UNIVERSITE DES SCIENCES ET DE LA TECHNOLOGIE USTHB
DEPARTEMENT D'INFORMATIQUE

Devoir complexité BioInfo

Implémentation et analyse de complexité d'algorithmes sur les graphes

Présenté par :
GUELFEN Abdelheq

Année universitaire 2023-2024

Contents

0.1 Introduction

Les graphes, essentiels pour modéliser des situations réelles, trouvent leur application dans divers domaines tels que les réseaux routiers, les systèmes informatiques, ou encore la planification de projets. Dans ce projet, nous nous concentrons sur la manière de stocker ces graphes dans un ordinateur et sur les opérations courantes associées. Deux méthodes principales, la matrice d'adjacence et la liste d'adjacence, seront explorées pour comprendre comment elles influent sur la complexité, c'est-à-dire l'espace mémoire requis et le temps d'exécution des opérations. Cette exploration vise à fournir des insights pratiques pour utiliser efficacement les graphes dans la résolution de problèmes concrets.

0.2 Objectifs de travail

Rappels sur la représentation de graphes en mémoire (différentes représentations) Opérations sur les graphes avec calcul de complexité Evaluation et comparaison de complexités théoriques et expérimentales

0.3 Partie I

On a établir les principales définition de la structure de graphe

- **Graphe**: est un ensemble de relations entre un ou plusieurs objets.
- **Graphe orienté**: est un graphe dans lequel chaque sommet a une direction spécifique.
- **Graphe non orienté**: est un graphe dans lequel les sommet n'ont pas de direction spécifique.
- **arête**: est une relation entre deux sommets dans un graphe non orienté.
- **arc**: est une relation entre deux nœuds dans un graphe orienté.
- **cycle**: est une séquence d'arêtes qui forme une boucle fermée dans un graphe.
- **circuit**: est un cycle qui parcourt tous les sommets d'un graphe eulérien.
- **graphe pondéré**: est un graphe dans lequel chaque arête a une valeur associée, généralement numérique, appelée poids.
- **degré d'un nœud**: est le nombre d'arêtes incidentes à ce nœud.
- **arc entrant**: est une arête qui se termine au nœud sans en sortir.
- **arc sortant**: est une arête qui part du nœud sans y entrer.

- **Graphe simple**: est un graphe sans boucles (arêtes reliant un nœud à lui-même) ni arêtes multiples entre les mêmes paires de nœuds.
- **multi-graphe**: Un multi-graphe permet d'avoir plusieurs arêtes entre les mêmes paires de nœuds.
- **graphe eulérien**: est un graphe qui contient un circuit eulérien, passant par chaque arête exactement une fois.
- **arbre**: Un arbre est un graphe orienté sans cycle, et muni d'une racine.
- **Densité d'un graphe**: La densité d'un graphe est le rapport entre le nombre d'arêtes présentes et le nombre d'arêtes potentielles dans un graphe complet.
- **graphe connexe**: Un graphe connexe est un graphe dans lequel il existe un chemin entre chaque paire de nœuds.
- **composante connexe**: Une composante connexe est un sous-graphe maximal dans un graphe non orienté connexe.
- **profondeur**: La profondeur d'un nœud dans un graphe est la longueur du plus court chemin depuis un nœud racine jusqu'à ce nœud
- **graphe complet**: Un graphe complet est un type particulier de graphe non orienté dans lequel il existe une arête entre chaque paire distincte de nœuds.

0.4 Partie II

Les représentations d'un graphe en mémoire

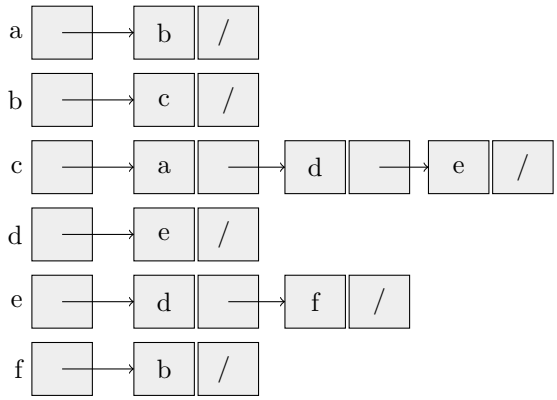
- **La definition de matrice d'adjacence:** La matrice d'adjacence est une représentation tabulaire d'un graphe où chaque élément M[i][j] indique s'il existe une arête entre les sommets i et j.La matrice utlise utilise (0 ou 1) pour représenter l'absence ou la présence d'une arête.
- **La definition de liste d'adjacence:** La liste d'adjacence est une représentation basée sur des tableaux ou des listes où chaque sommet du graphe est associé à une liste de ses voisins (les sommets auxquels il est directement connecté).

0.4.1 Graphe 1

La matrice d'adjacence

	a	b	c	d	e	f
a	0	1	0	0	0	0
b	0	0	1	0	0	0
c	1	0	0	1	1	0
d	0	0	0	0	1	0
e	0	0	0	1	0	1
f	0	1	0	0	0	0

La liste d'adjacence

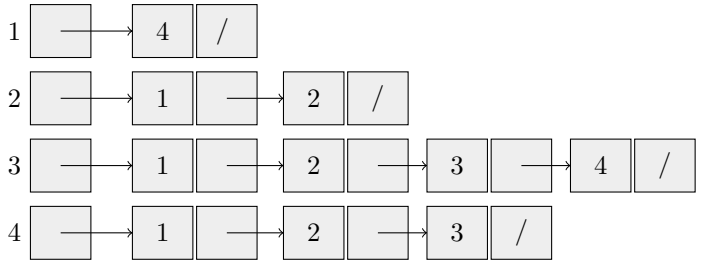


0.4.2 Graphe 2

La matrice d'adjacence

	1	2	3	4
1	0	0	0	1
2	1	1	0	0
3	1	1	1	1
4	1	1	1	0

La liste d'adjacence

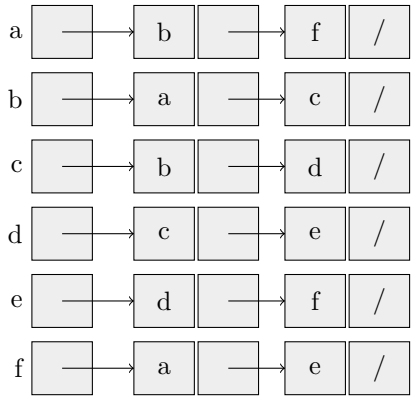


0.4.3 Graphe 3

La matrice d'adjacence

	a	b	c	d	e	f
a	0	1	0	0	0	1
b	1	0	1	0	0	0
c	0	1	0	1	0	0
d	0	0	1	0	1	0
e	0	0	0	1	0	1
f	1	0	0	0	1	0

La liste d'adjacence



0.5 Partie III

0.5.1 Construction d'un graphe

utilisant la matrice d'adjacence

La complexite de cet algorithme est:$O(n^2)$
on a deux boucle qui repete N fois

Algorithm 1:

```
   Data: M: matrice/entier
   Data: N: taille de graph/entier
1  for i de 0 a N do
2  |   for j de 0 a N do
3  |   |   M[i][j] = 0
4  for arc in arces do
5  |   M[arc[0]][arc[1]] = 1
```

utilisant la liste d'adjacence

La complexite de cet algorithme est:$O(n)$
on a un seul boucle qui repete N fois

Algorithm 2:

```
   Data: T: tablaux/entier
   Data: N: taille de graph/entier
1  for i de 0 a N do
2  |   T[i] = 0
3  for arc in arces do
4  |   M[arc[0]] = arc[1]
```

0.5.2 Affichage de graphe

utilisant la matrice d'adjacence

La complexite de cet algorithme est:$O(n^2)$
on a deux boucle qui repete N fois

Algorithm 3:

```
   Data: M: matrice/entier
1  for i de 0 a N do
2  |  for j de 0 a N do
3  |  |  printf(M[i][j])
```

utilisant la liste d'adjacence

La complexite de cet algorithme est:$O(n * m)$
on a deux boucle qui repete N et M fois respectivement

Algorithm 4:

```
   Data: T: tablaux/entier
   Data: M: taille de tablaux/entier
1  for i de 0 a N do
2  |  printf(i)
3  |  for j de 0 a M do
4  |  |  printf(T[i][j])
```

0.5.3 Calcule de densité

utilisant la matrice d'adjacence

Algorithm 5:

```
   Data: M: matrice/entier
1  m = 0
2  n = 0
3  for i de 0 a N do
4  |  for j de 0 a N do
5  |  |  if M[i][j] == 1 then
6  |  |  |  m = m + 1
7  return (2 * m) / (n * (n - 1))
```

La complexite de cet algorithme est:$O(n^2)$on a deux boucle qui repete N fois

utilisant la liste d'adjacence

La complexite de cet algorithme est:$O(n * m)$
on a deux boucle qui repete N et M fois respectivement

Algorithm 6:

```
   Data: M: matrice/entier
1  m = 0
2  n = 0
3  for i de 0 a N do
4  |  n = n + 1
5  |  for j de 0 a M do
6  |  |  m = m + 1
7  return (2 * m) / (n * (n - 1))
```

0.5.4 Calcule de degre

utilisant la matrice d'adjacence

La complexite de cet algorithme est:$O(n^2)$
on a deux boucle qui repete N fois

Algorithm 7:

```
   Data: M: matrice/entier
1  m = 0
2  n = 0
3  for i de 0 a N do
4  |  for j de 0 a N do
5  |  |  if M[i][j] == 1 then
6  |  |  |  m = m + 1
7  return m
```

utilisant la liste d'adjacence

La complexite de cet algorithme est:$O(n)$
on un seul boucle qui repete N fois

Algorithm 8:

```
   Data: M: matrice/entier
1  m = 0
2  for i de 0 a N do
3  |  m = m + 1
4  return m
```

0.5.5 Verification de graphe complet

utilisant la matrice d'adjacence

La complexite de cet algorithme est:$O(n^2)$
on a deux boucle qui repete N fois

Algorithm 9:

```
   Data: M: matrice/entier
1  m = 0
2  n = 0
3  for i de 0 a N do
4  |  for j de 0 a N do
5  |  |  if M[i][j] == 1 then
6  |  |  |  m = m + 1
7  return m == n * (n - 1) / 2
```

utilisant la liste d'adjacence

La complexite de cet algorithme est:$O(n * m)$
on a deux boucle qui repete N et M fois respectivement

Algorithm 10:

```
   Data: M: matrice/entier
1  m = 0
2  n = 0
3  for i de 0 a N do
4  |  n = n + 1
5  |  for j de 0 a M do
6  |  |  m = m + 1
7  return m == n * (n - 1) / 2
```

0.5.6 Recherche d'un noeud dans le graphe

utilisant la matrice d'adjacence

La complexite de cet algorithme est:$O(n)$
la complexite est paraport la taille de noeud

Algorithm 11:

```
   Data: M: matrice/entier
1  T = []
2  for i de 0 a M[node] do
3  |  if M[node][i] == 1 then
4  |  |  T[I] = 1
5  return T
```

utilisant la liste d'adjacence

La complexite de cet algorithme est:$O(1)$
c'est un algorithme constante

Algorithm 12:

```
   Data: G: graphe/entier
1  return G[node]
```

0.5.7 Recherche de tous les chemins

La complexite de cet algorithme est:$O(n^2)$
On a un boucle avec une appelle recursive

Algorithm 13:

```
   Data: M: matrice/entier
1  file = File()
2  visited = []
3  enfilier(file, node)
4  visited[0] = node
5  while non vide(file) do
6  |  currentnode = defilier(file)
7  |  for neighbor in fils(currentnode) do
8  |  |  if neighbor not in visited then
9  |  |  |  dfs(neighbor)
```

0.5.8 Recherche le chemin le plus court

La complexite de cet algorithme est le meme de complexite de $dfs()$

Algorithm 14:

Data: M: matrice/entier
paths = dfs()
return min(paths)

0.6 Evaluation expérimentale

On a effectué plusieurs tests avec des graphe de taille 10, 20, 50, et on a trouver ces resultats

Taille n	Construction(Tps)	Affichage(Tps)	Densité(Tps)	Dergee(Tps)
10	2.3e-05	2.36e-05	1.3e-05	9.05e-06
20	2.6e-05	2.6e-05	1.4e-05	5.7e-06
50	4.5e-05	4.5e-05	1.2e-05	6.9e-06

Taille n	Graphe est complet(Tps)	Recherche(Tps)	Tous chemins(Tps)	petite chemin
10	1.7e-05	4.5e-06	6.6e-05	1.9e-05
20	1.4e-05	4.5e-06	4.05e-05	4.05e-05
50	1.4e-05	4.7e-06	0.002	0.002

0.7 Conclusion

En résumé :
Complexité spatiale: Matrice d'adjacence :$O(n^2)$ inefficace pour les grands graphes. Liste d'adjacence : Espace proportionnel aux arêtes et sommets, plus efficace pour les graphes épars.

Complexité temporelle: Matrice d'adjacence : Accès constant $O(1)$, itération $O(n^2)$. Liste d'adjacence : Performances temporelles généralement meilleures pour les graphes épars.

Taille et densité: Matrice d'adjacence: Plus grande pour les graphes denses. Liste d'adjacence : Plus performante pour les graphes épars.

En conclusion, la matrice d'adjacence offre un accès constant, mais son espace peut être inefficace pour les graphes épars. La liste d'adjacence est plus efficace pour ces graphes, offrant une meilleure performance pour de nombreuses opérations. Le choix dépend du cas d'utilisation, de la densité du graphe et des opérations envisagées.

0.8 Annexe

Quelque example d'afichage de graphes apres executer le source code

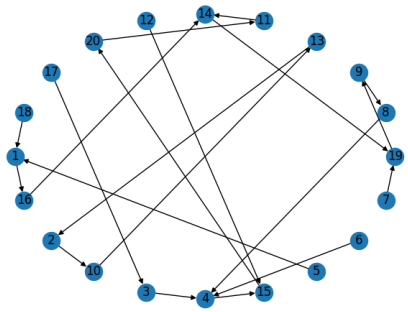


Figure 1: Une affichage de graphe de taille 10

```
1: Construction d'un graphe orienté (/non orienté)
2: Affichage de graphe
3: Calculer la densité de graphe
4: Calculer le degree de graphe
5: Verifie si le graphe est complet
6: Recherche d'un nœud a dans le graphe (afficher le nœud et ses liens)
7: Trouver tous les chemins entre 2 points
8: Trouver le plus court chemins entre 2 points

> 1
Execution time: 2.9087066650390625e-05 seconds
> 3
Execution time: 1.3828277587890625e-05 seconds
La densité est 0.10526315789473684
> 7
entre le noeud de depart et fin: 9 16
Execution time: 4.5299530029296875e-05 seconds
tous les chemins entre 9 et 16 sont: []
> 7
entre le noeud de depart et fin: 8 9
Execution time: 3.314018249511719e-05 seconds
tous les chemins entre 8 et 9 sont: [[8, 4, 15, 20, 11, 14, 19, 9]]
>
```

Figure 2: Capture d'ecran illustrant l'execution de quelque fonctions