Université Blida 1
Faculté des Sciences
Département d'Informatique
Master IL (Ingénierie Logiciel)
Semestre 1

TP4 – ALGORITHMES CLASSIQUES SUR LES GRAPHES

Module : Algorithme avoncé
Professeur : 1 Mme AROUSSI
(s_aroussi@esi.dz)
Année : 2025 - 2026

GROUPE 8

HADJ MOHAMMED DOUAE

ESCHROUGUI YOUSRA

FARES YASMINE

LEZOUL IMENE

SEKKAL WASSILA

PLAN :

01 Environnement utilisé

02 Les bibliothèque:Définition et Rôle

03 Structures de données et Opérations

04 Mode d’emploi

05 Résultats

06 Analyse critique et Conclusion

01 ENVIRONNEMENT UTILISÉ

-**Processeur** : Intel(R)core(TM)i7-5600U CPU @ 2.60GHZ

-**RAM**: 4.00 GB

-**Carte Graphique** : Intel(R) HD Graphics 5500

Environnement logiciel :

-**Os utilisé** : Windows 10 PRO

-**Language** :Python “3.13.9”

-**Environnement de dévlopement** : Visual Studio Code

02

LES BIBLIOTHÈQUE

sys

Permet d’interagir avec le système (fermer le programme, lire les arguments, etc.).

math

Contient des fonctions mathématiques (cos, sin, sqrt...).

random

Génère des valeurs aléatoires (choix d’un nombre, tirage aléatoire...).

collections.defaultdict

Un dictionnaire spécial qui crée automatiquement une valeur par défaut pour éviter les erreurs.

02 LES BIBLIOTHÈQUE

heapq

Gère une file de priorité (priority queue), très importante dans l'algorithme de Dijkstra.

PyQt5.QtWidgets

Contient tous les éléments graphiques :

- **Fenêtres (QMainWindow)**
- **Boutons (QPushButton)**
- **Tableaux (QTableWidget)**
- **Zones de texte (QLabel, QTextEdit)**
- **Onglets (QTabWidget)**
- **Zone graphique (QGraphicsView, QGraphicsScene)**

PyQt5.QtCore

Contient :

- **Les constantes Qt (alignement, curseurs…)**
- **Les minuteries (QTimer)**
- **Les points et positions (QPointF)**
- **La gestion des événements**

PyQt5.QtGui

Tout ce qui touche au dessin et aux couleurs :

- **Les pinceaux (QBrush)**
- **Les stylos (QPen)**
- **Les couleurs (QColor)**
- **Les polices (QFont)**
- **Le peintre (QPainter)**

I

PARTIE 1 :

- APPROCHE GLOUTONNE
- PCS.Le problème de coloration des sommets
-Algorithme de DSATUR :
  1. Définition d’algorithme DSATUR
  2.Schéma glouton
  3.entrée,sortie,objectif
  4.le principe de l’algoritme dsatur
  5.exmeple de déroulement
  6.Complexité de l’algorithme DSATUR
  7.la structure de données ,Pseudo code

03 STRUCTURES DE DONNÉES ET OPÉRATIONS

01 APPROCHE GLOUTONNE :

Définition de l’approche gloutonne :

L’approche gloutonne est une méthode d’optimisation qui construit une solution étape par étape en faisant, à chaque étape, le choix localement optimal,c’est-à-dire le meilleur choix immédiat possible.

Elle ne revient jamais en arrière, car elle suppose que ces choix locaux conduiront à une solution globalement satisfaisante, voire optimale.

Caractéristiques principales :

- Simple à implémenter
- Très rapide (efficacité temporelle élevée)
- Pas toujours optimale
- Utilisée dans plusieurs algorithmes classiques

Où utilise-t-on l’approche gloutonne ?

- Coloration des graphes
- Arbres couvrants
- Ordonnancement
- Compression et codage

02

PCS.Le problème de coloration des sommets :

Définition du problème de coloration :

- Soit G = (X,E) un graphe non orienté simple tel que:
  - X est l'ensemble de sommet avec |X| = n.
  - E est l'ensemble des arêtes avec |E| = m

Le problème de coloration d'un graphe consiste à attribuer une couleur à chaque sommet de manière que deux sommets adjacents n'aient jamais la même couleur, tout en minimisant le nombre total de couleurs utilisées.

Objectif :

Trouver le nombre chromatique $\chi(G)$, c'est-à-dire le plus petit nombre de couleurs permettant de colorer le graphe correctement.

Complexité :

Ce problème est NP-difficile, donc on utilise souvent des heuristiques gloutonnes pour obtenir des solutions proches de l'optimal.

🔹 Pourquoi la coloration est importante ?

La problématique de la coloration apparaît dans de nombreuses applications pratiques, telles que :

Allocation des fréquences radio

Planification d’examens / emplois du temps

Coloration de cartes géographiques

Optimisation des ressources

Planification des tâches

🔹 Les algorithmes utilisés dans la coloration :

- Algorithme glouton simple (Greedy Coloring)
- Welsh & Powell (1967)
- Largest First (LF)
- DSATUR (1979) – Le plus puissant parmi les heuristiques gloutonnes

03

Algorithme de DSATUR :

Définition d’algorithme DSATUR :

- L’algorithme DSATUR (Degree of Saturation)est une amélioration de l’algorithme glouton de Welsh et Powell.
- IL est une méthode gloutonne de coloration des sommets d’un graphe.
- Il repose sur le concept de degré de saturation, défini comme :

> le nombre de couleurs distinctes déjà utilisées dans le voisinage d’un sommet.

- L’idée principale est de colorer en priorité les sommets les plus « contraints », c’est-à-dire ceux dont les voisins possèdent déjà plusieurs couleurs différentes. Ainsi, DSATUR choisit toujours le sommet ayant le degré de saturation le plus élevé, ce qui permet d’obtenir un coloriage souvent proche du coloriage optimal.

ENTRÉE,SORTIE,OBJECTIF

Entrée :

- G = (X, E), un graphe simple et non orienté
- X est l'ensemble de sommet avec |X| = n
- E est l'ensemble des arêtes avec |E| = m

Sortie :

- Ensemble des couleurs utilisées et l'affectation couleur → sommet.

Objectif :

- Colorier les sommets du graphe en utilisant le plus petit nombre possible de couleurs, ou au moins un nombre raisonnablement faible selon une approche gloutonne.

LE PRINCIPE DE L'ALGORITME DSATUR

L'algorithme DSATUR (Degree of Saturation) repose sur une idée centrale :
À chaque étape, on colore le sommet non colorié le plus "contraint", c'est-à-dire celui dont les voisins utilisent déjà le plus grand nombre de couleurs différentes.
Ce nombre s'appelle le degré de saturation (DSAT).

◆ Etapes principales :

1. Initialiser

- **Tous les sommets sont non coloriés.**
- **Pour chaque sommet, on fixe DSAT(x) = 0.**
- **On calcule également le degré classique de chaque sommet (nombre de voisins).**

2. Tant qu'il reste des sommets non coloriés :

a. Sélectionner le sommet ayant :

- le degré de saturation maximal ;

-en cas d'égalité → celui ayant le degré maximal.

b. Déterminer la plus petite couleur disponible,

- c'est-à-dire une couleur qui n'apparaît chez aucun voisin déjà colorié.

c. Affecter cette couleur au sommet sélectionné.

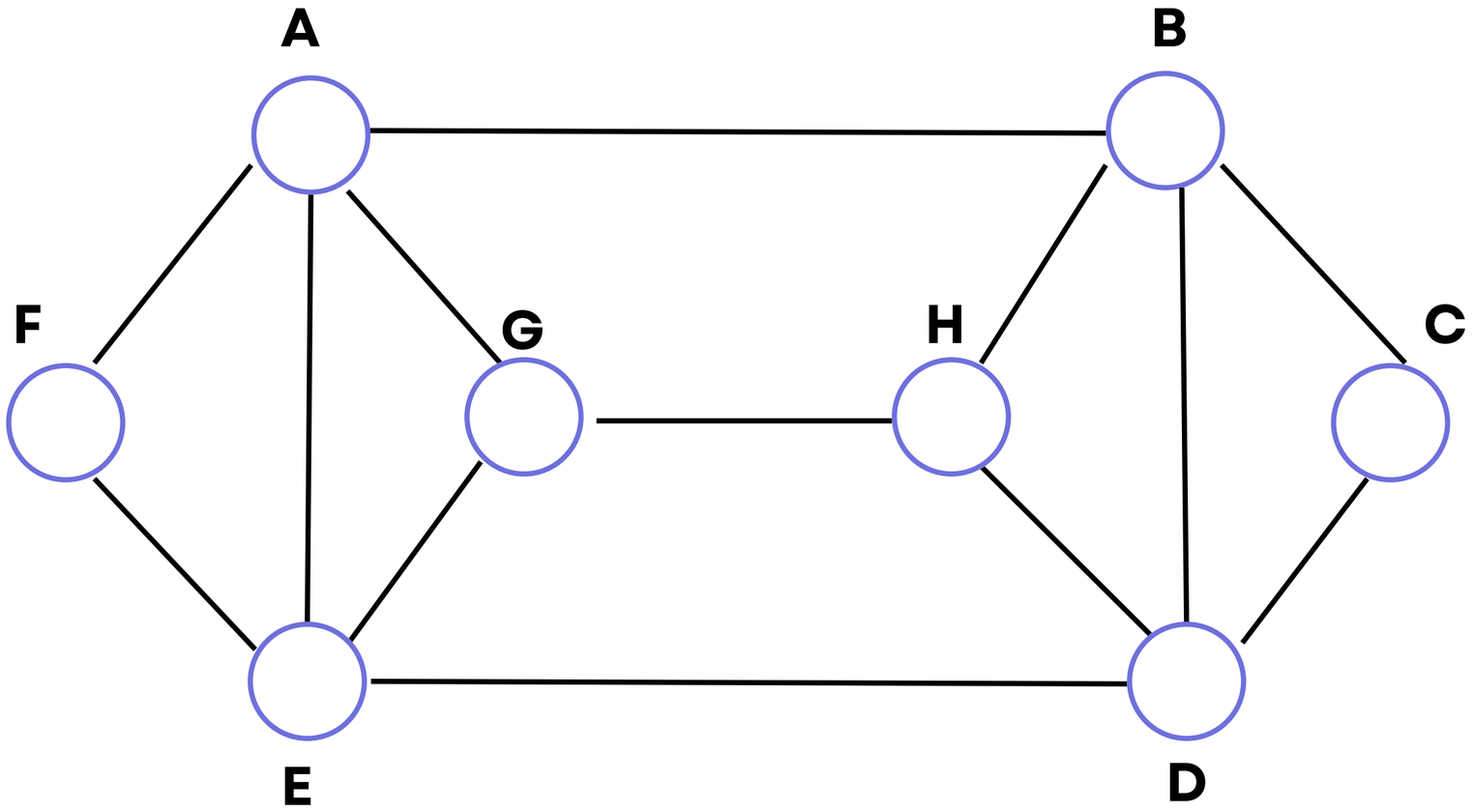
d. **Mettre à jour la saturation de tous les voisins non coloriés :**

- Si le voisin n’avait pas encore vu cette couleur dans son voisinage, alors DSAT(voisin) = DSAT(voisin) + 1.

3. **Répéter jusqu’à coloration de tous les sommets.**

EXMEPLE DE DÉROULEMENT

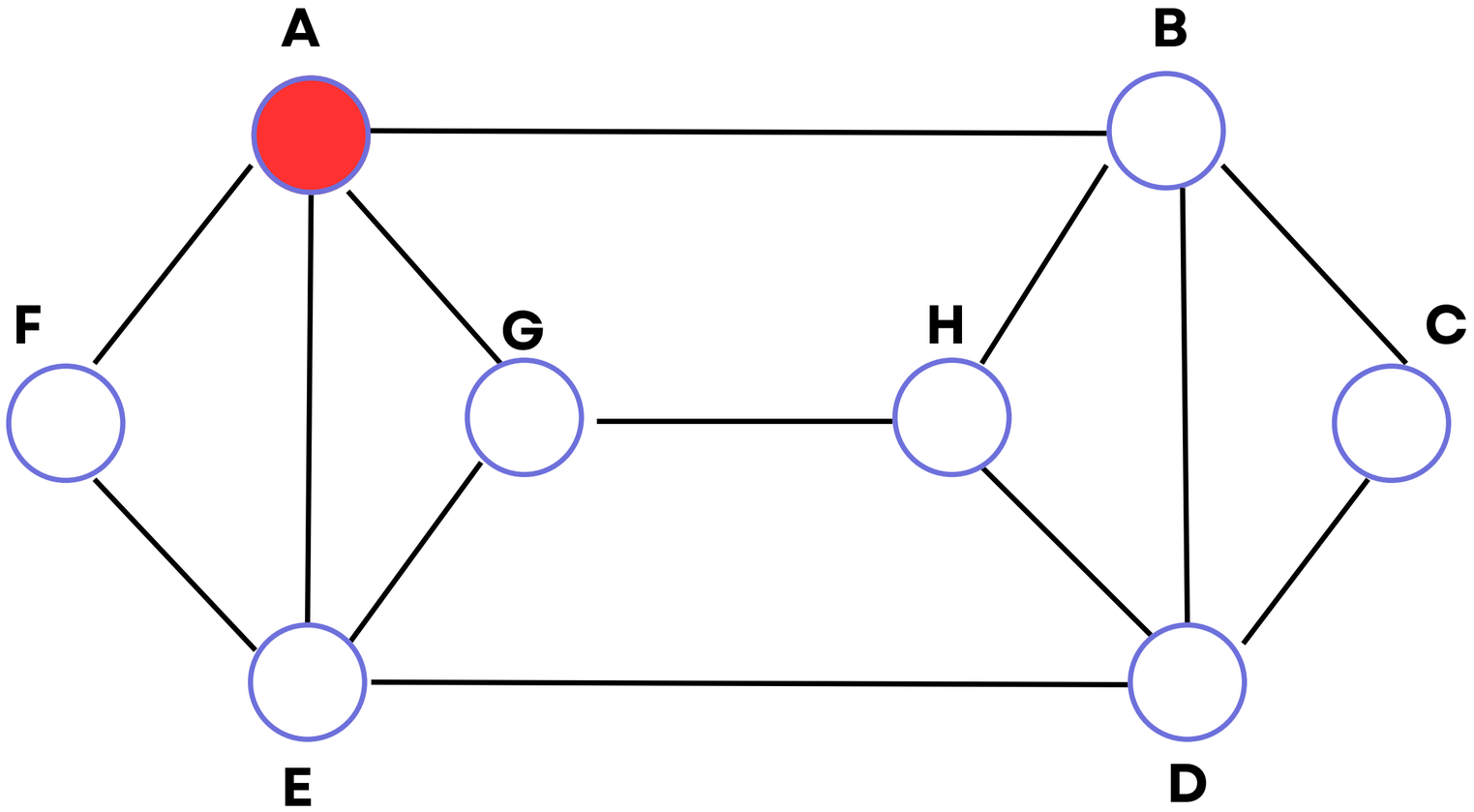
- **Etat intial-Avant DSATUR :**



Sommet	Degré	Dsat	coleur
A	4	0	
B	4	0	
D	4	0	
E	4	0	
G	3	0	
H	3	0	
C	2	0	
F	2	0	

EXMEPLE DE DÉROULEMENT

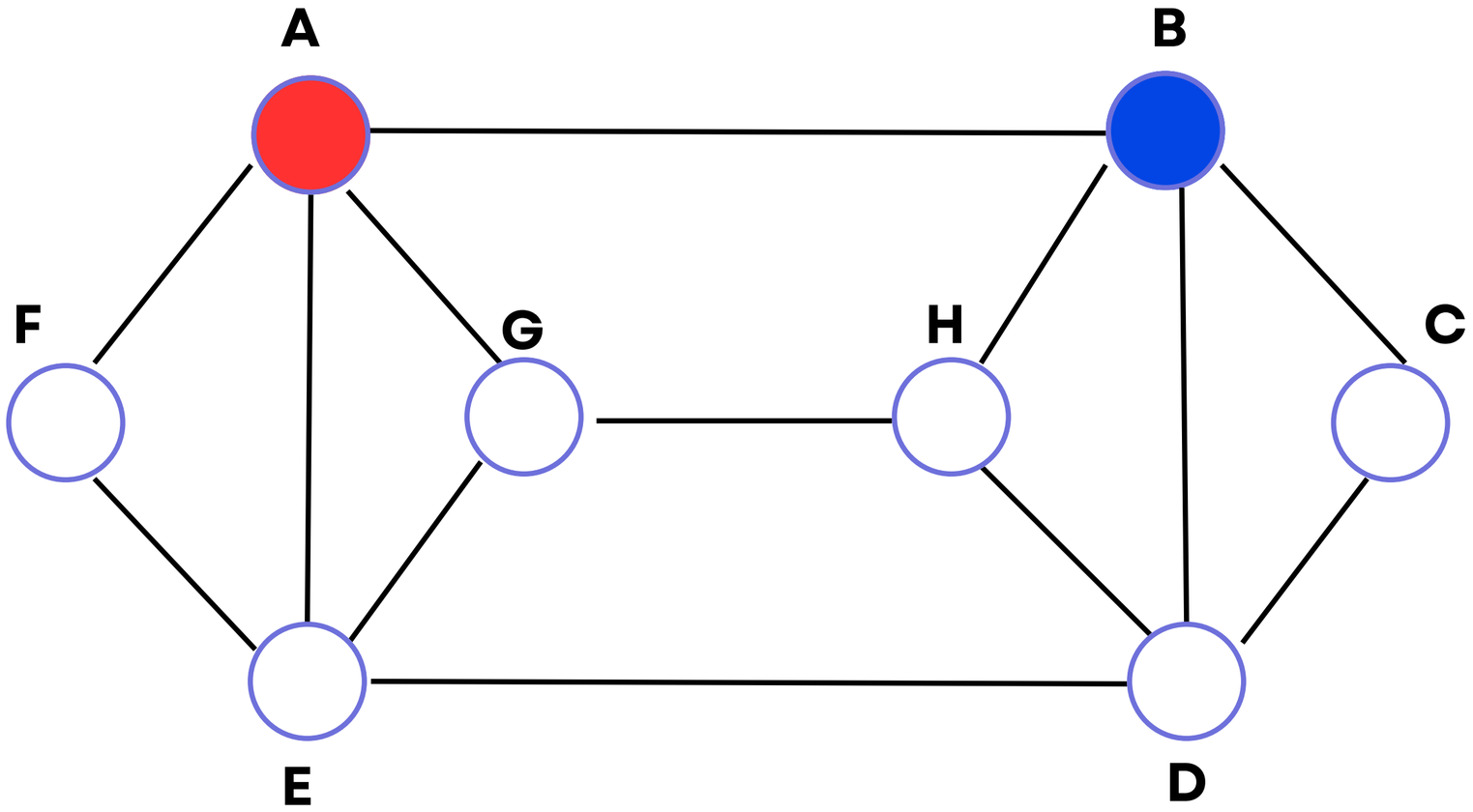
- **Etape1 :**



Sommet	Degré	Dsat	coleur
A	4		Rouge
B	4	1	
D	4	0	
E	4	1	
G	3	1	
H	3	0	
C	2	0	
F	2	1	

EXMEPLE DE DÉROULEMENT

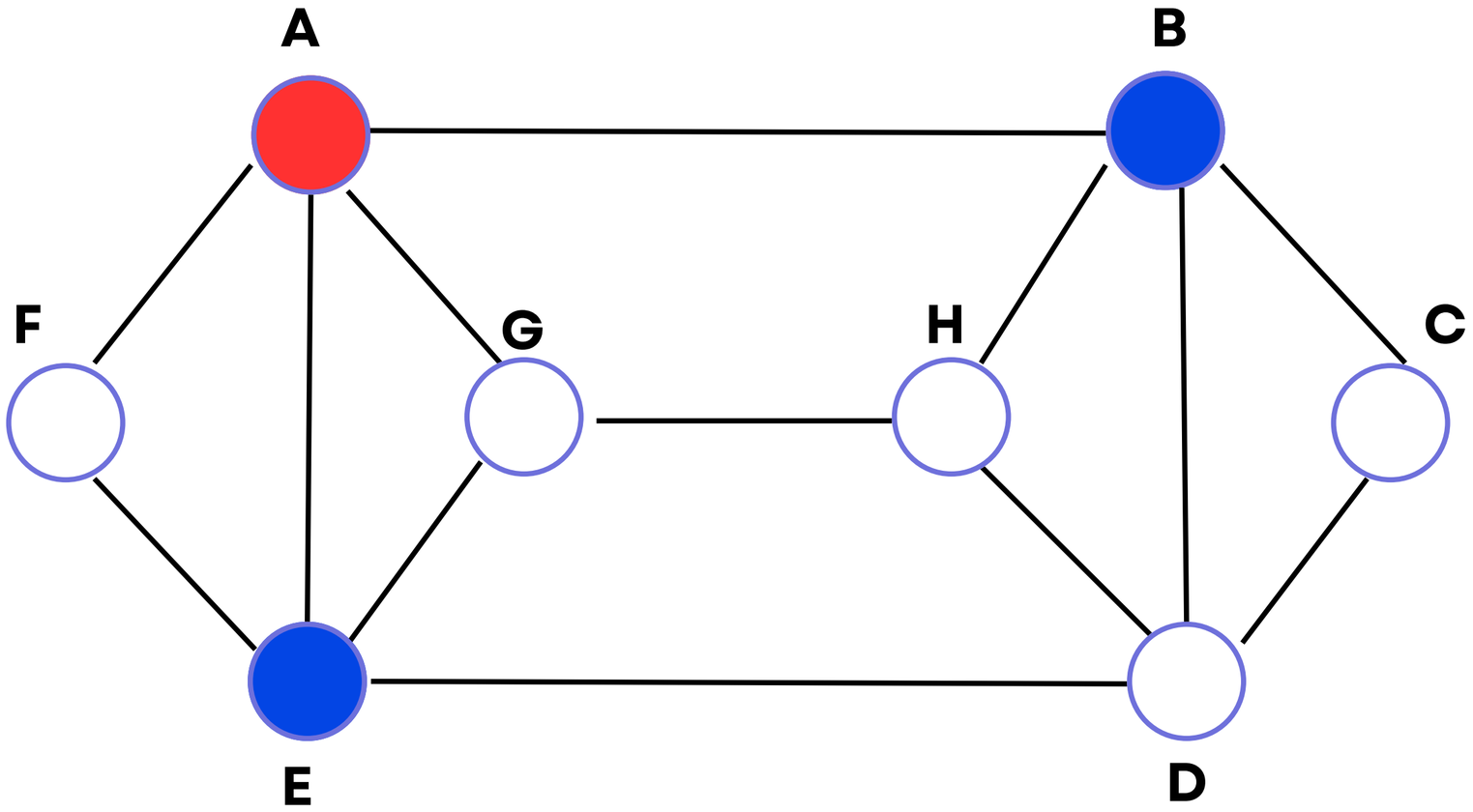
- **Etape 2 :**



Sommet	Degré	Dsat	coleur
A	4		Rouge
B	4		Bleu
D	4	1	
E	4	1	
G	3	1	
H	3	1	
C	2	1	
F	2	1	

EXMEPLE DE DÉROULEMENT

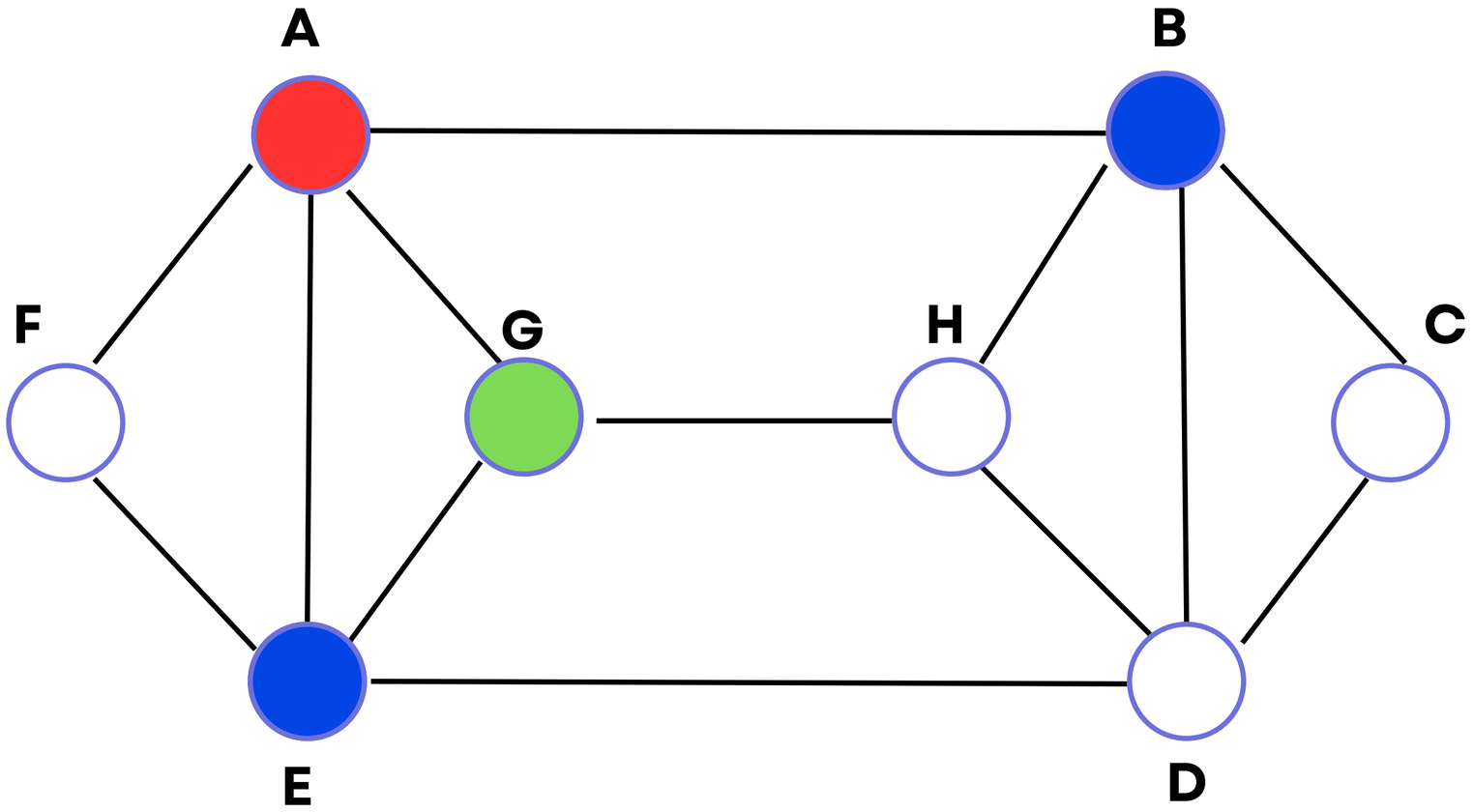
- **Etape 3 :**



Sommet	Degré	Dsat	coleur
A	4		Rouge
B	4		Bleu
D	4	1	
E	4		Bleu
G	3	2	
H	3	1	
C	2	1	
F	2	2	

EXMEPLE DE DÉROULEMENT

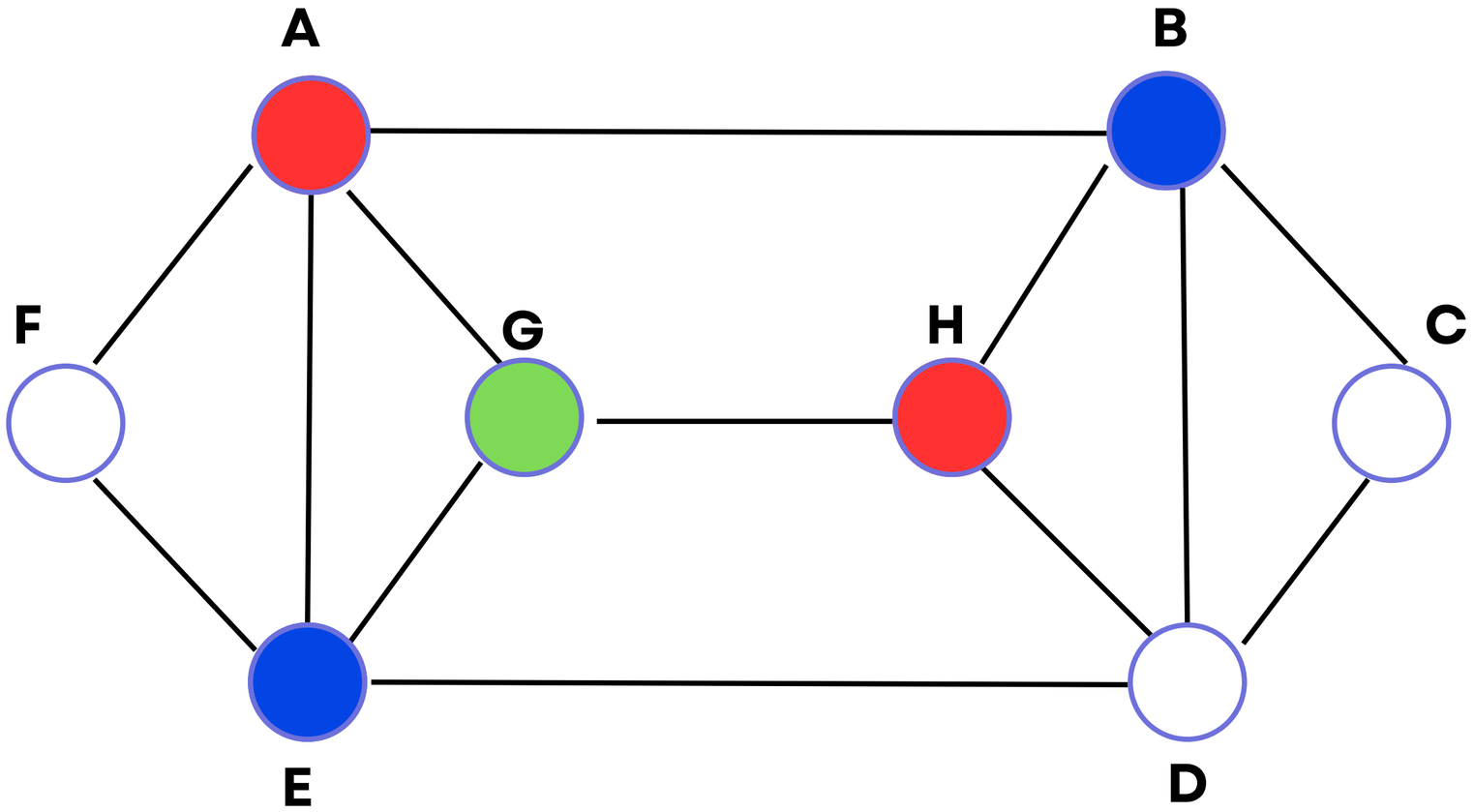
- **Etape 4 :**



Sommet	Degré	Dsat	coleur
A	4		Rouge
B	4		Bleu
D	4	1	
E	4		Bleu
G	3		Vert
H	3	2	
C	2	1	
F	2	2	

EXMEPLE DE DÉROULEMENT

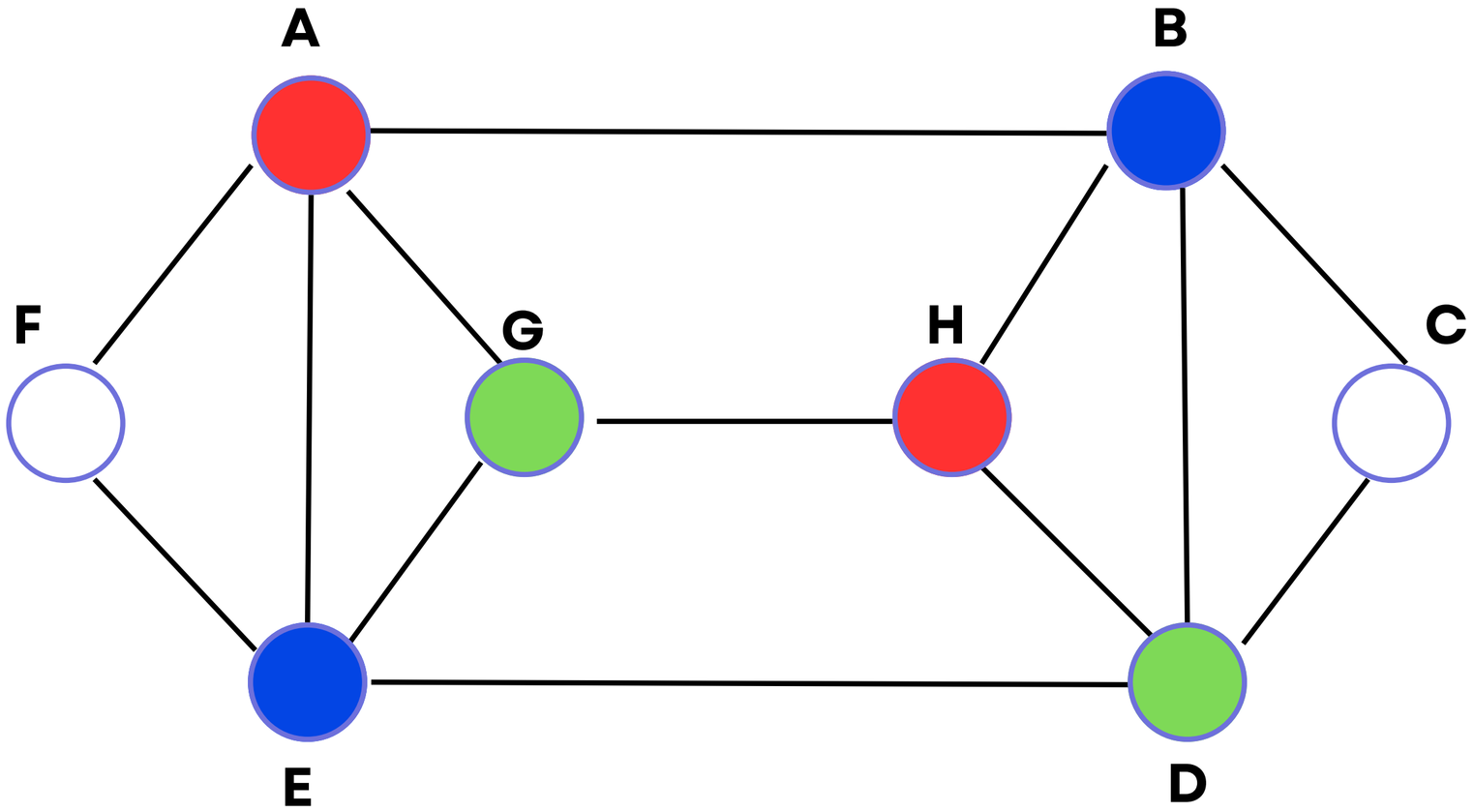
- **Etape 5 :**



Sommet	Degré	Dsat	coleur
A	4		Rouge
B	4		Bleu
D	4	2	
E	4		Bleu
G	3		Vert
H	3		Rouge
C	2	1	
F	2	2	

EXMEPLE DE DÉROULEMENT

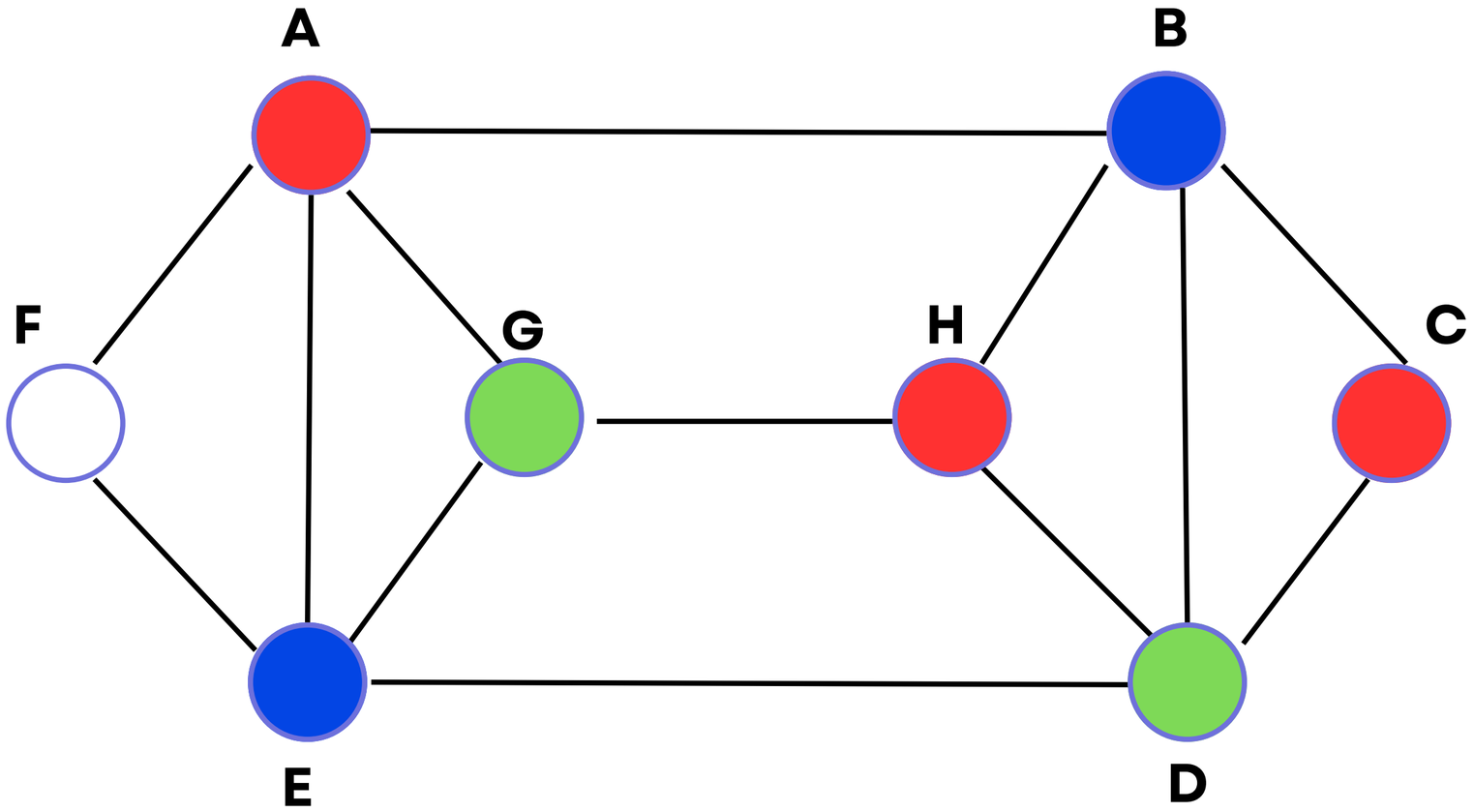
- **Etape 6 :**



Sommet	Degré	Dsat	coleur
A	4		Rouge
B	4		Bleu
D	4		Vert
E	4		Bleu
G	3		Vert
H	3		Rouge
C	2	2	
F	2	2	

EXMEPLE DE DÉROULEMENT

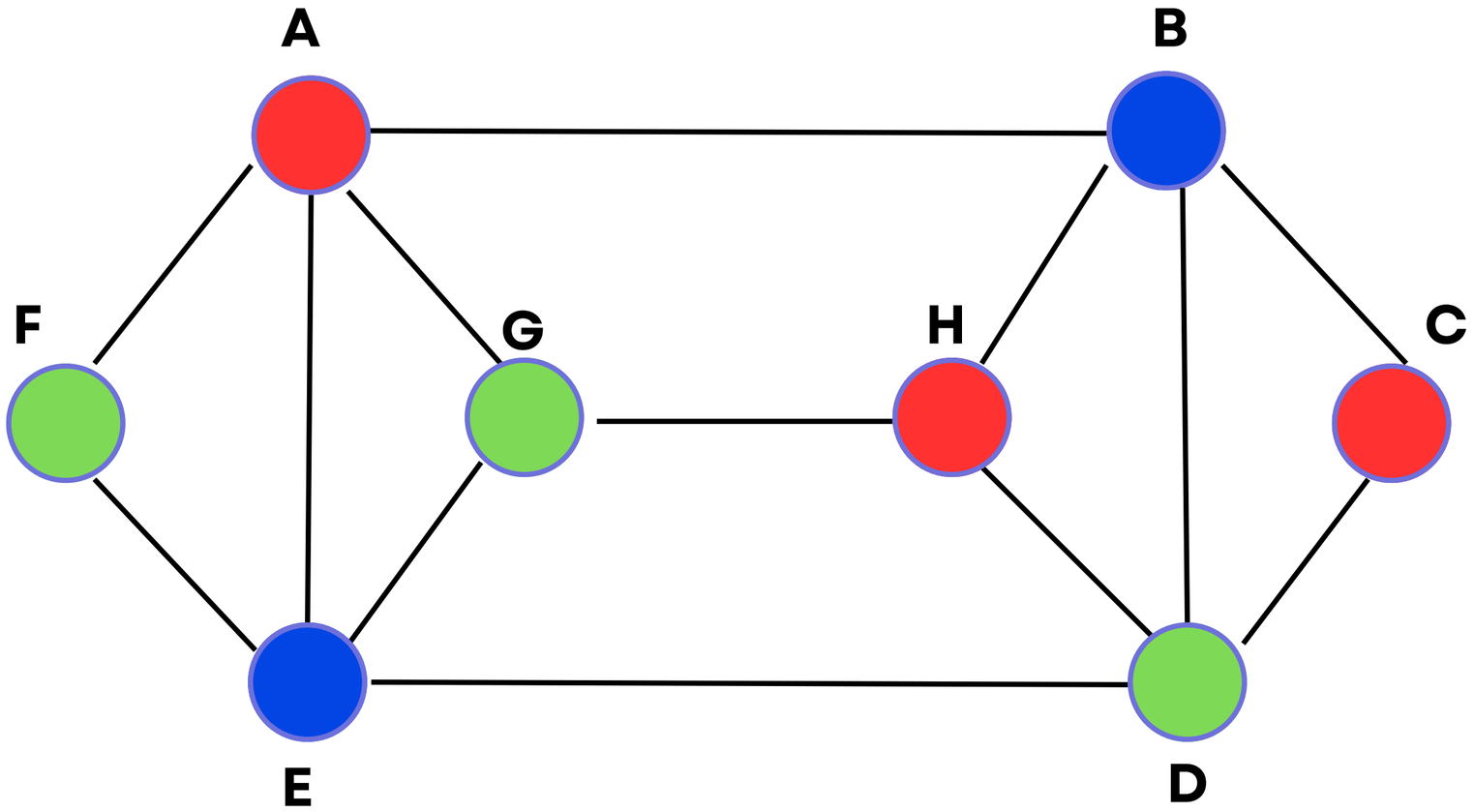
- **Etape 7 :**



Sommet	Degré	Dsat	coleur
A	4		Rouge
B	4		Bleu
D	4		Vert
E	4		Bleu
G	3		Vert
H	3		Rouge
C	2		Rouge
F	2	2	

EXMEPLE DE DÉROULEMENT

- **Etape dernière :**



Sommet	Degré	Dsat	coleur
A	4		Rouge
B	4		Bleu
D	4		Vert
E	4		Bleu
G	3		Vert
H	3		Rouge
C	2		Rouge
F	2		Vert

Complexité de l’algorithme DSATUR

On suppose un graphe = G(V,E) avec :

- **sommets ∣ V ∣ = n**
- **arêtes ∣ E ∣ = m**
- **degré maximal d’un sommet Δ**

1.Coût de la sélection du sommet à chaque étape :

À chaque itération, l’algorithme doit choisir le sommet non colorié :

- **ayant le plus grand degré de saturation DSAT**
- **et en cas d’égalité → le plus grand degré**

Cela nécessite souvent parcours de tous les sommets non coloriés.

➡ Coût : O(n)

Comme on fait cette opération pour chaque sommet, soit n fois :

➡ Total : $O(n^2)$

2. Coût de la recherche de la couleur disponible

Pour colorier un sommet, DSATUR regarde les couleurs présentes chez ses voisins.

Δ= Nombre maximum de voisins

➡ Coût par sommet : O(Δ)

➡ Total : O(n·Δ)

Comme Δ<=n, ce terme est borné par $O(n^2)$.

3. Mise à jour des degrés de saturation DSAT

Chaque fois qu'on colore un sommet, on met à jour les DSAT de ses voisins.

Chaque arête contribue au plus une mise à jour DSAT.

➡ Coût total : O(m)

Complexité totale :

En combinant tous les coûts :

$$O(m)+O(n\Delta)+O(n2)$$

Comme dans la majorité des graphes, on a :

- **n2 >= m**
- **n >= Δ**

Donc la complexité finale de DSATUR est :

★ **Complexité : O(n2)**

★ **Classe de complexité :**

Polynomiale (classe P).

la structure de données ,Pseudo code

la structure de données:

- Liste d’adjacence : adj[v] = liste des voisin
- Tableau des couleurs : couleur[v]
- Tableau DSAT : dsat[v]
- Tableau degrés : deg[v]
- Ensemble des sommets non coloriés : non_colories

Pseudo-code:

```
# Initialisation
pour chaque sommet v dans le graphe :
   couleur[v] = NON_COLORE
   dsat[v] = 0
   deg[v] = nombre de voisins de v
   ensemble_non_colories = ensemble de tous les sommets
tant que ensemble_non_colories n’est pas vide :

   # Sélection du sommet avec DSAT maximal puis degré maximal
   x = sommet dans ensemble_non_colories ayant :
         (dsat[x] maximal, puis deg[x] maximal en cas d'égalité)

   # Déterminer la plus petite couleur possible
   couleurs_voisins = couleurs utilisées par les voisins de x
   couleur_possible = 1
   tant que couleur_possible est dans couleurs_voisins :
      couleur_possible = couleur_possible + 1
      couleur[x] = couleur_possible

   # Mise à jour de la saturation des voisins
   pour chaque voisin y de x :
      si couleur[y] == NON_COLORE :
         couleurs_de_y = couleurs des voisins de y
         si couleur_possible n’est pas dans couleurs_de_y :
            dsat[y] = dsat[y] + 1
   retirer x de ensemble_non_colories
```

II

PARTIE 2 :

- ALGORITHME DYNAMIQUE
- PCC.Le plus court chemin
-Algorithme de JHONSON :
    1. Définition d’algorithme JHONSON
    2.entrée,sortie,objectif
    3.le principe de l’algoritme JHONSON
    4.Exmeple de déroulement
    5.Complexité de l’algorithme JHONSON
    6.la structure de données ,Pseudo code

03

Algorithme de Johnson:

Définition d’algorithme DSATUR :

- **L’algorithme de Johnson est une méthode efficace permettant de calculer les plus courts chemins entre toutes les paires de sommets dans un graphe orienté, pouvant contenir des poids négatifs, tout en évitant les cycles négatifs.**
  Il utilise :

L’algorithme Bellman-Ford pour repondérer le graphe,

L’algorithme Dijkstra pour accélérer les calculs après ré-pondération.

ENTRÉE,SORTIE,OBJECTIF

Entrée :

- Un graphe orienté G=(V,E)
- Des poids pouvant être positifs ou négatifs
- Aucune présence de cycle négatif

Sortie :

- Une matrice D contenant les plus courtes distances entre toutes les paires de sommets .

Objectif :

- Calculer rapidement toutes les distances minimales poids ,
- Exploiter la puissance de Dijkstra après une transformation des poids

LE PRINCIPE DE L'ALGORITME DSATUR

L'algorithme de Johnson repose sur l'idée de transformer un graphe qui contient des poids négatifs en un graphe dont tous les poids deviennent positifs, afin de pouvoir utiliser efficacement l'algorithme de Dijkstra.
Il combine donc deux techniques : Bellman-Ford pour la repondération, puis Dijkstra pour le calcul rapide des plus courts chemins.

◆ Etapes principales :

1. Étape 1 : Ajouter un sommet auxiliaire

- **Introduire un nouveau sommet S .**
- **Relier S à tous les sommets existants par des arcs de poids 0.**

. Ce sommet sert de point de départ unique pour le calcul des valeurs h(v) de chaque sommet.

2. Étape 2 : Exécuter Bellman-Ford depuis

a. Lancer l'algorithme de Bellman-Ford à partir du sommet S.
b.Calculer pour chaque sommet une valeur qui représente le potentiel du sommet.
c.Vérifier l'absence de cycle négatif : si un cycle négatif est détecté, l'algorithme s'arrête.
d.Les valeurs serviront à transformer les poids du graphe.

3. Étape 3 : Repondérer les arcs

- Pour chaque arc (u,v) , recalculer le poids selon :

  $$w'(u,v) = w(u,v) + h(u) - h(v)$$

- Les distances relatives entre les sommets restent identiques.

4. Étape 4 : Appliquer Dijkstra depuis chaque sommet

-Pour chaque sommet S du graphe :

- Exécuter l'algorithme de Dijkstra sur le graphe avec les poids modifiés w' .
-Obtenir les distances modifiées d'(s,v) vers tous les autres sommets.

5. Étape 5 : Calculer les distances réelles

-Pour chaque couple de sommets (s,v) :

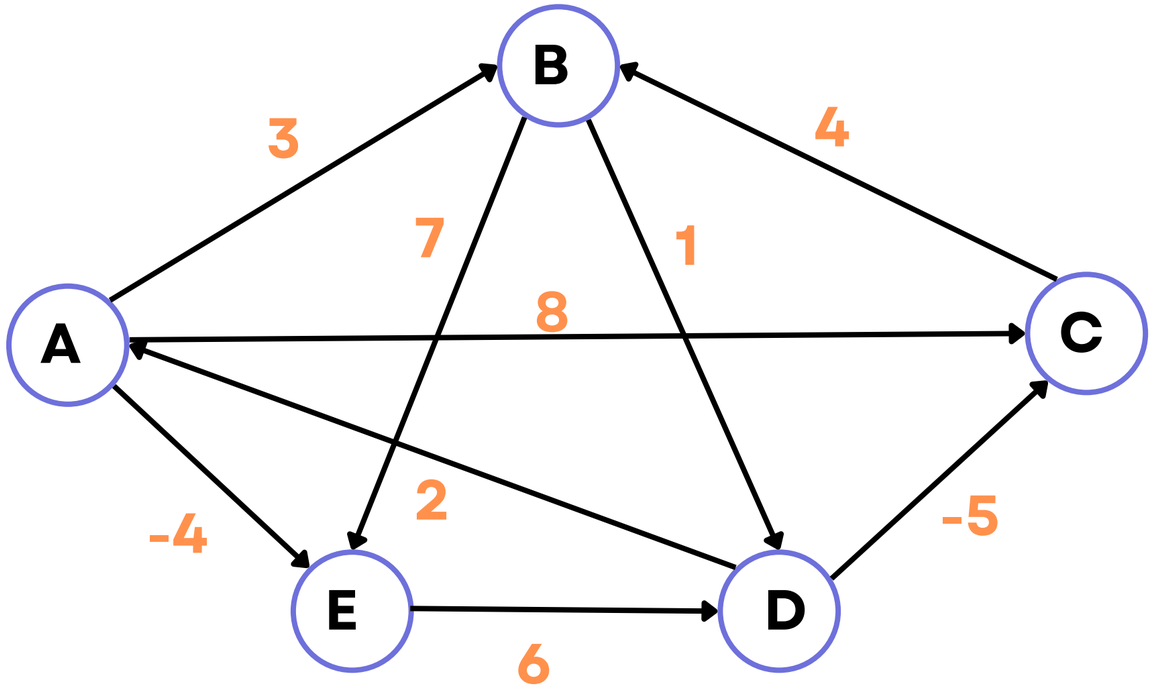
$$d(s,v) = d'(s,v) - h(s) + h(v)$$

-Les distances peuvent être négatives si le graphe contient des arcs négatifs, mais aucun cycle négatif n'est présent.

EXMEPLE DE DÉROULEMENT

- **Présentation du graphe :**

« Soit un graphe orienté G =(V,E)
|V|=5
|E|=9
avec des poids positifs et négatifs

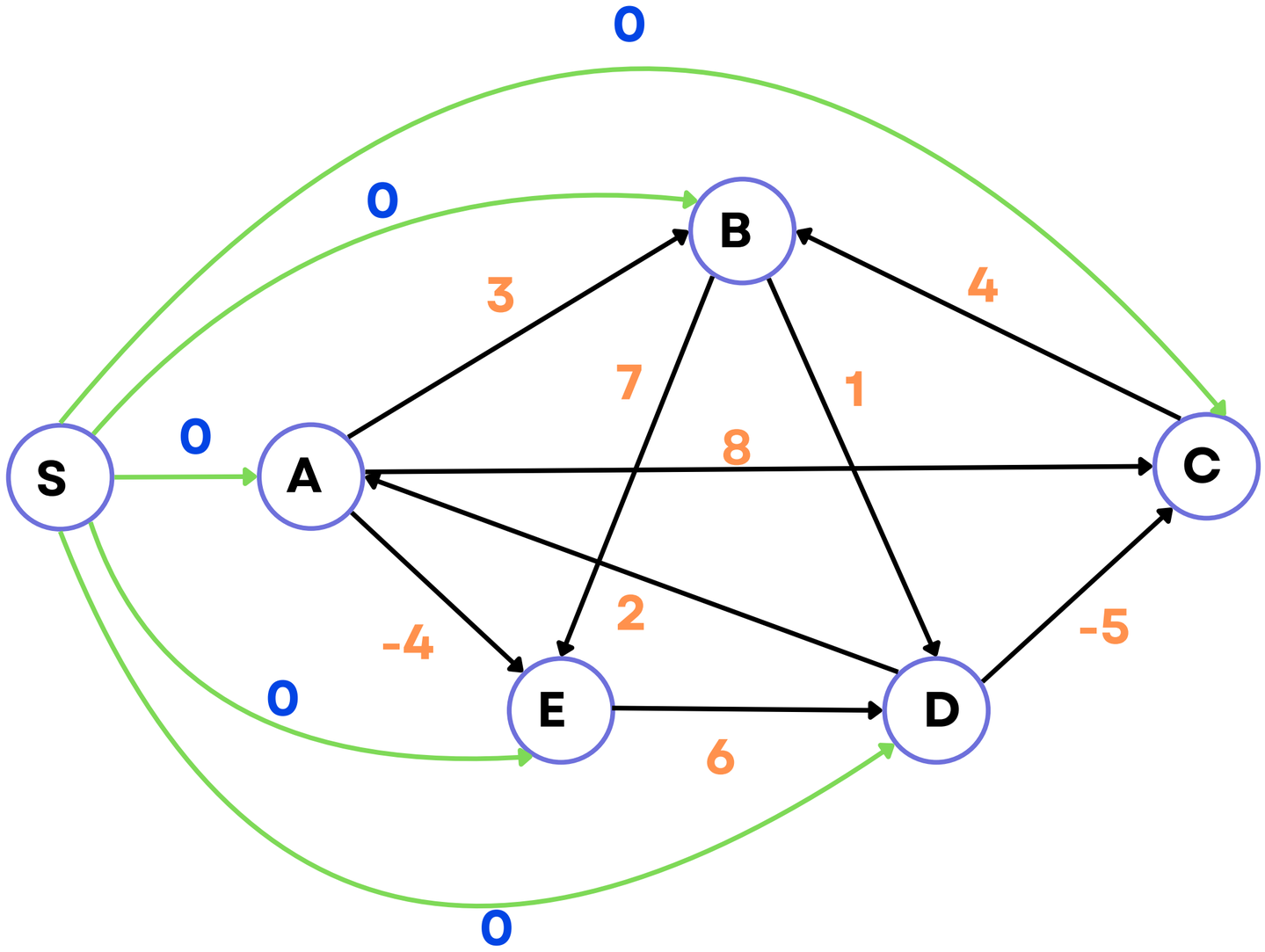


◆ Etape 1 : “Ajouter un sommet auxiliaire S “

- Ajouter un sommet S et relier S à tous les sommets de G avec des arcs de poids 0

G’ =(V’,E’)

|V’| = 5+1=6



◆ Etape 2 : “Exécuter Bellman-Ford depuis S “

- **Exécuter Bellman-Ford à partir du sommet q pour calculer les potentiels h(v) pour chaque sommet v.**

k	$\lambda^k(A)$	$\lambda^k(B)$	$\lambda^k(C)$	$\lambda^k(D)$	$\lambda^k(E)$
0(init))	∞	∞	∞	∞	∞
1	0	0	0	0	0
2	0	0	-5	0	-4
3	0	-1	-5	0	-4
4	0	-1	-5	0	-4
5(fin)	0	-1	-5	0	-4

◆ Etape 3 : “Repondérer les arcs (réducation des poids)”

- **Pour chaque arc (u,v) dans G,définir :**

  w'(u,v) = w(u,v) + h(u) - h(v)

- **On a:**

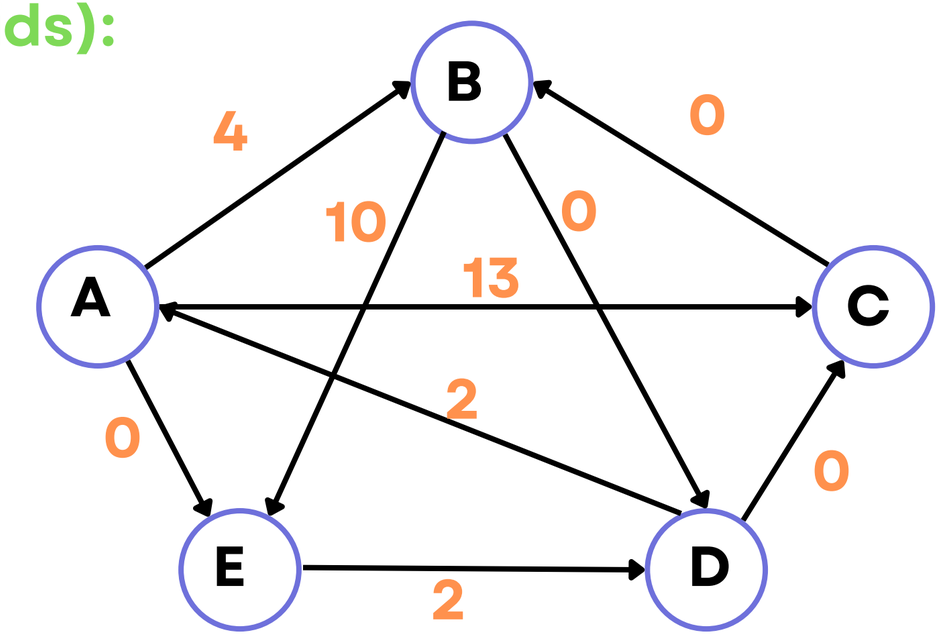
  h(A)=0,h(B)=-1,h(C)=-5,h(D)=0,h(E)=-4

- **alors:**

  w'(A,B) = w(A,B) + h(A) - h(B)=3+0-(-1)=4

  w'(A,C) = w(A,C) + h(A) - h(C)=8+0-(-5)=13 .....

Le nouveau grave(aprés le recalcul des poids):



	w(u,v)	h(u)	h(v)	w’(u,v)
A->B	**3**	0	-1	4
A->C	8	0	-5	13
A->E	-4	0	-4	0
B->D	1	-1	0	0
B->E	7	-1	-4	10
C->B	4	-5	-1	0
D->A	2	0	0	2
D->C	-5	0	-5	0
E->D	6	-4	0	2

◆ Etape 4 : “Appliquer Dijkstra depuis chaque sommet”

- **Pour chaque sommet $s \in V$, exécuter Dijkstra sur le graphe avec les poids w' pour obtenir les distances $d'(s,v)$.**

La matrice des distances modifiées:

Source / But	A	B	C	D	E
A	0	2	2	2	0
B	2	0	0	O	2
C	2	0	0	0	2
D	2	0	0	0	2
E	4	2	2	2	0

◆ Etape 5 : “Calculer les distances finales”

- **Pour chaque couple (s,v), convertir chaque valeur en distance réelle en utilisant la formule :**

  $d(u,v)=d'(u,v)-h(u)+h(v)$

  où h représente les potentiels calculés précédemment avec Bellman–Ford.

  $h(A)=0, h(B)=-1, h(C)=-5, h(D)=0, h(E)=-4$

  Donc:

  $d(A,A)=d'(A,A)-h(A)+h(A)=0-0+0=0$

  $d(A,B)=d'(A,B)-h(A)+h(B)=2-0+(-1)=1$.....

La matrice des distance réelles:

Source \ But	A	B	C	D	E
A	0	1	-3	2	-4
B	3	0	-4	1	-1
C	7	4	0	5	3
D	2	-1	-5	0	-2
E	8	5	1	6	0

Complexité de l’algorithme de Johnson

L’algorithme comprend 3 grandes parties :

- **Bellman–Ford**
- **Repondération des arcs**
- **Dijkstra lancé depuis chaque sommet**

1. Complexité de Bellman–Ford :

Bellman–Ford calcule les distances d’un sommet sss vers tous les autres.

◆ Fonctionnement

Bellman-Ford fait |V| – 1 itérations.

À chaque itération, il parcourt toutes les arêtes du graphe.

◆ Coût d’une itération :

Parcourir toutes les arêtes → O(E)

◆ Nombre total d’itérations :

| V | –1 ≈ O(V)

✓ Complexité totale :

$$O(V \times E)$$

2. Complexité de la repondération des arcs :

On modifie le poids de chaque arête :

$$w'(u,v)=w(u,v)+h(u)-h(v)$$

Cela nécessite :

✓ **Parcourir toutes les arêtes**

✓ **Effectuer une opération constante par arête**

✓ **Complexité totale :**

$$O(E)$$

Très faible par rapport aux autres étapes.

3. Complexité de Dijkstra lancé |V| fois :

Après repondération, tous les poids sont positifs, donc on peut appliquer Dijkstra efficacement.

Selon les structures de données utilisées :

▶ **Avec un tas binaire (binary heap) (standard)**

Dijkstra pour un seul sommet :

$$O(ElogV)$$

Pourquoi ?

Opération	Nombre	Coût	Total
Extraire min	V fois	log V	V log V
Mise à jour (relaxation)	E fois	log V	E log V

L’opération dominante : E log V
Dijkstra pour tous les sommets :
On répète Dijkstra |V| fois.

$$O(V \times E \log V)$$

⬅ END Complexité totale de l’algorithme de Johnson :

On additionne les 3 parties :

$$O(VE)+O(E)+O(VE\log V)$$

On garde les termes les plus importants :

$$O(VE+VElogV)$$

: Comme VElogV domine presque toujo :

$$O(VElogV)$$

⚠ Cas particuliers selon le type de graphe

◆ Graphe dense : $E \approx V^2$

Alors :

$$O(V(V2)logV)=O(V3logV)$$

Dans ce cas, Johnson ≈ Floyd–Warshall ?

- **Johnson : V3logV**
- **Floyd : V3**

➡ Floyd devient plus rapide si le graphe est très dense.

◆ Graphe sparse (creux) : E ≈ V

Alors :

$$O(V2logV)$$

➡ Johnson devient beaucoup plus rapide que Floyd-Warshall.

🎯 Classe de complexité

L’algorithme est polynomial.

Classe : P

la structure de données ,Pseudo code

- **la structure de données:**

Graphe

Représenté par :
Liste d'adjacence (adjacency list) : chaque sommet a une liste des arcs sortants avec leur poids.
Ou matrice d'adjacence (moins efficace pour grands graphes).

Potentiels

Un dictionnaire ou tableau h[v] pour stocker les valeurs calculées par Bellman-Ford.

Distances

Un dictionnaire ou tableau d[v] pour stocker les distances depuis un sommet donné.
Pour Dijkstra, on peut utiliser une file de priorité (priority queue / min-heap) pour sélectionner le sommet avec la distance minimale.

Prédécesseurs

Tableau prev[v] pour reconstruire le chemin le plus court.

Pseudo-code:

```
Algorithme Johnson(G):
  # G est le graphe avec poids w(u,v)

  # Étape 1 : Ajouter sommet q
  Ajouter un sommet q
  Pour chaque sommet v dans G:
    Ajouter l'arc (q, v) avec w(q,v) = 0

  # Étape 2 : Calcul des potentiels avec Bellman-Ford
  h = BellmanFord(G, q)
  Si Bellman-Ford détecte un cycle négatif:
    Retourner "Impossible, cycle négatif"

  # Étape 3 : Repondération des arcs
  Pour chaque arc (u,v) dans G:
    w'(u,v) = w(u,v) + h[u] - h[v]

  # Étape 4 : Dijkstra depuis chaque sommet
  Pour chaque sommet s dans G:
    d' = Dijkstra(G, w', s)

  #Étape 5 : Calculer les distances réelles
    Pour chaque sommet v:
      d[s][v] = d'[v] - h[s] + h[v]

  Retourner la matrice d[s][v] contenant toutes les distances
```

04 MODE D’EMPLOI

LES INSTALLATIONS REQUISES:

-**Language de dévloppement** : Python (version 3.13.9) :

-**lien** :https://www.python.org/downloads/

-**Environnement de dévloppement** : Visual Studio Code (version 1.90)

-**lien**: https://code.visualstudio.com/

-**Bibliothèque** :

05

RÉSULTATS

- **Algorithme DSATUR :**
  - **Démonstration d’algorithme Dsatur :**

Algorithme JHONSON :

Démonstration d’algorithme Jhonson :

06

ANALYSE CRITIQUE ET CONCLUSION

Algorithme DSATUR :

Difficultés rencontrée :

- **Implémentation complexe : gestion dynamique du degré de saturation à chaque étape.**
- **Mise à jour fréquente des couleurs voisines, ce qui augmente la charge computationnelle.**
- **Nécessite une structure de données efficace (priority queue) pour maintenir l'ordre des sommets.**

Limites :

- **Pas optimal pour les graphes très grands ou très denses (croissance quasi-exponentielle dans le pire cas).**
- **Reste un algorithme glouton : peut produire une coloration non optimale selon la structure du graphe.**
- **Dépend fortement de l'ordre de choix des sommets (heuristique sensible à la topologie).**

Points positifs :

- **Très performant sur la majorité des graphes réels (structure non pathologique).**
- **Souvent proche de l'optimal et donne un nombre de couleurs minimal dans de nombreux cas pratiques.**
- **Implémentation efficace possible avec de bonnes structures de données.**
- **Simple à comprendre conceptuellement : saturations → couleurs.**

Algorithme JHONSON :

Difficultés rencontrée :

- **Implémentation technique : étape de repondération avec Bellman-Ford puis Dijkstra.**
- **Gestion de graphes avec arêtes négatives nécessite une attention particulière.**
- **Combinaison de plusieurs algorithmes augmente la complexité du code.**

Limites :

- **Pas applicable si le graphe contient un cycle de poids négatif (échec du Bellman-Ford).**
- **Plus coûteux que d'autres algorithmes dans les très petits graphes (overhead inutile).**
- **La performance dépend fortement de la qualité de l'implémentation du tas (priority queue).**

Points positifs :

- **Permet d’utiliser Dijkstra efficacement même en présence de poids négatifs → très puissant.**
- **Temps d’exécution optimal pour les graphes clairsemés et de taille moyenne.**
- **Combinaison robuste : Johnson = Bellman-Ford + Dijkstra → résultats fiables et rapides.**
- **Très adapté au calcul de tous les plus courts chemins dans les grands graphes.**

MERCI POUR VOTRE ATTENTION