INF8775 – Analyse et conception d'algorithmes

TP2 – Automne 2021

Nom, prénom, matricule des membres	Desgagné, Kassandra, 1897918 Tremblay-Rheault, Maxime, 1946878
Note finale / 17	0

Informations techniques

Répondez directement dans ce document ODT avec LibreOffice. Veuillez ne pas inclure le texte en italique servant de directive.

La correction se fait à même le rapport.

Vous devez faire une remise électronique avant le 17 novembre 23h59 en suivant les instructions suivantes:

- Le dossier remis doit se nommer matricule1_matricule2_tp2 et doit être compressé sous format zip.
- À la racine de ce dernier, on doit retrouver :
  - Ce rapport sous format ODT.
  - Un script nommé *tp.sh* servant à exécuter les différents algorithmes du TP. L'interface du script est décrite à la fin du rapport.
  - Le code source et les exécutables

Vous avez le choix du langage de programmation utilisé mais vous devrez utiliser les mêmes langage, compilateur et ordinateur pour toutes vos implantations. Le code et les exécutables soumis devront être compatible avec les ordinateurs de la salle L-4714.

Si vous utilisez des extraits de codes (programmes) trouvés sur Internet, vous devez en mentionner la source, sinon vous serez sanctionnés pour plagiat.

Présentation des résultats

0	/ 5 pt

Tableau des résultats

Tableau #1 : Temps moyen d'exécution selon le nombre de sommets pour chaque algorithme

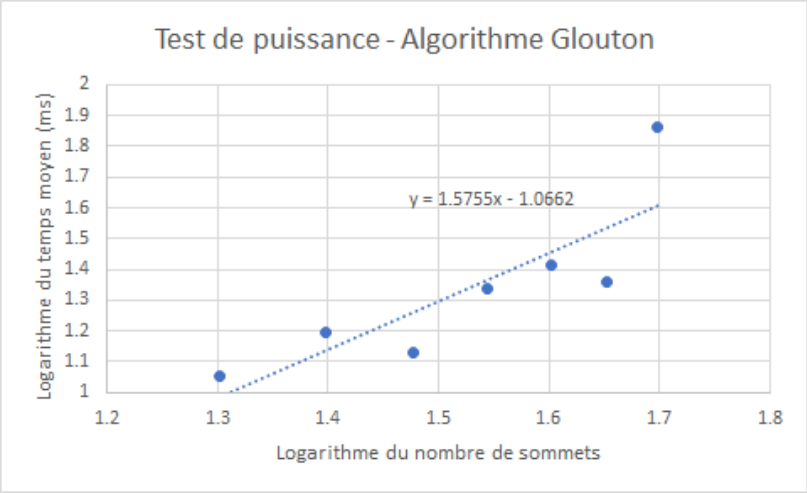
Algorithme	Nombre de sommets	Temps d'exécution(ms)	Nombre min de couleurs utilisées
Glouton	20	11.35	11.8
Branch and bound		7.66	11.8
Recherche Tabou		149.17	11.8
Glouton	25	15.69	14
Branch and bound		23.56	13.8
Recherche Tabou		399.70	14
Glouton	30	13.46	16.8
Branch and bound		126.96	16
Recherche Tabou		1542.68	16.4
Glouton	35	21.62	18
Branch and bound		124.81	17.8
Recherche Tabou		2035.28	18
Glouton	40	26.07	21
Branch and bound		4729.82	20.2
Recherche Tabou		7320.55	20.4
Glouton	45	23.00	22.8
Branch and bound		3327.76	21.2
Recherche Tabou		15684.96	22
Glouton	50	72.75	24
Branch and bound		306181.35	22.6
Recherche Tabou		19226.08	23.4

Tableau #2 : Temps moyen d'exécution selon l'algorithme pour chaque nombre de sommets

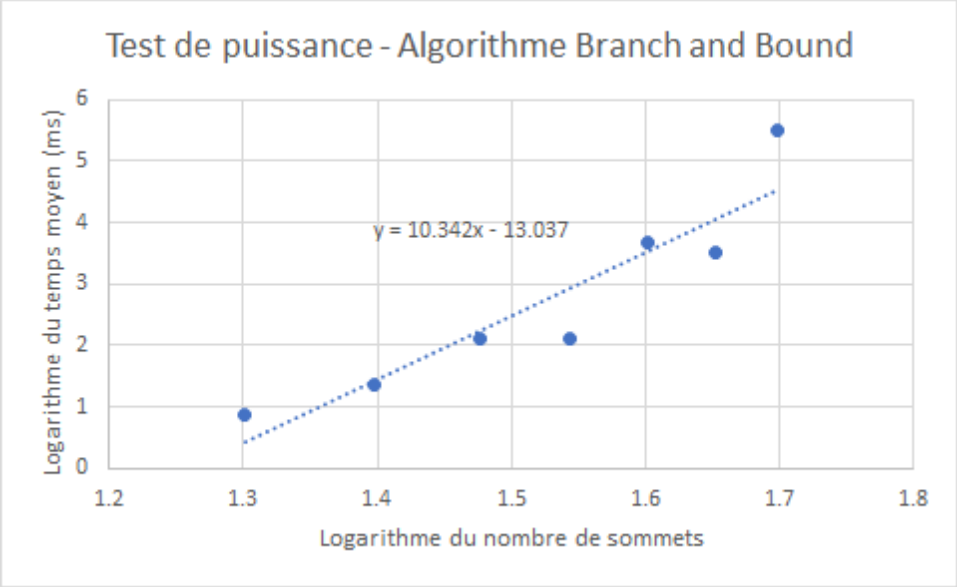
Algorithme	Nombre de sommets	Temps d'exécution(ms)	Nombre min de couleurs utilisées
Glouton	20	11.35	11.8
	25	15.69	14
	30	13.46	16.8
	35	21.62	18
	40	26.07	21
	45	23.00	22.8
	50	72.75	24
Branch and bound	20	7.66	11.8
	25	23.56	13.8
	30	126.96	16
	35	124.81	17.8
	40	4729.82	20.2
	45	3327.76	21.2
	50	306181.35	22.6
Recherche Tabou	20	149.17	11.8
	25	399.70	14
	30	1542.68	16.4
	35	2035.28	18
	40	7320.55	20.4
	45	15684.96	22
	50	19226.08	23.4

Graphiques pour l'analyse hybride

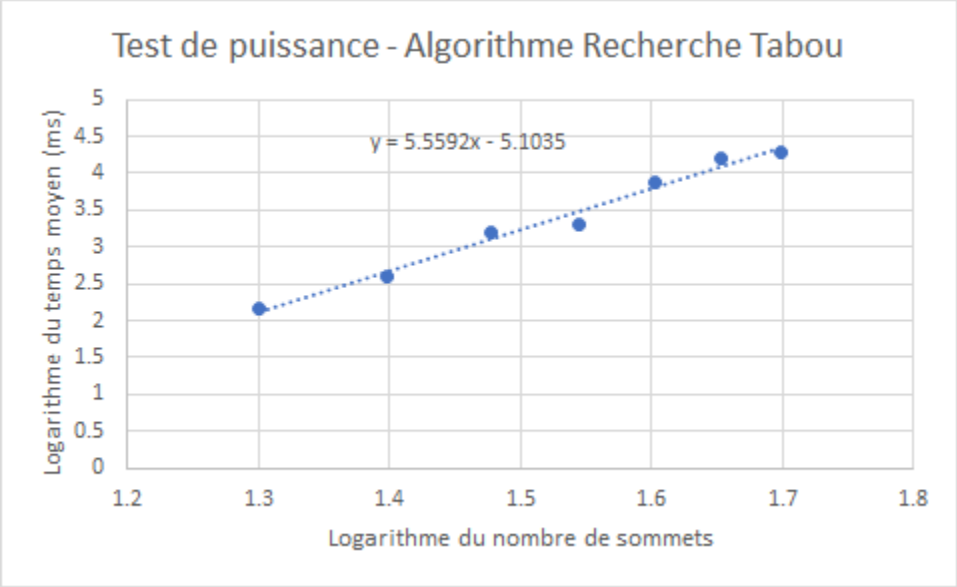
Tests de puissance



Graphique #1 : Test de puissance pour l'algorithme glouton

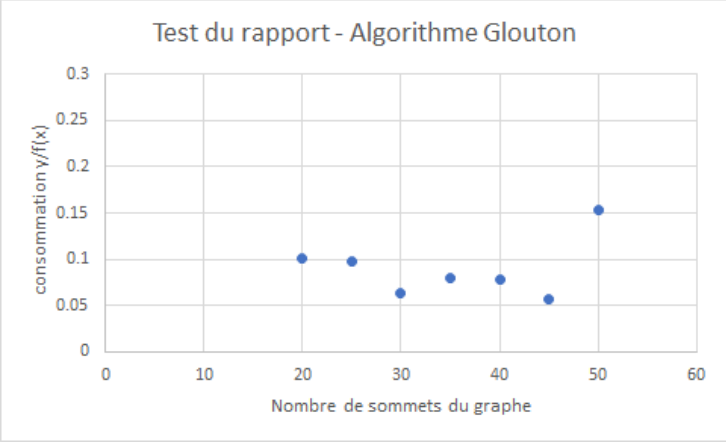


Graphique #2 : Test de puissance pour l'algorithme branch and bound

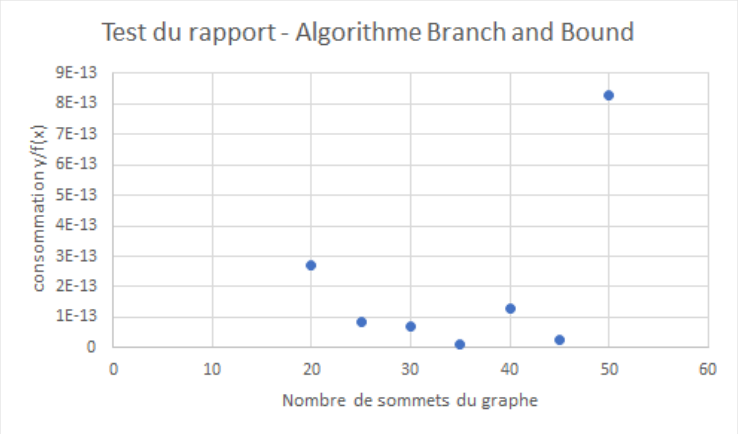


Graphique #3 : Test de puissance pour l'algorithme recherche tabou

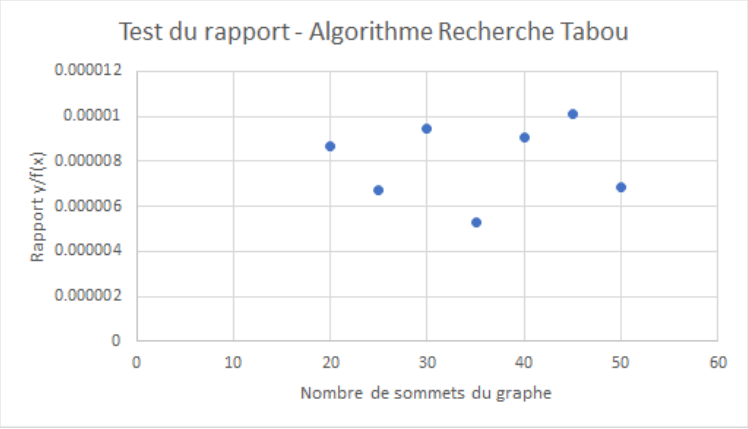
Tests du rapport



Graphique #4: Test du rapport pour l'algorithme glouton

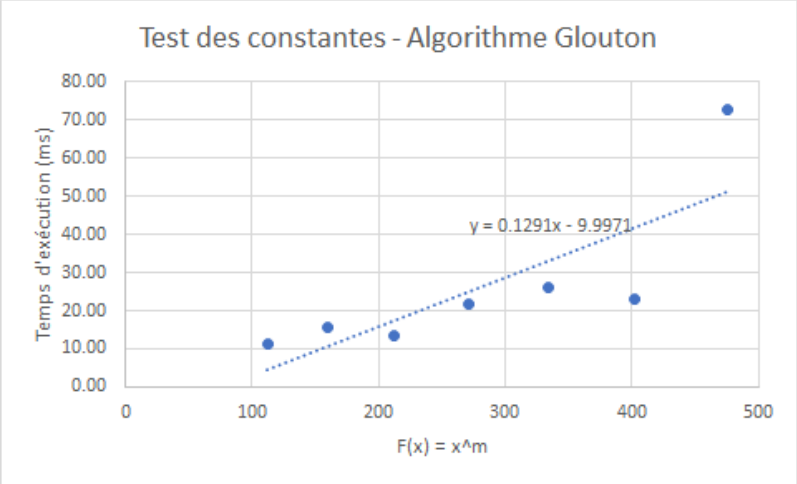


Graphique #5 : Test du rapport pour l'algorithme branch and bound

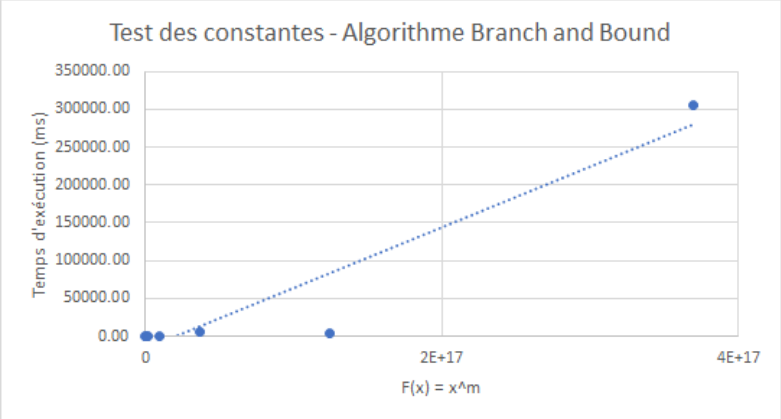


Graphique #6 : Test du rapport pour l'algorithme recherche tabou

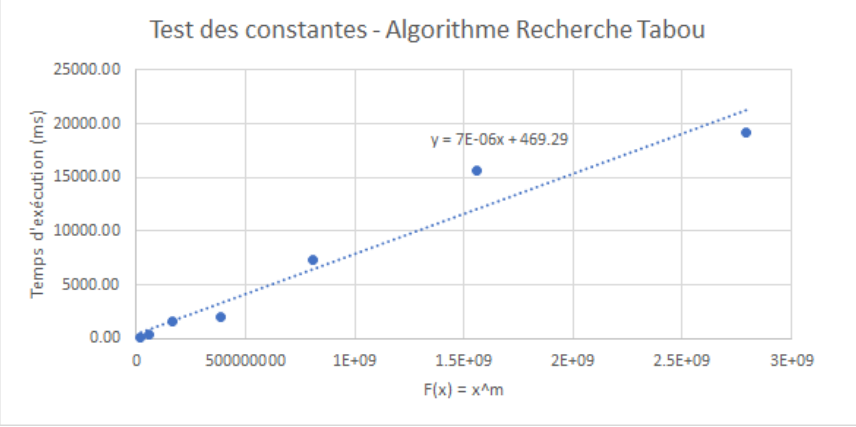
Tests des constantes



Graphique #7: Test des constantes pour l'algorithme glouton



Graphique #8 : Test des constantes pour l'algorithme branch and bound



Graphique #9 : Test des constantes pour l'algorithme recherche tabou

Analyse et discussion

0	/ 8 pt

Tentez une analyse asymptotique du temps de calcul pour chaque algorithme.

Pour les 3 algorithmes implémentés dans ce laboratoire, l'analyse théorique sera faite en pire cas seulement. On va alors poser les hypothèses suivantes :

1. Le graphe en entrée est complet.
2. Le résultat de l'algorithme glouton (pour l'algorithme branch and bound et liste tabou) utilise une couleur par sommet.
3. Les algorithmes prennent en entrée un graphe G = {V,E}.
4. La densité d'arêtes est de 0.9. On peut alors déduire le nombre d'arêtes en fonction du nombre de sommets d'un graphe avec la formule suivante :

$$D = \frac{2|E|}{|V|\cdot(|V|-1)}$$

$$|E| = \frac{D\cdot|V|\cdot(|V|-1)}{2}$$

Une complexité trouvée appartenant à O(E) est équivalente à O(0.9 * V*(V-1) / 2) ce qui est équivalent à $O(V^2)$ à une constante multiplicative près.

Analyse algorithme glouton

Figure #1 : Code de l'algorithme glouton

```
3  def glutton(graph):
4      v = Helper.getVerticeWithMaxDegree(graph)
5      coloration = {}
6      coloration[v] = 0
7
8      while len(coloration) < len(graph):
9          v = Helper.findMaxSaturatedVertice(graph, coloration)
10         neighboursColor = []
11         for neighbour in graph[v]:
12             if neighbour in list(coloration.keys()):
13                 neighboursColor.append(coloration[neighbour])
14         for i in range(len(coloration) + 1):
15             if i not in neighboursColor:
16                 coloration[v] = i
17                 break
18
19     return coloration
20
```

Tableau #3 : Analyse théorique ligne par ligne de l'algorithme glouton

# Ligne de l'instruction	Complexité asymptotique	Justification
4	$\Theta(V)$	On parcourt chaque sommet du graphe pour trouver le sommet ayant le plus grand degré.
5	$\Theta(1)$	Initialisation.
6	$\Theta(1)$	Assignation.
8-17	$\Theta(V)$	On assigne une couleur à chaque sommet du graphe alors on va parcourir la boucle pour V fois.
9	$\Theta(V)*O(E) \approx \Theta(V)*O(V^2) \approx O(V^3)$	Pour chaque sommet du graphe on parcourt les arêtes connectées pour calculer le degré de saturation, puis on retourne le sommet ayant le plus grand degré de saturation. Dans le pire cas, le graphe est complet et on parcourt chaque arête pour chaque sommet.
10	$\Theta(1)$	Initialisation
11-13	$O(E) \approx O(V^2)$	On parcourt les voisins du sommet ayant le plus grand degré de saturation. Dans le pire cas, on parcourt tous les sommets si le graphe est complet.
12	$\Theta(V)$	On parcourt la liste des colorations pour vérifier si le voisin s'y trouve. Dans le pire cas, la liste de coloration contient tous les sommets du graphe et en moyenne elle contient V/2 sommets. À une constante multiplicative près on a donc une complexité $\Theta(V)$.
13	$\Theta(1)$	Ajout d'un élément dans la liste.
14-17	$\Theta(V)*\Theta(V) = \Theta(V^2)$	Pour chaque couleur déjà attribuée qui est en pire cas de la taille V et en moyenne de taille V/2, on cherche la plus petite couleur à assigner au sommet. Dans le pire cas, on assigne une nouvelle couleur.
15	$\Theta(V)$	On parcourt la liste des voisins pour voir si la couleur est assignée à un voisin du sommet. Dans le pire cas, le graphe est complet.

16	Θ(1)	Assignation.
17	Θ(1)	On quitte la boucle.
19	Θ(1)	On retourne la coloration.

Selon la règle du maximum O(f+g) = O(max(f,g)) on peut regarder seulement l'instruction qui prend le plus de temps dans la boucle while. Pour les lignes 9-17 on peut donc dire que la complexité est en $O(V^2)$.

Avec la formule h1*h2 ∈ Θ(f*g) on peut aussi dire que la boucle while est de complexité O(V) * $O(V^2)$ = $O(V^3)$.

La complexité de l'algorithme glouton est donc de l'ordre de $O(V^3)$

Analyse algorithme branch and bound

Figure #2 : Code de l'algorithme explore_node

```
14  def explore_node(graph, coloration):
15      node_list = []
16      vertice = Helper.findMaxSaturatedVertice(graph, coloration)
17      for i in range(len(coloration) + 1):
18          if not isColorIsConflicting(i, vertice, graph, coloration):
19              colorationCopy = copy.deepcopy(coloration)
20              colorationCopy[vertice] = i
21              node_list.append(colorationCopy)
22      return node_list
```

Figure #3 : Code de l'algorithme branch and bound

```
32  def branch_bound(G) :
33      currentBestSolution = glutton(G)
34      UB = Helper.findNbOfUniqueColorsInSolution(currentBestSolution) # Upper bound
35
36      node_pile = []
37      coloration = {}
38      vertice = Helper.getVerticeWithMaxDegree(G)
39      coloration[vertice] = 0
40      node_pile.append(coloration) # LIFO for deep search
41      while node_pile:
42          coloration = node_pile.pop()
43
44          if checkIfAllVerticesInSolution(coloration, G):
45              nbOfUniqueColorsFound = Helper.findNbOfUniqueColorsInSolution(coloration)
46              if nbOfUniqueColorsFound < UB:
47                  UB = nbOfUniqueColorsFound
48                  currentBestSolution = coloration
49
50          elif Helper.findNbOfUniqueColorsInSolution(coloration) < UB:
51              node_list = explore_node(G, coloration)
52              for new_coloration in node_list:
53                  node_pile.append(new_coloration)
54
55      return currentBestSolution
```

Tableau #4 : Analyse théorique ligne par ligne de l'algorithme branch and bound

# Ligne de l'instruction	Complexité asymptotique	Justification
15	$\Theta(1)$	Initialisation.
16	$\Theta(V)*O(E) \approx O(V) * O(V^2) \approx O(V^3)$	Pour chaque sommet du graphe, on parcourt les arêtes connectées pour calculer le degré de saturation, puis on retourne le sommet ayant le plus grand degré de saturation. Dans le pire cas, le graphe est complet et on parcourt chaque arête pour chaque sommet.
17-21	$\Theta(V)$	On parcourt la liste de coloration dans l'ordre de $\Theta(V/2)$ en moyenne.
18	$O(V)$	Dans le pire cas, le graphe est complet et on doit parcourir tous les sommets pour vérifier si la couleur est en conflit avec celle de son voisin.
19	$\Theta(V)$	On copie le tableau de coloration qui est de la taille du nombre de sommets.
20	$\Theta(1)$	Assignation.
21	$\Theta(1)$	Ajout sur la pile.
22	$\Theta(1)$	Retour de la fonction
15-22	$\Theta(V^3)$	Avec $O(f+g) = O(\max(f,g))$ et $h1*h2 \in \Theta(f*g)$ on peut déduire que "explore_node()" est de complexité $\Theta(V^3)$ à cause de la fonction "findMaxSaturatedVertice".
33	$O(V^3)$	On utilise l'algorithme glouton pour avoir une première solution valide.
34	$\Theta(V)$	On parcourt la liste de coloration pour trouver le nombre de couleurs utilisées, ce qui correspond à parcourir la liste de sommets.
36	$\Theta(1)$	Initialisation.
37	$\Theta(1)$	Initialisation.
38	$\Theta(V)$	On parcourt la liste des sommets pour trouver celui qui a le degré le plus élevé.

39	$\Theta(1)$	Assignation.
40	$\Theta(1)$	Ajout sur la pile.
41-54	$O(V!)$	En pire cas, on explore toutes les combinaisons de coloration possibles à partir d'une coloration initiale. Avec une coloration initiale de taille 1, l'arbre généré en pire cas sera proportionnel à V! puisque pour chaque nouveau niveau de profondeur de l'arbre, une couleur est ajoutée[1].
42	$\Theta(1)$	On retire l'élément sur la pile.
44	$\Theta(V)$	Dans le pire cas, on parcourt tous les sommets du graphe pour déterminer s'ils se trouvent dans la coloration.
45	$\Theta(V)$	On parcourt la liste de coloration pour trouver le nombre de couleurs utilisées, ce qui correspond à parcourir la liste de sommets.
46	$\Theta(1)$	Comparaison.
47	$\Theta(1)$	Assignation.
48	$\Theta(1)$	Assignation.
50	$\Theta(V)$	On parcourt la liste de coloration pour trouver le nombre de couleurs utilisées, ce qui correspond à parcourir la liste de sommets.
51	$\Theta(V^3)$	La fonction "explore_node" a été évaluée précédemment aux lignes 15-22.
52-53	$O(V)$	Parcours de toutes les colorations trouvées par "explore_node". $O(V/2)$ en moyenne puisqu'à chaque itération, la taille de coloration augmente de 1.
53	$\Theta(1)$	Ajout sur la pile.
54	$\Theta(1)$	Retour de la fonction.

Avant le début de la boucle while, un appel à l'algorithme glouton est fait avec une complexité en pire cas $O(V^3)$, ainsi que deux appels de fonction appartenant à $O(V)$ chacun. La somme de ces complexités appartient à $O(V^3)$ selon la règle du maximum.

[1] voir explication à la page suivante

Quant à elle, la boucle while aura au plus V! itérations puisqu'une pile LIFO est bâtie à partir d'une coloration initiale. Ainsi, avec une coloration initiale {A:0}, on aura par la suite en pire cas {A:0, B:0, B:1, C:0, C:1, C:2, D:0, D:1, D:2, D:3, E:0, E:1, E:2, E:3, E:4 etc.} qui comprend V! combinaisons possibles. Toujours selon la règle du maximum, l'intérieur de la boucle while est borné supérieurement par "explore_node()" qui prend un temps en $\Theta(V^2)$.

Avec la formule h1*h2 $\in \Theta$(f*g) on peut aussi dire que la boucle while appartient à

$O(V!) * O(V^2)$.

Selon la règle du maximum O(f+g) = O(max(f,g)) on peut considérer uniquement le temps pris par la boucle while puisque celle-ci est de complexité supérieure.

Donc la complexité de l'algorithme branch and bound appartient à $O(V!*V^2)$ ce qui est considérablement supérieur à l'algorithme glouton.

Analyse algorithme de recherche tabou

Figure #4 : Code de l'algorithme tabou

```
107  def tabou(graph):
108      bestColorationWithoutConflict = glutton(graph)
109
110      #Loop between colorReduction and tabouSearch until tabouSearch fails
111      while (True):
112          newColorationWithConflict = colorReduction(graph, bestColorationWithoutConflict)
113          resultTabou = tabouSearch(graph, newColorationWithConflict)
114
115          numberOfConflictTabou = getNumberOfConflict(graph, resultTabou)
116          if (numberOfConflictTabou > 0):
117              break
118          else:
119              bestColorationWithoutConflict = copy.copy(resultTabou)
120      return bestColorationWithoutConflict
```

Tableau #5 : Analyse théorique ligne par ligne de l'algorithme tabou (incluant la réduction de couleur)

# Ligne de l'instruction	Complexité asymptotique	Justification
108	$O(V^3)$	On utilise l'algorithme glouton pour avoir une première solution valide.
111-120	O(1)	Boucle while tant que le nombre de conflits est gérable. À partir d'une approximation glouton, le nombre de couleurs qui peut être réduit sans conflits varie empiriquement de 1 à 3.
112	$O(V)*O(V^2) = O(V^3)$	Pour chaque sommet dans la coloration, si la couleur est K-1 alors on change la couleur pour celle qui génère le moins de conflits. La vérification du nombre de conflits est de l'ordre

		de O((K-1)*V). Dans le pire cas, le nombre de couleurs utilisées est égal au nombre de sommets, alors La vérification du nombre de conflits serait de l'ordre de $O(V^2)$.
113	$O(nV^4)$	Résultat de la recherche tabou expliqué dans le tableau suivant et au point [2].
115	$O(V^2)$	On parcourt chaque sommet du graphe, et pour chaque sommet on parcourt tous ses voisins pour vérifier s'ils ont la même couleur que le sommet visité. Dans le pire cas, le graphe est complet et chaque sommet est relié à tous les autres sommets.
116	$\Theta(1)$	Comparaison.
117	$\Theta(1)$	On sort de la fonction.
119	$\Theta(V)$	On copie le tableau de coloration qui est de la taille du nombre de sommets.
120	$\Theta(1)$	Retour de la fonction.

Figure #5 : Code de la recherche tabou (partie 1)

```
50  def tabouSearch(graph, coloration):
51      # Tabou search
52      bestColoration = coloration
53      currentColoration = bestColoration
54      tabouList = []
55
56      nbOfIterations = 0
57      while nbOfIterations < 2*len(coloration): # TODO: choix de condition while
58          # Used to facilitate python code, tabou list is under the form ((vertice, color), time)
59          if len(tabouList) > 0:
60              trunkatedTabouList = [x[0] for x in tabouList]
61          else:
62              trunkatedTabouList = []
63
64          # Neighbours generation
65          # i.e generate new colorations from current coloration
66          generatedNeighbours = []
67          for vertice in currentColoration.keys():
68              for color in range(max(currentColoration.values())):
69                  if currentColoration[vertice] != color and (vertice, color) not in trunkatedTabouList:
70                      newColoration = copy.copy(currentColoration)
71                      newColoration[vertice] = color
72                      generatedNeighbours.append((newColoration, (vertice, color)))
73
74          # Find best neighbour (min conflits)
75          bestIndex = 0
76          minNumberOfConflict = float('inf')
77          for idx, neighbour in enumerate(generatedNeighbours):
78              nbOfConflicts = getNumberOfConflict(graph, neighbour[0])
79              if nbOfConflicts < minNumberOfConflict:
80                  bestIndex = idx
81                  minNumberOfConflict = nbOfConflicts
82
```

Figure #5 : Code de la recherche tabou (partie 2)

```
83          bestNeighbour = generatedNeighbours[bestIndex][0]
84          currentColoration = bestNeighbour
85
86          # Add reference couple to tabou list under the form ((vertice, color), time)
87          tupleVerticeColorOfReference = generatedNeighbours[bestIndex][1]
88          tabouList.append([(tupleVerticeColorOfReference), 2*minNumberOfConflict + randrange(1,10)])
89
90          # Evaluate if the solution is better than bestColoration
91          nbOfConflictsOfCurrentColoration = getNumberOfConflict(graph, currentColoration)
92          nbOfConflictsOfBestColoration = getNumberOfConflict(graph, bestColoration)
93          if nbOfConflictsOfCurrentColoration < nbOfConflictsOfBestColoration:
94              bestColoration = currentColoration
95              nbOfIterations = 0
96
97          # Update tabou list
98          for item in tabouList:
99              item[1] -= 1
100         tabouList[:] = [x for x in tabouList if x[1] > 0]
101
102         nbOfIterations += 1
103
104     return bestColoration
```

Tableau #6 : Analyse théorique ligne par ligne de la recherche tabou

# Ligne de l'instruction	Complexité asymptotique	Justification
52	Θ(V)	On copie le tableau de coloration qui est de la taille du nombre de sommets.
53	Θ(V)	On copie le tableau de coloration qui est de la taille du nombre de sommets.
54	Θ(1)	Initialisation.
56	Θ(1)	Initialisation.
57-103	Θ(n)	Dans le pire des cas on fait n itérations, dans ce cas-ci n=2*V. Cette valeur à été trouvée de manière empirique. Elle correspond au nombre qui permet d'atteindre à coup sûr une résolution de conflit sans faire des itérations inutiles.
59	Θ(1)	Comparaison.
60	$O(V*(K-1)) \approx O(V^2)$	On parcourt la liste tabou pour récupérer les couples sans le temps. Voir l'explication [1] à la suite de ce tableau. On suppose ensuite qu'en pire cas on a un nombre de couleur égal au nombre de sommet, et donc cette étape se fait en $O(V^2)$.
62	Θ(1)	Comparaison
66	Θ(1)	Initialisation.
67-72	O(V)	On parcourt la liste de coloration, dans le pire

		cas celle-ci contient tous les sommets du graphe.
68-72	$O(K) \approx O(V)$	On parcourt toutes les couleurs disponibles pour générer une nouvelle couleur au sommet, dans le pire cas on a une couleur pour chaque sommet.
69	$O(V*(K-1)) \approx O(V^2)$	Pour vérifier que l'élément n'est pas dans la liste tabou on doit parcourir celle-ci. Voir l'explication [1] à la suite de ce tableau. On suppose ensuite qu'en pire cas on a un nombre de couleurs égal au nombre de sommets, et donc cette étape se fait en $O(V^2)$.
70	$\Theta(V)$	On copie le tableau de coloration qui est de la taille du nombre de sommets.
71	$\Theta(1)$	Assignation.
72	$\Theta(1)$	Ajout à la liste.
75	$\Theta(1)$	Initialisation.
76	$\Theta(1)$	Initialisation.
77-81	$O(KV) \approx O(V^2)$	On parcourt la liste de tous les voisins générés. On a un voisin pour chaque couleur possible de sommet qui ne sont pas dans la liste tabou. Dans le pire cas, avec une liste tabou vide on a toutes les combinaisons de (sommet, couleur) qui est de l'ordre $O(K*V)$.
78	$O(V^2)$	On parcourt chaque sommet du graphe, et pour chaque sommet on parcourt tous ses voisins pour vérifier s'ils ont la même couleur que le sommet visité. Dans le pire cas, le graphe est complet et chaque sommet est relié à tous les autres sommets.
79	$\Theta(1)$	Comparaison.
80	$\Theta(1)$	Assignation.
81	$\Theta(1)$	Assignation.
83	$\Theta(V)$	On copie le résultat du meilleur voisin qui est de la taille du nombre de sommets.

84	$\Theta(V)$	On copie le meilleur voisin pour devenir la nouvelle coloration qui est de la taille du nombre de sommets.
87	$\Theta(1)$	On copie un couple de liste tabou qui est de taille 3. C'est donc en temps constant par une constante multiplicative.
88	$\Theta(1)$	Ajout d'un élément au tableau.
91	$O(V^2)$	On parcourt chaque sommet du graphe, et pour chaque sommet on parcourt tous ses voisins pour vérifier s'ils ont la même couleur que le sommet visité. Dans le pire cas, le graphe est complet et chaque sommet est relié à tous les autres sommets.
92	$O(V^2)$	On parcourt chaque sommet du graphe, et pour chaque sommet on parcourt tous ses voisins pour vérifier s'ils ont la même couleur que le sommet visité. Dans le pire cas, le graphe est complet et chaque sommet est relié à tous les autres sommets.
93	$\Theta(1)$	Comparaison.
94	$\Theta(V)$	On copie le tableau de coloration qui est de la taille du nombre de sommets.
95	$\Theta(1)$	Assignation.
98-99	$O(V*(K-1)) \approx O(V^2)$	Voir l'explication [1] à la suite de ce tableau.
99	$\Theta(1)$	Décrémentation de variable.
100	$O(V*(K-1)) \approx O(V^2)$	On parcourt la liste tabou pour récupérer les couples. Voir l'explication [1] à la suite de ce tableau. On suppose ensuite que en pire cas on a un nombre de couleur égal au nombre de sommet, et donc cette étape se fait en $O(V^2)$.
102	$\Theta(1)$	On incrémente la variable "nbOfIterations".
104	$\Theta(1)$	Retour de la fonction.

[1] Taille de la liste tabou :

La liste tabou contient des couples (sommet, couleur) pour une certaine durée. Elle contient donc au minimum V couples, soit un pour chaque sommet après la première itération de la recherche tabou qui aura mis chaque sommet avec la coloration courante dans la liste tabou. Dans le pire des cas, le temps minimal associé à un sommet dans la liste tabou est plus grand que le nombre d'itérations pour passer à travers tous les sommets et toutes les colorations, alors on se retrouve avec la liste tabou qui contient tous les couples possibles de (sommet, couleur).

[2]Complexité de la recherche tabou :

Avant la boucle while, on fait seulement des assignations ou des copies de tableau en complexité $\Theta(V)$.

Dans la boucle while on a trois boucles for. La première boucle for a une autre boucle for imbriquée, on utilisera donc la formule $h1*h2 \in \Theta(f*g)$ pour déterminer la complexité de cette section de l'algorithme. Pour la boucle for imbriquée (ligne 68) la plus grande opération est lorsqu'on vérifie que l'élément n'est pas dans la liste tabou et ceci se fait en $O(V^2)$. On fait cette vérification V fois en pire cas, alors la complexité des lignes 68-72 est en $O(V^3)$. Ensuite on fait toute cette boucle V fois encore une fois pour la boucle de la ligne 67. On a donc une complexité pour les boucles for imbriquées en $O(V^4)$.

La deuxième boucle for dans la boucle while, à la ligne 77, itère sur tous les voisins générés, ce qui est en $O(V^2)$. Pour chaque voisin, elle doit ensuite calculer le nombre de conflits qui se fait aussi en $O(V^2)$. En utilisant la formule $h1*h2 \in \Theta(f*g)$ on obtient que les lignes 77-81 sont en $O(V^4)$.

La dernière boucle for dans la boucle while, à la ligne 98, parcourt la liste tabou, ce qui se fait en $O(V^2)$.

La plus grande opération de la boucle while est donc la première ou la deuxième boucle for en $O(V^4)$. On exécute cette boucle n fois, la recherche tabou se fait donc au total en $O(nV^4)$ en pire cas.

Complexité pour l'algorithme tabou au complet :

Pour calculer la complexité de l'algorithme tabou, on va appliquer la formule du maximum pour l'intérieur de la boucle while, puis la formule $h1*h2 \in \Theta(f*g)$ pour la boucle while en entier. La boucle while est de l'ordre $O(1)$ comme expliqué dans le tableau 5 et l'opération ayant la plus grande complexité à l'intérieur de la boucle while est la recherche tabou qui est en $O(nV^4)$, on obtient donc une complexité pour l'algorithme tabou en $O(nV^4)$.

Servez-vous de vos temps d'exécution pour confirmer et/ou préciser l'analyse asymptotique théorique de vos algorithmes avec la méthode hybride de votre choix.

Pour l'analyse hybride, un test de puissance a tout d'abord été réalisé sur chacun des algorithmes. En utilisant une échelle log-log on peut déduire la puissance de l'algorithme si celui-ci croît de façon polynomiale. Avec l'équation (1) on peut isoler la puissance m et ainsi trouver la consommation expérimentale.

$$log_a y = m \cdot log_a x + b \text{ (1)}$$

Par la suite, nous avons utilisé le test du rapport pour confirmer ou infirmer les résultats trouvés avec les tests de puissance et pour trouver la constante multiplicative. Avec l'équation (2), si le test du rapport converge vers une constante b > 0, alors on peut conclure :

$$y = b \cdot f(x) \text{ (2)}$$

Puis pour terminer, nous avons effectué un test des constantes pour trouver le coût fixe de chacun des algorithmes avec l'équation (3).

$$y = c \cdot f(x) + b \text{ (3)}$$

Analyse algorithme glouton

Pour l'algorithme glouton, le test de puissance nous révèle une droite d'équation y = 1.5755x - 1.0662. On peut déduire que la complexité est polynomiale et que la puissance m trouvée est m=1.5755 avec l'équation (1). Toutefois, nous estimons que pour avoir des résultats plus proches de la réalité il aurait fallu exécuter les trois algorithmes sur des tailles de graphes plus grands que 50. Nos ordinateurs personnels ne sont pas en mesure de réaliser cela puisque pour l'algorithme branch and bound, résoudre un graphe de taille 55 et plus prenait plusieurs heures. Nous voulions donc rester consistants pour tous les algorithmes et nous avons basé notre échantillon sur des graphes de taille 20 à 50 par incrémentation de 5. La valeur m expérimentale est différente de celle que nous avons trouvé par analyse théorique, mais cela peut s'expliquer par les raisons énoncées juste avant, et par le fait que l'analyse théorique a été réalisée entièrement pour le pire cas.

Avec le test du rapport, on peut voir que le graphique semble converger vers une valeur proche de 0.1, qui serait la constante multiplicative b. Cela permet aussi de valider que l'algorithme croît bel et bien de façon polynomiale.

Finalement, avec le test des constantes, on peut confirmer une fois de plus que l'algorithme croît selon f(x)=x^1.5755. On peut également trouver le coût fixe à l'aide de l'équation (3) qui est pour l'algorithme glouton de 0.1291.

Analyse algorithme branch and bound

Pour l'algorithme branch and bound, le test de puissance nous révèle une droite d'équation y=10.34x - 13.04. On peut déduire que la complexité est polynomiale, la puissance m trouvée est m=10.34. Ceci est considérablement différent de la complexité théorique trouvée de $O(V!*V^2)$. Encore une fois, nous croyons qu'avec un plus grand échantillon nous aurions trouvé de meilleurs résultats. De plus, notre analyse a été faite en pire cas. Ainsi, il est normal de constater une différence entre les résultats trouvés empiriquement et les résultats théoriques. Si la taille des exemplaires n'avait pas été limitée entre 20 et 50, la valeur de 'm' déduite de la régression linéaire aurait pu être largement supérieure à 10. Notre analyse théorique fait intervenir V!. Ainsi, la taille du 'm' varie selon la taille de l'exemplaire ce qui rend l'analyse de complexité difficile.

Le test du rapport de l'algorithme branch and bound ne montre pas de convergence claire. Cependant, l'échelle du graphique selon y est extrêmement petite (ordre de 10^{-13}). Ceci laisse présager qu'une erreur a pu être introduite dans notre analyse. L'allure du graphique montre qu'il s'agit d'une sous-estimation.

Le test des constantes ne semble pas corroborer nos résultats. En fixant les valeurs estimées sur l'axe des abscisses et les valeurs trouvées sur l'ordonnée, on constate que les courbes apparaissant sur les graphiques sont linéaires jusqu'à la valeur x = 50. Cela confirme que l'approximation faite par $f(x)$ pour de petites tailles est correcte, mais n'est pas adéquate en général.

Analyse algorithme de recherche tabou

Pour l'algorithme de recherche tabou, le test de puissance nous révèle une droite d'équation y = 5.56x - 5.1035. On peut déduire que la complexité est polynomiale, et que la puissance m trouvée est m = 5.56 à partir de l'équation (1). La complexité expérimentale trouvée se rapproche de celle trouvée théoriquement, soit $O(nV^4)$. De plus, nous avons utilisé une valeur de n=2*V, alors on peut supposer que la complexité de notre algorithme de recherche tabou est en fait en $O(V^5)$ tout comme le confirment nos résultats expérimentaux.

Le test du rapport peut également confirmer les résultats trouvés avec le test de puissance puisque les données semblent converger vers une constante b=000008. Cette valeur serait donc la constante multiplicative pour l'algorithme tabou.

Finalement, le test des constantes nous révèle que le temps d'exécution croît bel et bien de la même façon que la fonction $F(x) = x^{5.56}$ puisque nous obtenons une droite d'équation $y = 7x10^{-6}x + 469.29$. Avec l'équation (3) on peut déduire le coût fixe de l'algorithme qui correspond à la valeur $7x10^{-6}$ ce qui est assez négligeable comme coût. En général, l'analyse expérimentale semble valider les hypothèses posées pour l'algorithme de recherche tabou.

Discutez des trois algorithmes en fonction de la qualité respective des solutions obtenues, de la consommation de ressources (temps de calcul, espace mémoire) et de la difficulté d'implantation.

En ce qui concerne la qualité respective des solutions obtenues, lorsque le nombre de sommets est relativement petit, les résultats sont semblables pour les 3 algorithmes. Toutefois, lorsque le nombre de sommets augmente (par exemple pour des graphes de 25 sommets et plus) l'algorithme branch and bound devient plus optimal que l'algorithme glouton. Toutefois, celui-ci prend beaucoup plus de temps d'exécution que les deux autres algorithmes pour des tailles de graphes plus grandes. Par exemple, pour des graphes ayant 50 sommets, l'algorithme branch and bound colore le graphe avec 1.4 couleurs de moins en moyenne, mais prend environ 4000 fois plus de temps à résoudre que l'algorithme glouton. On peut donc se questionner sur l'avantage de prendre cet algorithme.

En ce qui concerne la difficulté d'implantation, l'algorithme glouton est facile à implémenter, mais ne garantit jamais l'optimalité. L'algorithme branch and bound est un peu plus difficile à implémenter que le glouton mais reste tout de même pas trop difficile à implémenter. En plus, il s'agit d'un algorithme exact puisqu'il peut explorer toutes les possibilités et retourner la solution optimale, mais au détriment du temps d'exécution. L'algorithme de recherche tabou est par contre très difficile à implémenter et il s'agit d'un algorithme heuristique à base de trajectoire. On n'est pas garanti de trouver l'optimum global mais on peut facilement trouver un optimum local qui est assez bon. De plus, cet algorithme est rapide à exécuter, semblable à l'algorithme glouton en temps d'exécution. Il peut donc être une belle alternative si on surmonte la difficulté d'implantation.

Indiquez sous quelles conditions vous utiliseriez chaque algorithme.

L'algorithme glouton est préférable pour de petites tailles de graphes puisqu'il est rapide à implémenter et à résoudre, mais ne garantit pas la solution optimale. Toutefois, pour de petites tailles de graphe la solution optimale est souvent celle retournée par le glouton alors on ne gagne pas en optimalité de prendre un des autres algorithmes. On peut également utiliser le glouton pour de grande taille de graphe dans le cas où on ne veut pas de solution optimale mais une solution rapide.

L'algorithme branch and bound est préférable pour avoir une garantie d'optimalité. Puisque toutes les solutions potentiellement optimales sont considérées par l'algorithme, une coloration minimale est nécessairement générée. Cependant, pour des tailles de matrices 50 ou plus, le temps est considérablement plus long. Selon le contexte, il pourrait être préférable de choisir un algorithme plus rapide, ou encore de patienter si l'enjeu en vaut la peine!

L'algorithme de recherche tabou est préférable pour de grande taille de graphe lorsqu'on veut s'approcher de la solution optimale et avoir un temps d'exécution relativement rapide. Comme expliqué à la section précédente, l'algorithme de recherche tabou est un algorithme heuristique à base de trajectoire. Il ne garantit pas l'optimalité mais peut s'en approcher tout en ayant un temps d'exécution rapide. Il est donc préférable à l'algorithme glouton pour de grandes tailles, mais peut être moins optimal que branch and bound, tout dépend de notre objectif et du temps à notre disposition.

Autres critères de correction

Respect de l'interface tp.sh

0	/ 1 pt

Utilisation

tp.sh -a [glouton | branch_bound | tabou] -e [path_vers_exemplaire] [-p] [-t]

Arguments optionnels

-p Affiche la solution sur deux lignes : la première contient le nombre de couleur utilisée et la deuxième contient la couleur de chaque sommet, sur une seule ligne. L'ordre des couleur doit correspondre à l'ordre des sommets dans la matrice d'adjacence. La première couleur doit donc être celle du sommet de la première ligne de la matrice, et ainsi de suite.

-t affiche le temps d'exécution en ms, sans unité ni texte superflu

Important: l'option -e doit accepter des fichiers avec des paths absolus.
Par exemple pour un exemplaire de taille 30, la première ligne donne :

```
./tp.sh -a glouton -e ex30_0 -p -t
9
4 7 0 1 3 2 0 5 3 1 5 0 2 2 3 1 2 0 7 4 1 4 5 4 3 6 6 5 8 1
0.683349609375
```

On y trouve d'abord le nombre de couleurs utilisées dans la solution. Puis les couleurs de chaque nœud sur une seule ligne. Et enfin le temps en ms.

Qualité du code

0	/ 1 pt

Présentation générale

0	/ 2 pt

Concision

Qualité du français

Pénalité retard

0

-1 pt / journée de retard, arrondi vers le haut. Les TPs ne sont plus acceptés après 3 jours.