensions que le *Génétique* devrait être tout même meilleur que le *Hill Climbing*. Pour le *Simulated Annealing*, c'est selon nous l'algorithme le moins optimal par rapport au problème parmi les trois. En effet, le fait que le voisin soit choisi de manière aléatoire et affecté à l'état courant avec une certaine probabilité qui peut être non négligeable lors des premières itérations de l'algorithme, fait qu'il est assez dépendant du hasard malgré la fonction Schedule qui est construite derrière pour faire performer l'algorithme. Voici deux exemples de résultats par rapport à cette expérimentation :

Résultat 1 :

```
Saisir le nombre de noeuds dans le graphe : 10
Saisir la proba de mise en place d'arc : 0.2
Saisir la taille de la population initiale pour l'aglo génètique : 20
Saisir le nombre d'itérations  : 20

Moyenne évolution limite Hill Climbing =  2.0
Moyenne évolution limite  Simulated Annealing =  5.25
Moyenne évolution limite  Genetic Algorithm =  3.25
```

Résultat 2 :

```
Saisir le nombre de noeuds dans le graphe : 60
Saisir la proba de mise en place d'arc : 0.2
Saisir la taille de la population initiale pour l'aglo génètique : 100
Saisir le nombre d'itérations  : 10

Moyenne évolution limite Hill Climbing =  47.4
Moyenne évolution limite  Simulated Annealing =  49.3
Moyenne évolution limite  Genetic Algorithm =  38.0
```

Les résultats confirment bien notre hypothèse initiale sur le *Simulated Annealing*. Que ce soit pour un petit graphe avec 10 nœuds ou un grand graphe avec 60 nœuds, les solutions qu'il retourne sont certes valides mais moins optimales que les deux autres algorithmes. Cependant les résultats sont légèrement différents de ce que nous attendions concernant les hypothèses sur *l'algorithme Génétique* et le *Hill Climbing*. Comme nous pouvons le voir, le *Génétique* est très bon pour de gros graphe, il donne des solutions avec près de 10 nœuds en moins que les autres algorithmes. Cependant, pour les petits graphes, le *Hill Climbing* présente des solutions plus optimales comme le montre le premier résultat.

En dernière expérimentation sur cette partie, nous nous sommes focalisés sur *l'algorithme Génétique.* En effet, nous voulions confirmer ou réfuter certaines hypothèses que nous avions sur l'algorithme. Effectivement, nous nous doutions que la taille de la population initiale aurait un fort impact sur les performances de l'algorithme. Nous avons vu plus haut que moins la population est grande plus la complexité temporelle de l'algorithme est bonne. Cependant, nous pensons qu'augmenter la taille de la population permet d'améliorer nettement les performances de l'algorithme car cela permet d'augmenter la diversification et donc la chance de tomber sur les meilleurs individus. Ainsi, nous avons joué sur la taille de la population pour

voir l'effet de la diversification sur la qualité des solutions retournées par l'algorithme génétique.

Voici quelques résultats de nos expérimentations :

Résultat 1 :

```
Saisir le nombre de noeuds dans le graphe : 39
Saisir la proba de mise en place d'arc : 0.1
Saisir la taille de la population initiale pour l'aglo génètique : 20
Saisir le nombre d'itérations  : 10
Moyenne nombre de noeuds à supprimer Genetic Algorithm =  22.2
Ecart type nombre de noeuds à supprimer Genetic Algorithm =  2.3151673805580453
```

Résultat 2 :

```
Saisir le nombre de noeuds dans le graphe : 39
Saisir la proba de mise en place d'arc : 0.1
Saisir la taille de la population initiale pour l'aglo génètique : 40
Saisir le nombre d'itérations  : 10
Moyenne nombre de noeuds à supprimer Genetic Algorithm =  22.6
Ecart type nombre de noeuds à supprimer Genetic Algorithm =  2.009975124224178
```

Résultat 3 :

```
Saisir le nombre de noeuds dans le graphe : 39
Saisir la proba de mise en place d'arc : 0.1
Saisir la taille de la population initiale pour l'aglo génètique : 9
Saisir le nombre d'itérations  : 10
Moyenne nombre de noeuds à supprimer Genetic Algorithm =  22.7
Ecart type nombre de noeuds à supprimer Genetic Algorithm =  2.6476404589747453
```

Dans les résultats nous voyons que la diversification de la population a un impact positif sur l'optimalité de l'algorithme. En effet, lorsque la taille de la population initiale est de 40 individus nous avons un écart-type des solutions de 2.01, lorsqu'elle est de 20 individus l'écart-type est de 2.32 et enfin lorsque nous choisissons une population très petite de 9 individus, nous voyons que l'écart à la moyenne est plus grand avec 2.65. Ainsi, plus la population est grande, plus les performances sont bonnes.

Partie 2 – Évaluation et comparaison des complexité temporelles des algorithmes sur différentes instances avec les mêmes paramètres

Le but de cette expérimentation, était de voir de manière générale sur n'importe quel graphe, quel algorithme est le plus performant en termes de complexité temporelle. En testant les algorithmes sur 10 instances de graphes différentes avec un modèle probabiliste, nous voyons que *l'algorithme Génétique* est très bon en termes de complexité temporelle qu'il ait une grande ou une petite taille de population initiale. Le *Hill Climbing* est quant à lui très lent par rapport aux autres. Voici quelques résultats :

Résultat 1 :

```
Saisir le nombre de noeuds dans le graphe : 39
Saisir la proba de mise en place d'arc : 0.1
Saisir la taille de la population initiale pour l'aglo génètique : 9
Saisir le nombre d'itérations  : 10

Temps moyen pour l'algorithme Hill Climbing =  2.4782230377197267  ms
Temps moyen pour l'algorithme Simulated Annealing =  0.7009679317474365  ms
Temps moyen pour l'algorithme Génètique =  0.24632713794708253  ms
Ecart type temps pour l'execution de l'algorithme Hill Climbing =  0.4335782500159609
Ecart type temps pour l'execution de Simulated Annealing =  0.1137640221982547
Ecart type temps pour l'execution de Génètique =  0.21972208147936384
```

Résultat 2 :

```
Saisir le nombre de noeuds dans le graphe : 39
Saisir la proba de mise en place d'arc : 0.1
Saisir la taille de la population initiale pour l'aglo génètique : 40
Saisir le nombre d'itérations  : 10

Temps moyen pour l'algorithme Hill Climbing =  2.130848836898804  ms
Temps moyen pour l'algorithme Simulated Annealing =  0.6039429426193237  ms
Temps moyen pour l'algorithme Génètique =  0.20241994857788087  ms
Ecart type temps pour l'execution de l'algorithme Hill Climbing =  0.2410344360214338
Ecart type temps pour l'execution de Simulated Annealing =  0.18880984243489346
Ecart type temps pour l'execution de Génètique =  0.19715376828724862
```

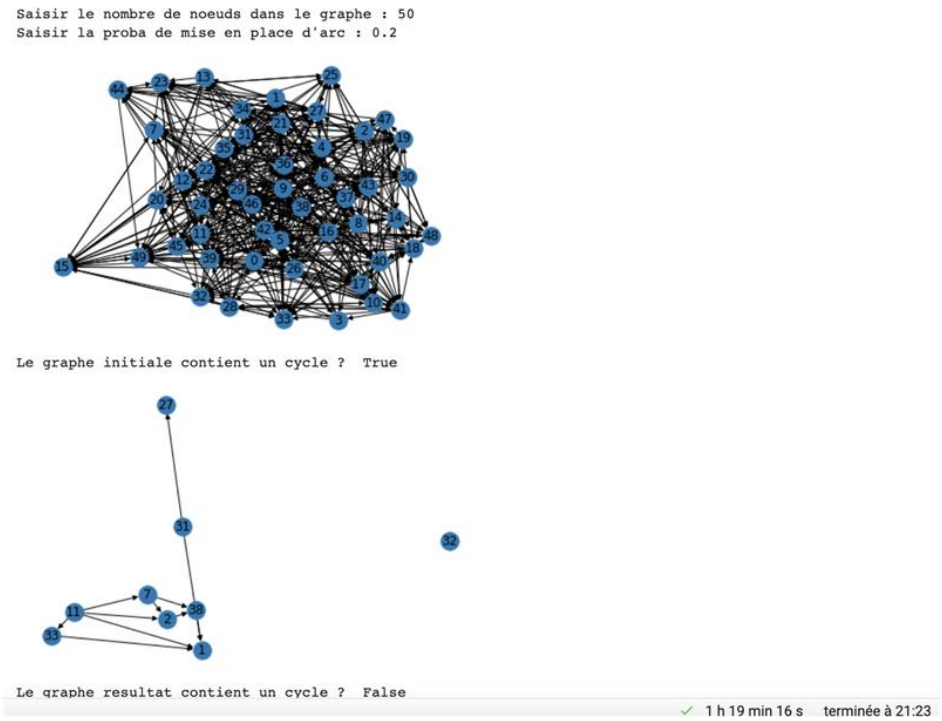
Ainsi, nous déduisons de ces résultats que *l'algorithme Génétique* est celui qui excelle en termes de complexité temporelle. Il est suivi du *Simulated Annealing* qui présente d'assez bons résultats loin devant la complexité temporelle du *Hill Climbing* qui est près de 3.5 fois plus long.

Ainsi, nous avons présenté dans cette partie nos différentes hypothèses et les résultats qui ont confirmés certaines et réfutés d'autres.

V – Difficultés rencontrées

Nous avons rencontré un grand nombre de difficultés au cours du projet. Ces difficultés étaient principalement fonctionnelles et non techniques.

L'algorithme Simulated Annealing est celui qui a été le plus difficile à coder. En effet, nous avons au départ eu du mal à nous détacher du pseudo code fourni dans le cours et d'adapter l'algorithme au problème donné. Par exemple, au départ, notre condition d'arrêt était uniquement basée sur le fait que la température soit nulle à un moment donné. Cependant, cette condition n'était pas suffisante pour notre problème car il se peut que la température ne soit pas nulle mais qu'une solution valide soit trouvée. Ainsi, lorsque nous testions notre algorithme, nous avions toujours des solutions extrêmement grandes (en nombre de nœuds) car nous n'avions pas ajouté dans la condition d'arrêt concernant les cycles du graphe. Voici un exemple de résultat aberrant que nous obtenions.



Dans l'exemple, le graphe initial contient 50 nœuds avec une probabilité d'arc de 0.2. Après une heure et vingt minutes, l'algorithme nous fournit un graphe acyclique avec seulement 9 nœuds. Nous avons résolu ce problème en ajoutant dans la condition d'arrêt que si le graphe est acyclique alors il faut renvoyer la solution.

Cependant, la plus grande difficulté rencontrée avec le Simulated Annealing est le dosage de la température et le choix de la fonction Schedule. En effet, nous avons eu beaucoup de mal à trouver une fonction Schedule qui ne fasse pas baisser trop rapidement la température. Au départ nous étions parties sur une fonction Schedule définit de la manière suivante : Soit T une température donnée, on a

Schedule(T) = alpha * T avec alpha le facteur de refroidissement appartenant]0,1[.

Nous avions beau fixer une température initiale élevée et un alpha très proche de 1, l'algorithme s'arrêtait toujours sans avoir pû trouver une solution valide.

Nous avons ensuite abandonnée l'idée du facteur de refroidissement, nous avons défini notre nouvelle Schedule de la façon suivant : Soit t une variable représentant le temps,
on a

$$Schedule(T) = 1/(1+t)$$

Avec cette fonction Schedule, les résultats étaient meilleurs mais toujours pas suffisant pour trouver la meilleure solution. Nous avons donc redéfini notre fonction Schedule de la façon suivante : Soit t une variable représentant le temps, on a

$$Schedule(T) = 100/\ln(1+t)$$

Cette fonction Schedule est la plus optimisée que nous avons trouvé. En effet la fonction ln étant une fonction qui croît très lentement, diviser par cette dernière permet d'avoir une fonction Schedule qui décroît très lentement et c'est ce que nous recherchions. La multiplication par 100 permet de ralentir encore plus la décroissance. Avec ce choix de fonction, nous avons pû avoir des solutions toujours valides et souvent optimales.

Pour l'algorithme génétique, nous avons rencontré un certain nombre de difficultés liées à la complexité de l'algorithme

Le principal problème était lié à la fonction de reproduction. En effet, au moment de la création des deux parents nous avons remarqué que certaines informations des parents étaient perdues avec la suppression de certains nœuds. Par exemple : Certaines arêtes du graphe initial disparaissaient au fur et à mesure des reproductions alors que les nœuds impliqués dans les arêtes sont dans l'enfant résultant de la reproduction. Pour pallier cette perte d'information, qui faussait nos résultats, nous nous sommes aussi basés sur les graphes parents pour la pallier ce problème. En effet, après avoir fait le cross over des deux parents, nous comparons les arcs de l'enfant à ceux des parents et nous rajoutons dans l'enfant toutes les arêtes manquantes, c'est-à-dire celles qui sont associés à des nœuds dans l'enfant et qui ont été perdues lors de la reproduction. Cela nous a permis de résoudre le problème.

Conclusion :

Ainsi, nous avons présentés tout au long de ce rapport notre travail et notre réflexion autour de ce projet. Nous avons également pu exposer les différentes stratégies que nous avons implémentés pour améliorer les performances des algorithmes. Dans la partie expérimentation nous avons mis en avant les différentes hypothèses que nous avions sur les performances des algorithmes. Certains résultats ont appuyé nos suppositions de départ, d'autres les ont réfutés en nous faisant découvrir de nouvelles propriétés sur les algorithmes. Enfin, dans la partie difficultés, nous avons pu montrer les différentes complications rencontrées tout au long du projet que ce soit en terme de compréhension ou de technicité.