Parcours des graphes

S2-3-3 GRAPHES - Parcours

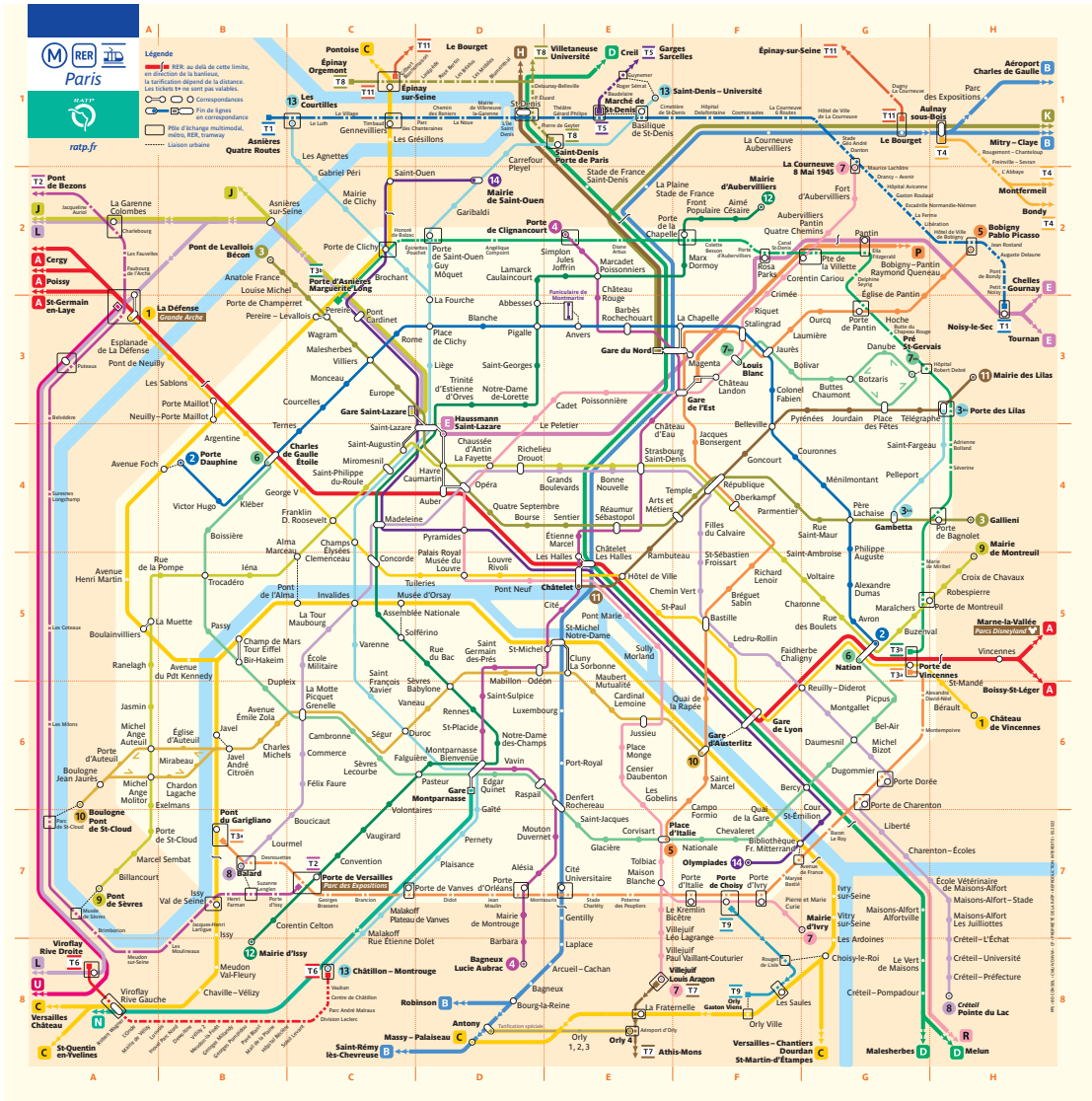
Germain Gondor

Lycée Carnot - Dijon, 2022 - 2023

Sommaire

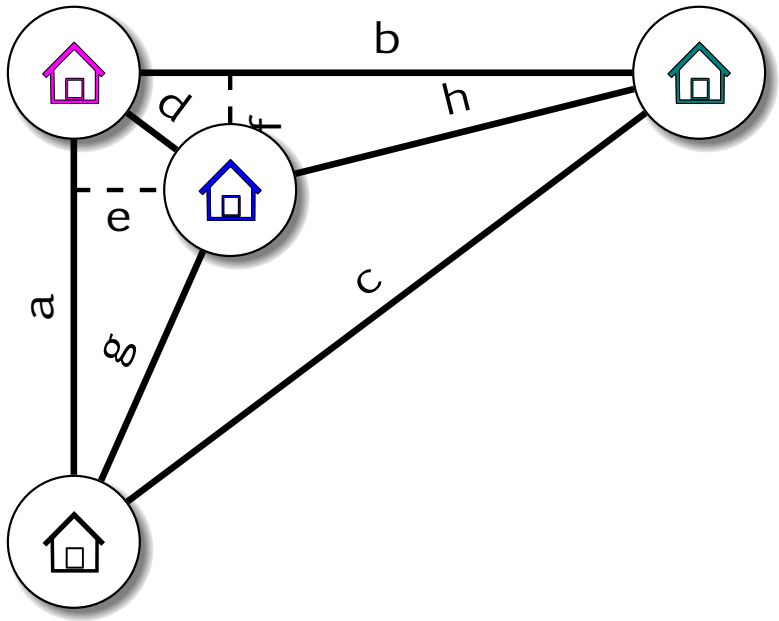
Sommaire

Métro parisien



Problématique

Lors de l'étude des algorithmes gloutons, le problème du voyageur de commerce était de passer une et une seule fois par chacun des sommets d'un graphe et retourner au sommet initial en minimisant la longueur du trajet :



Problématique

La problématique est bien différente quand il s'agit de se rendre de la ville A vers la ville B via le réseau routier, en minimisant la distance parcourue.

La théorie des 6 poignées de main énonce que chaque être humain est potentiellement en relation avec n'importe quel autre être humain sur la planète à l'aide d'une chaine de relations individuelles d'au plus 6 éléments. Comment alors trouver les maillons ?

Le but de ce cours est de voir différentes méthodes de parcours des graphes puis d'étudier des algorithmes permettant de déterminer le (ou un des) plus court(s) chemin(s) entre deux sommets.

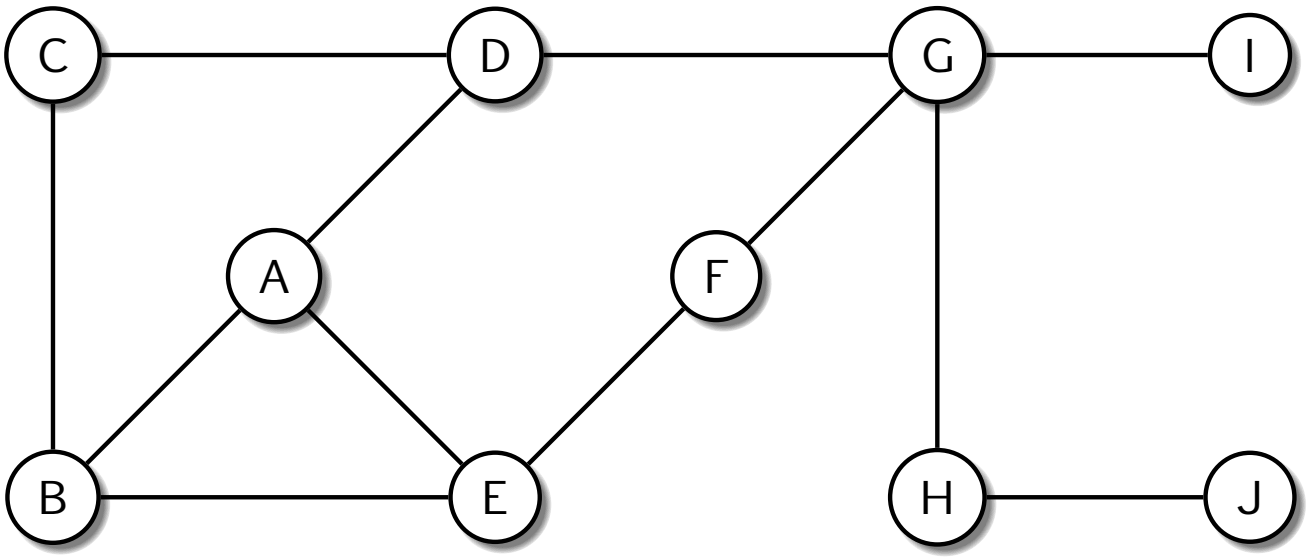
Sommaire

Méthodes de parcours des graphes

Définition : Parcourir un graphe

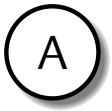
> *Parcourir un graphe, c'est obtenir la liste des sommets atteignables du graphe en partant d'un des sommets.*

Le graphe ci-contre sert de support à l'étude où le sommet A sera le sommet initial.



Méthodes de parcours des graphes

On adoptera les représentations graphiques suivantes :



Sommet non découvert

Sommet découvert

Sommet exploré

——— arrête explorée – – – arrête non découverte ····· arrête découverte mais inutile

Je suis ton père

Lors du parcours d'un graphe, on appelle sommet fils (*chil* ou successeur) d'un sommet père (*parent* ou prédécesseur) un sommet découvert depuis le sommet père. Si un sommet père peut avoir plusieurs sommets fils, un sommet fils n'a qu'un sommet père.

Dans ce cours nous verrons trois modes de parcours des graphes :

- parcours en profondeur
- parcours en largeur
- algorithme de Dijkstra et Dijkstra*

Parcours en profondeur

Principe :

> *A partir du sommet de départ, on explore un de ses sommets voisins, puis un des sommets voisin du sommet voisin et ainsi de suite. Lorsqu'un sommet n'a plus de voisin à explorer, on reprend l'exploration à partir du dernier sommet père dont au moins un voisin n'a pas été exploré.*

Les algorithmes Algo 1 et Algo 2 présentent une version récursive et une version itérative de parcours en profondeur (ou dfs - *depth first search*) d'un graphe stocké sous forme de dictionnaire de dictionnaires.

Version récursive

- Les sommets explorés sont stockés dans une liste `parcours`.
- Une liste `attente` recense les voisins d'un sommet non encore explorés.
- Pour éviter de chercher dans la liste `parcours` si un sommet a déjà été exploré, le dictionnaire `explore` a pour clés les noms des sommets et pour valeur `True` si le sommet a été exploré, `False` sinon.
- La fonction `dfs_recursif` prend en argument :
  - `graphe`, un graphe sous la forme d'un dictionnaire (clé = nom d'un sommet, valeur = noms des voisins du sommet clé)
  - `sommet`, le nom du sommet de départ
  - la liste `parcours`
  - le dictionnaire `explore`. Au premier appel, seule la clé `sommet` a une valeur `True`

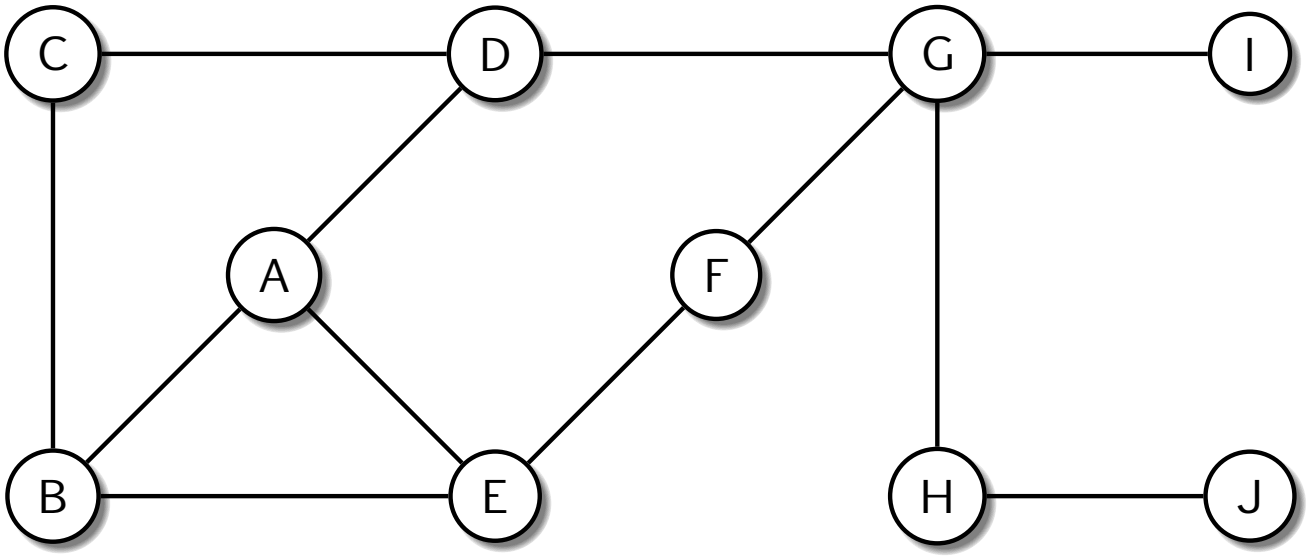
Version récursive

Algorithm 1 Parcours en profondeur récursif

entrée: `graphe`, `sommet`, `parcours` et `explore`
résultat: `parcours`, liste des sommets explorés
dfs_recursif(`graphe`, `sommet`, `parcours`, `explore`)

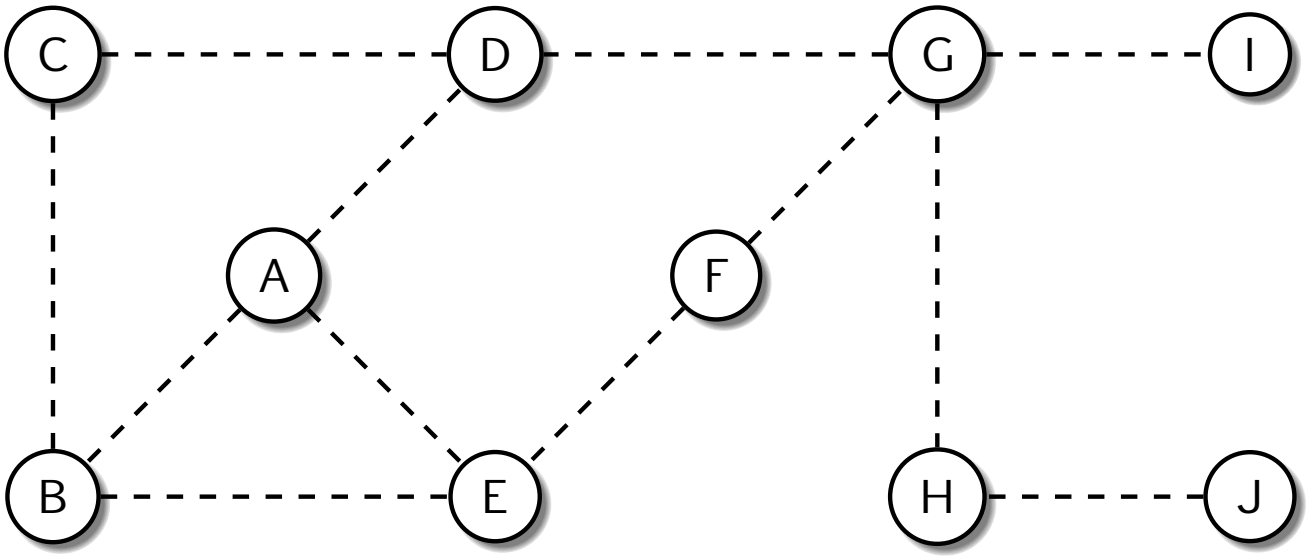
```
1:  si non explore[sommet] alors
2:      ajouter sommet à parcours
3:      explore[sommet] = Vrai
4:  fin si
5:  pour s voisin de sommet faire
6:      si non explore[s] alors
7:          ajouter s à attente
8:      fin si
9:  fin pour
10: pour s dans attente faire
11:     dfs_recursif(graphe, s, parcours, explore)
12: fin pour
13: renvoi: parcours
```

Version récursive



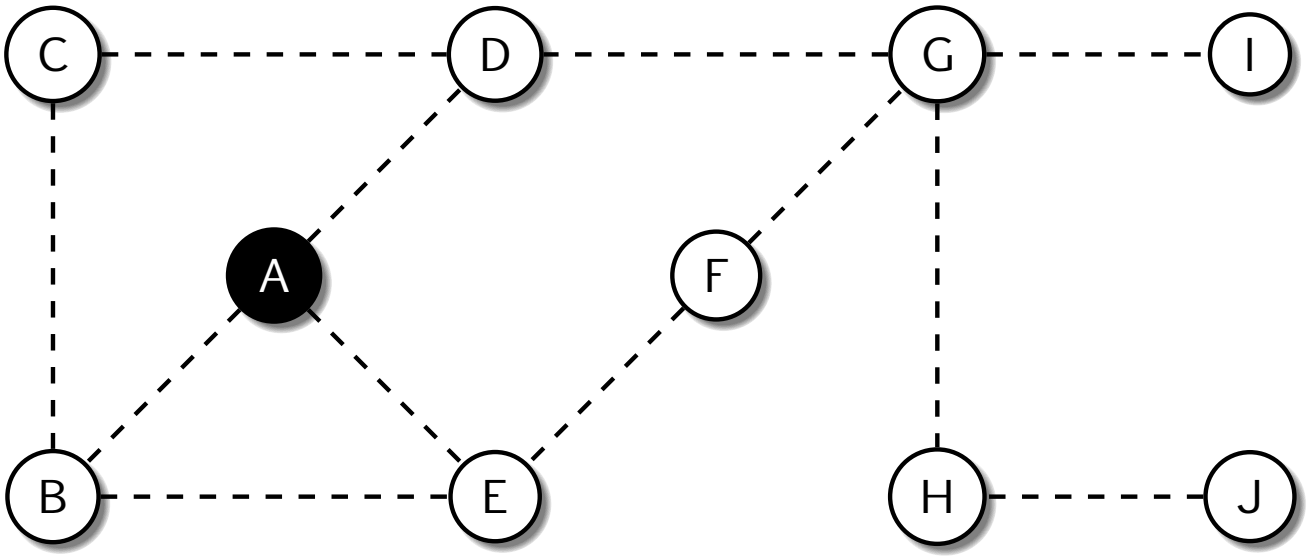
`parcours :`

Version récursive



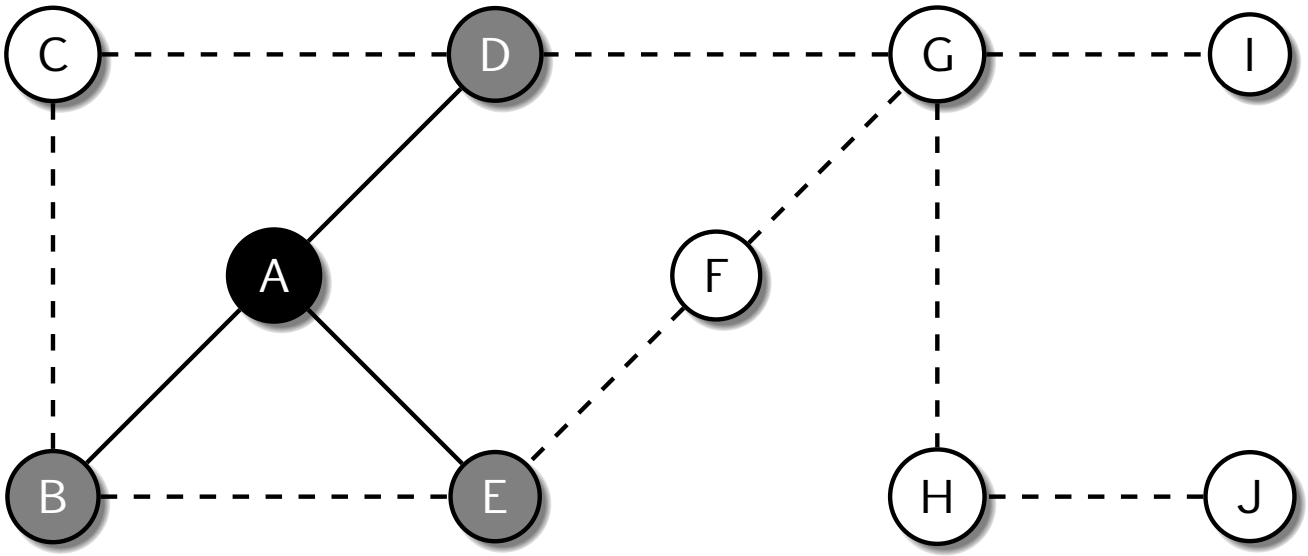

```
parcours:
 Attente:
```

Version récursive




```
parcours : A
 Attente :
```

Version récursive



`parcours : A`

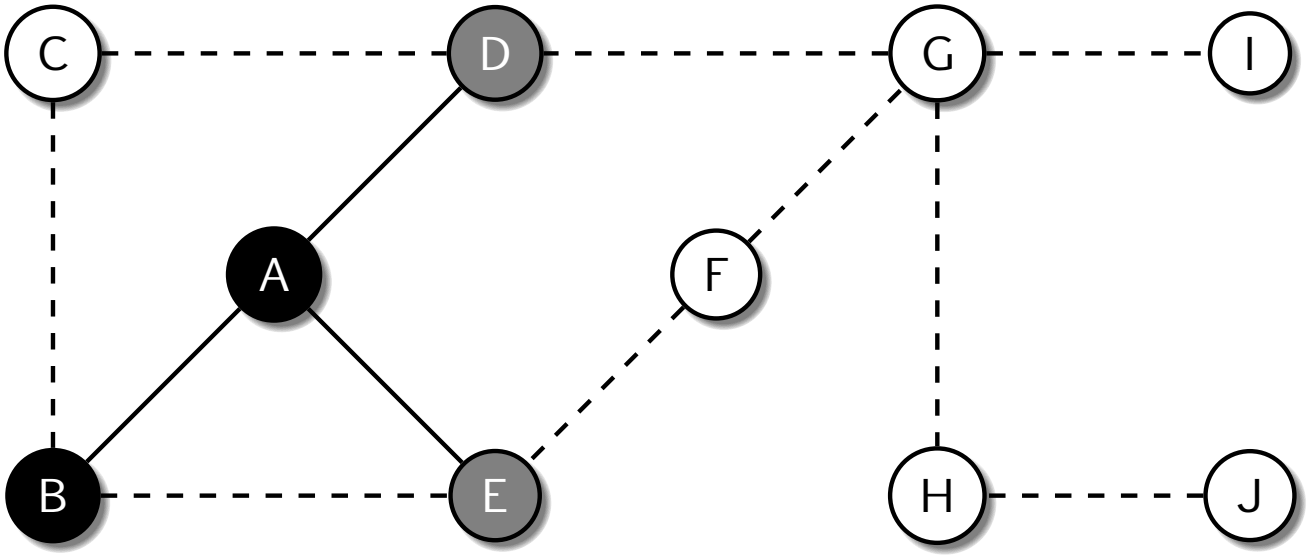
`Attente issue de A : B D E`

Version récursive

Parcours en profondeur récursif

entrée: `graphe`, `sommet`, `parcours` et `explore`
résultat: `parcours`, liste des sommets explorés
dfs_recursif(`graphe`, `sommet`, `parcours`, `explore`)
si non `explore[sommet]` **alors**
    ajouter `sommet` à `parcours`
    `explore[sommet] = Vrai`
fin si
pour `s` voisin de `sommet` **faire**
    si non `explore[s]` **alors**
        ajouter `s` à `attente`
    fin si
fin pour
pour `s` dans `attente` **faire**
    dfs_recursif(graphe, s, parcours, explore)
fin pour
renvoi: `parcours`

Version récursive

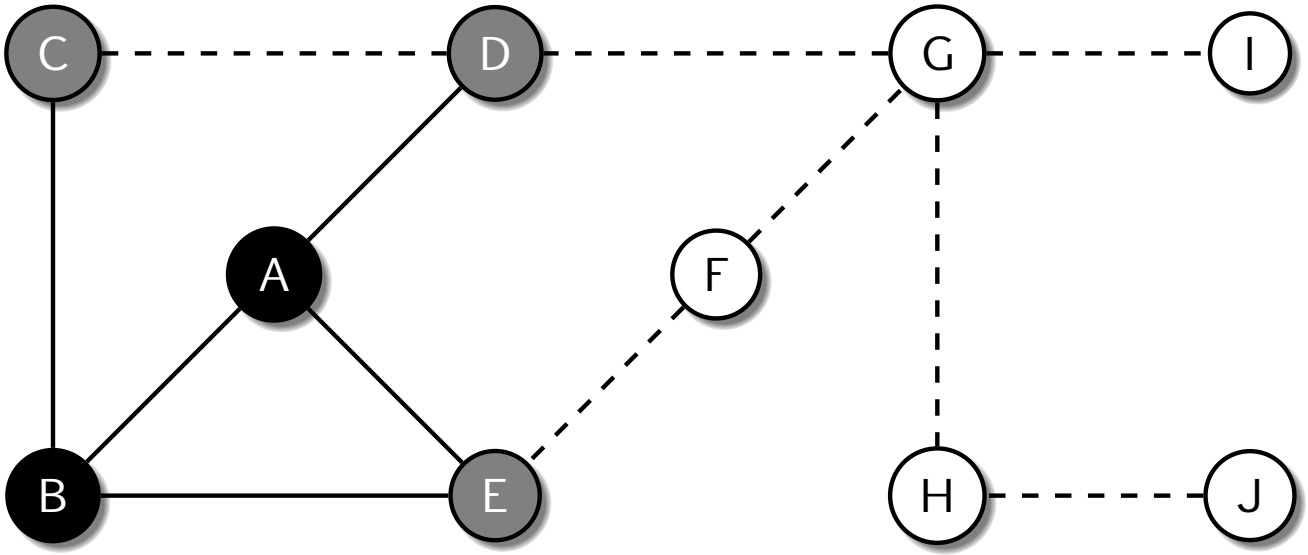


parcours : A B

Attente issue de A : D E

Attente issue de B :

Version récursive



parcours : A B

Attente issue de A : D E

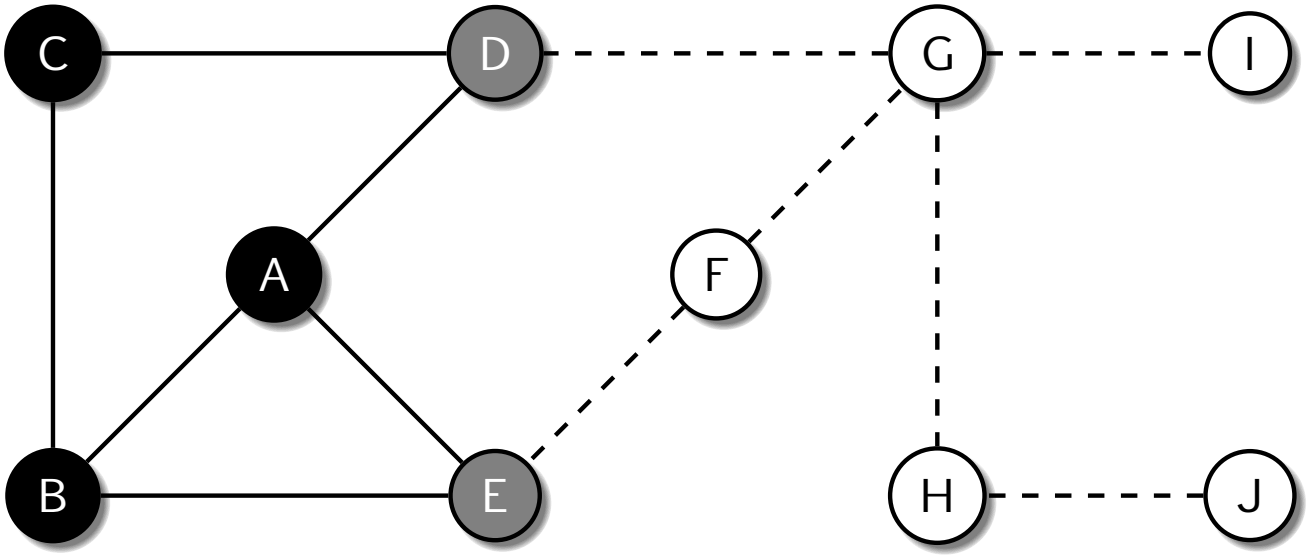
Attente issue de B : C E

Version récursive

Parcours en profondeur récursif

entrée: `graphe`, `sommet`, `parcours` et `explore`
résultat: `parcours`, liste des sommets explorés
dfs_recursif(`graphe`, `sommet`, `parcours`, `explore`)
si non `explore[sommet]` **alors**
    ajouter `sommet` à `parcours`
    `explore[sommet] = Vrai`
fin si
pour `s` voisin de `sommet` **faire**
    si non `explore[s]` **alors**
        ajouter `s` à `attente`
    fin si
fin pour
pour `s` dans `attente` **faire**
    dfs_recursif(graphe, s, parcours, explore)
fin pour
renvoi: `parcours`

Version récursive



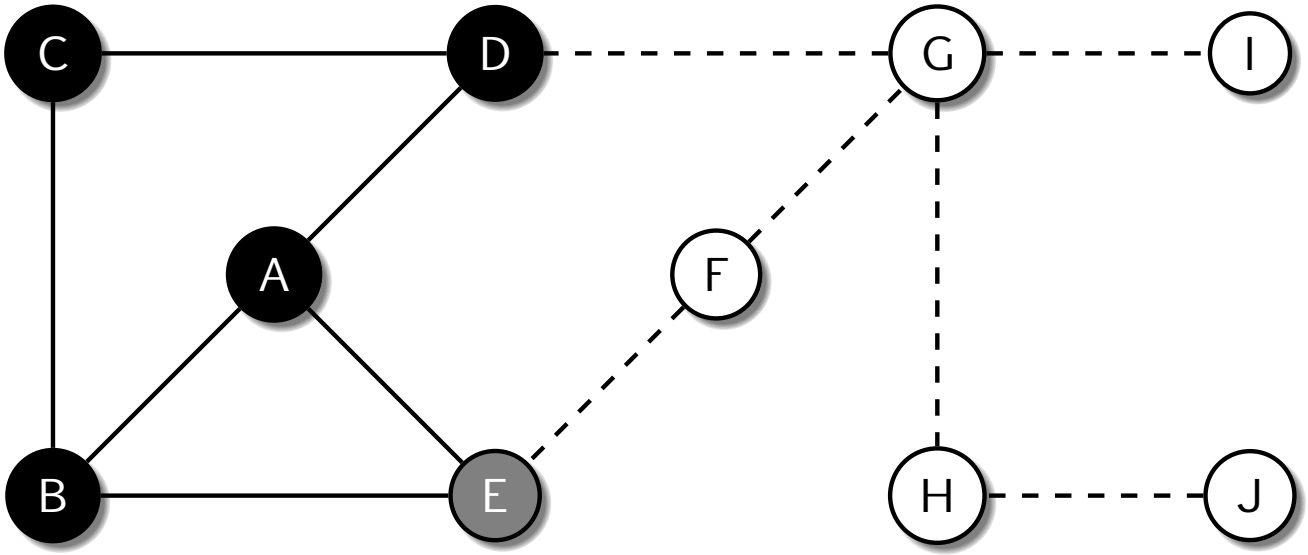
parcours : A B C

Attente issue de A : D E

Attente issue de B : E

Attente issue de C : D

Version récursive



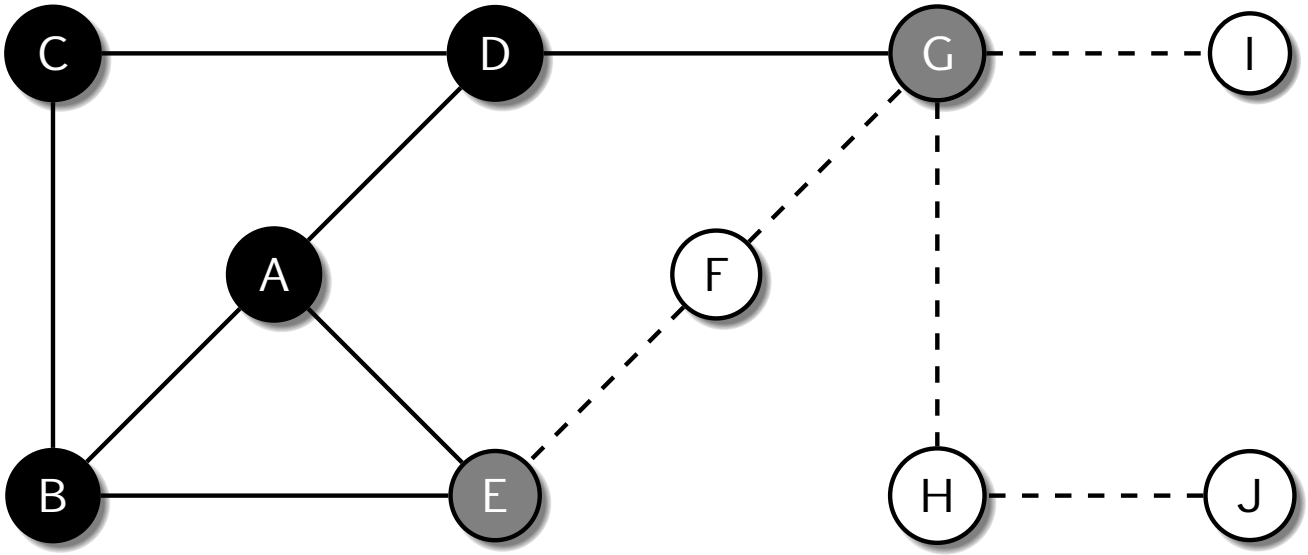
parcours : A B C D

Attente issue de A : D E

Attente issue de B : E

Attente issue de D :

Version récursive



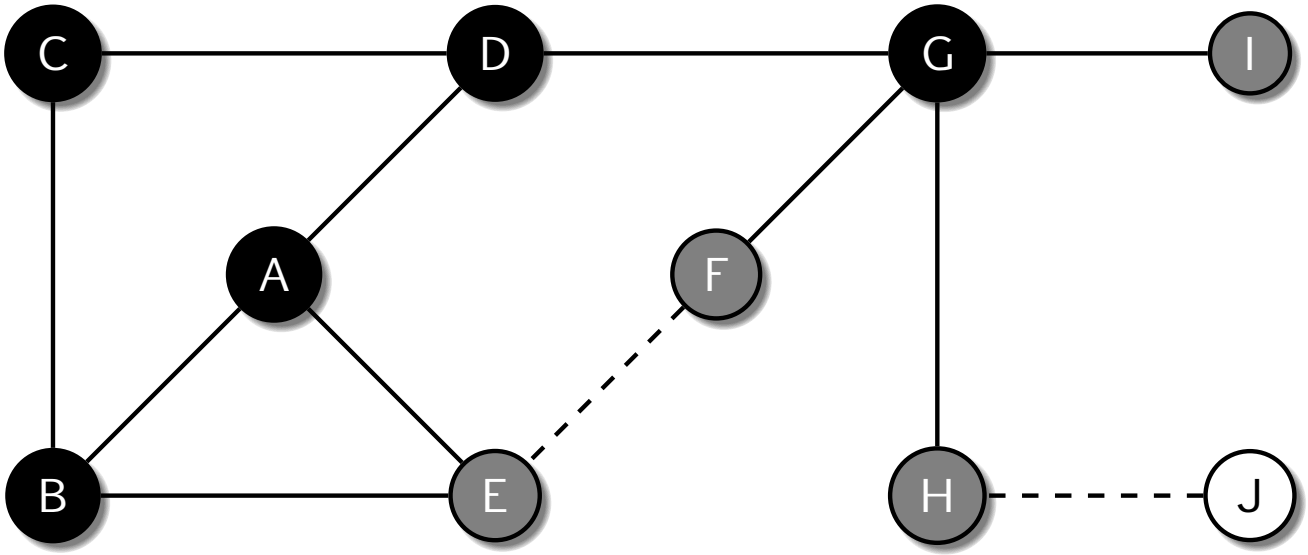
parcours : A B C D

Attente issue de A : D E

Attente issue de B : E

Attente issue de D : G

Version récursive



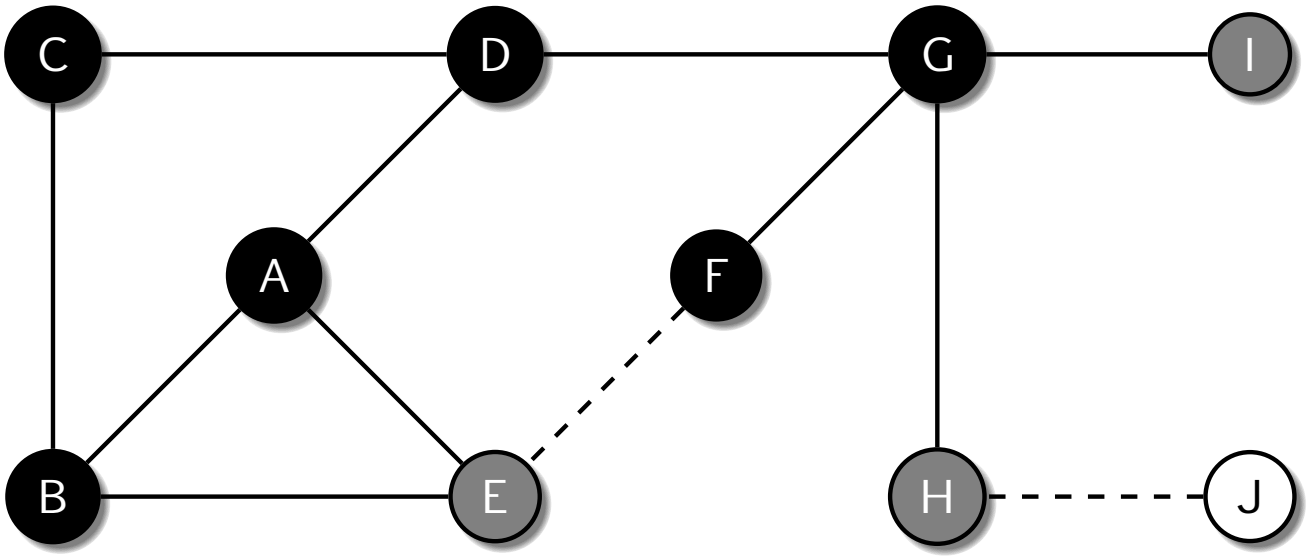
parcours : A B C D G

Attente issue de A : D E

Attente issue de B : E

Attente issue de G : F H I

Version récursive



parcours : A B C D G F

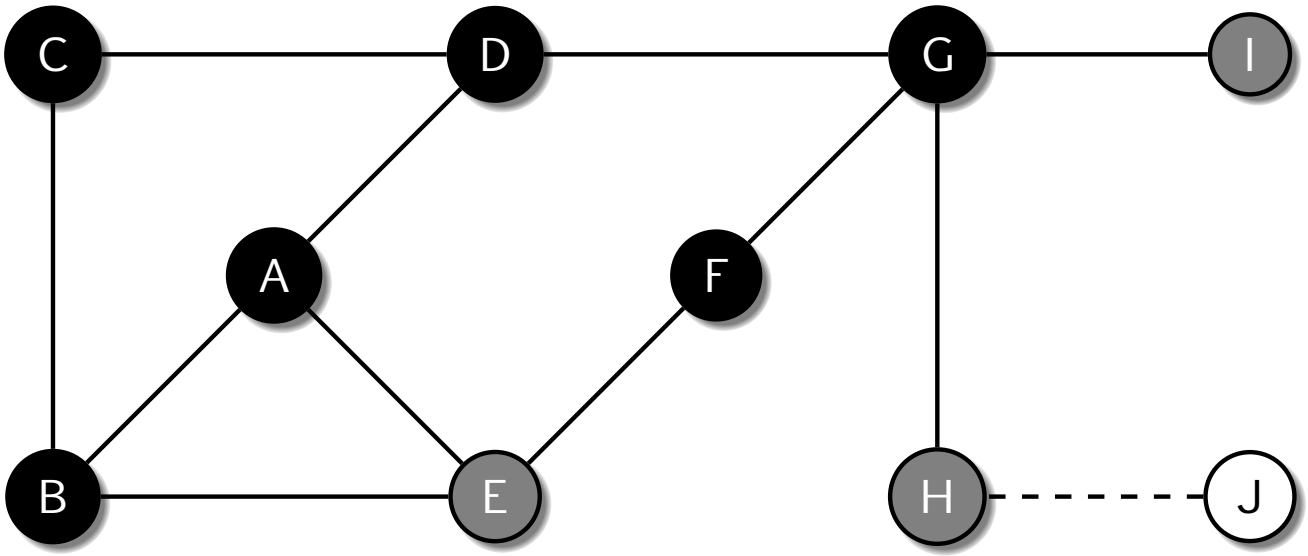
Attente issue de A : D E

Attente issue de B : E

Attente issue de G : H I

Attente issue de F :

Version récursive



parcours : A B C D G F

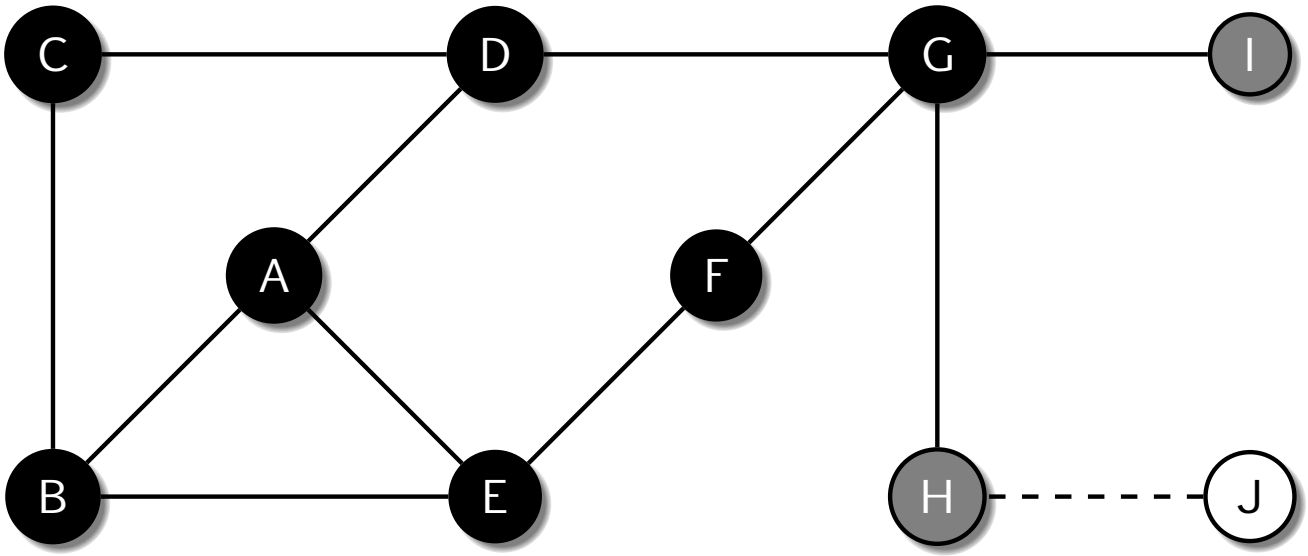
Attente issue de A : D E

Attente issue de B : E

Attente issue de G : H I

Attente issue de F : E

Version récursive



parcours : A B C D G F E

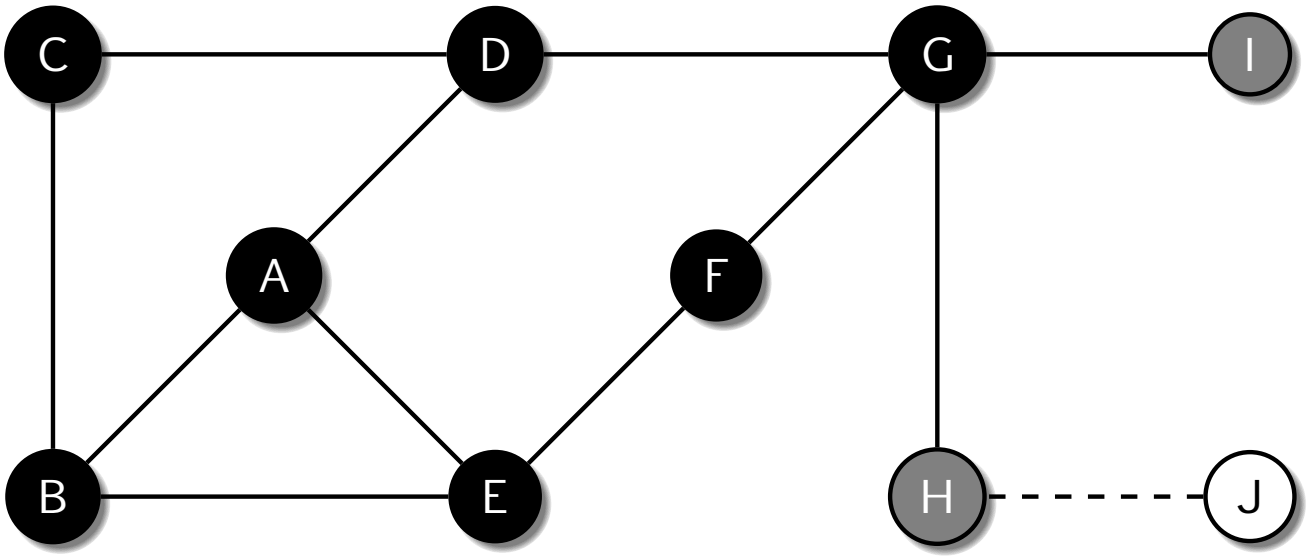
Attente issue de A : D E

Attente issue de B : E

Attente issue de G : H I

Attente issue de E :

Version récursive



parcours : A B C D G F E

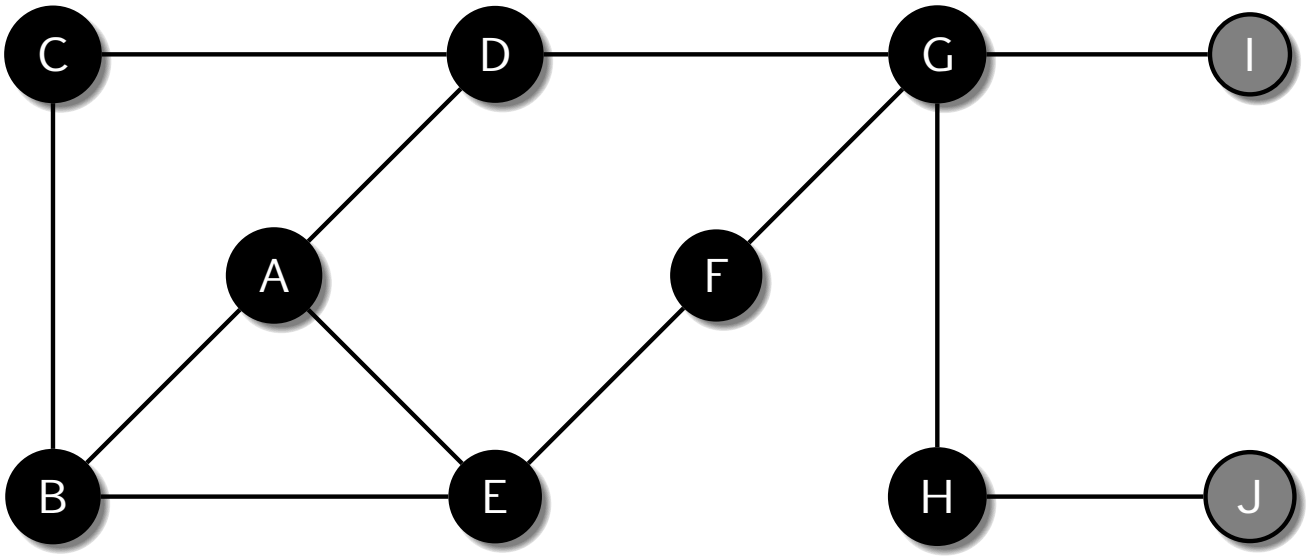
Attente issue de A : D E

Attente issue de B : E

Attente issue de G : H I

Attente issue de E :

Version récursive



parcours : A B C D G F E H

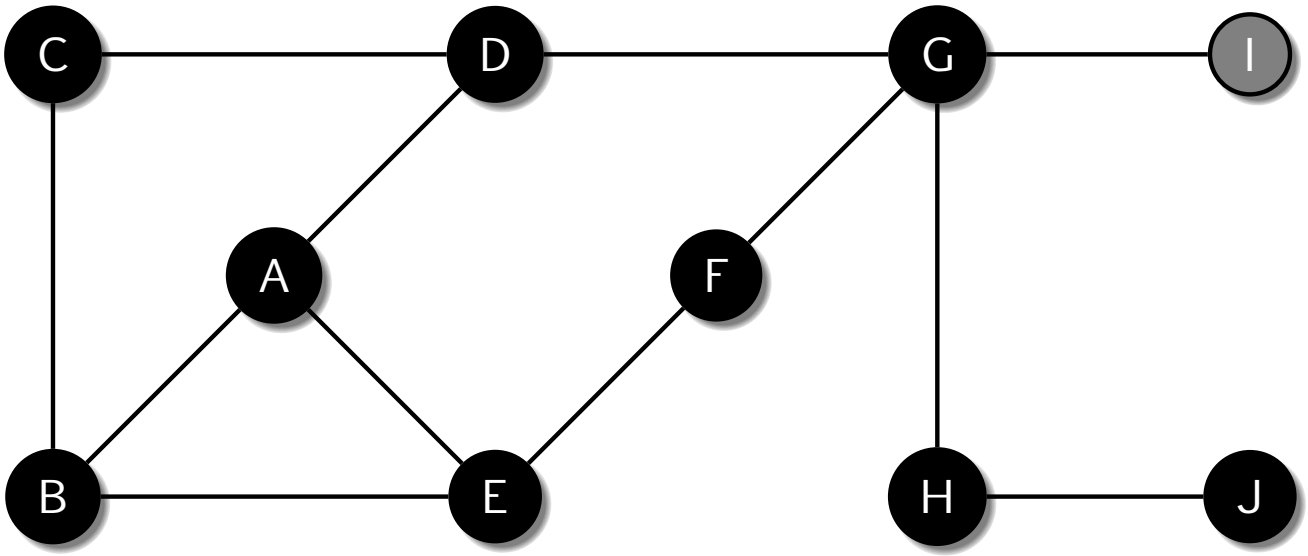
Attente issue de A : D E

Attente issue de B : E

Attente issue de G : I

Attente issue de H : J

Version récursive



parcours : A B C D G F E H J

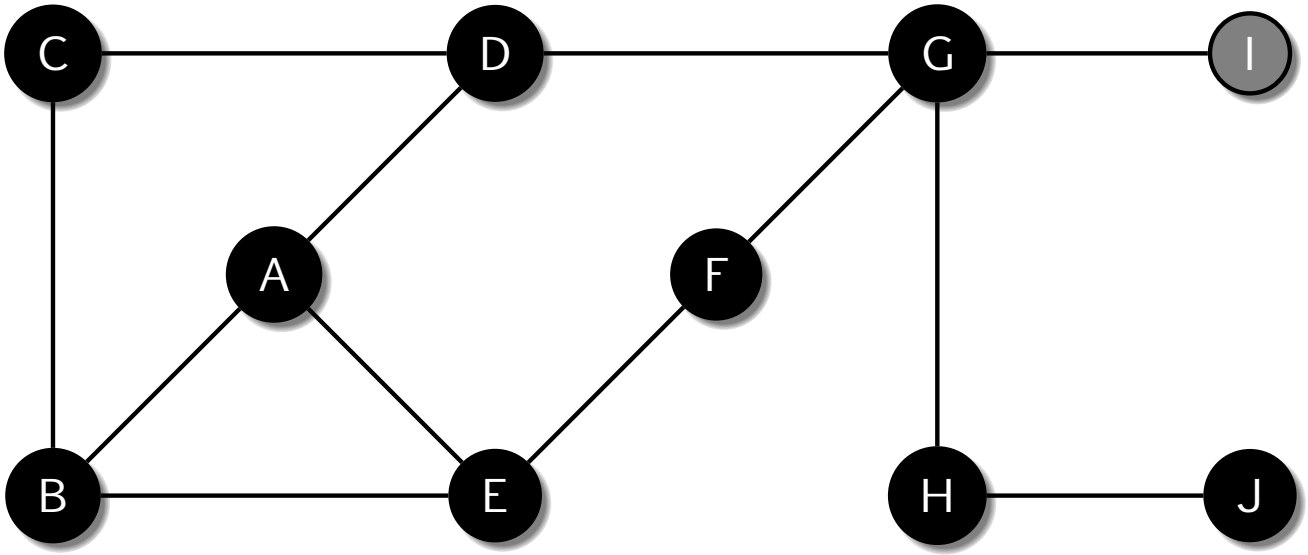
Attente issue de A : D E

Attente issue de B : E

Attente issue de G : I

Attente issue de J :

Version récursive



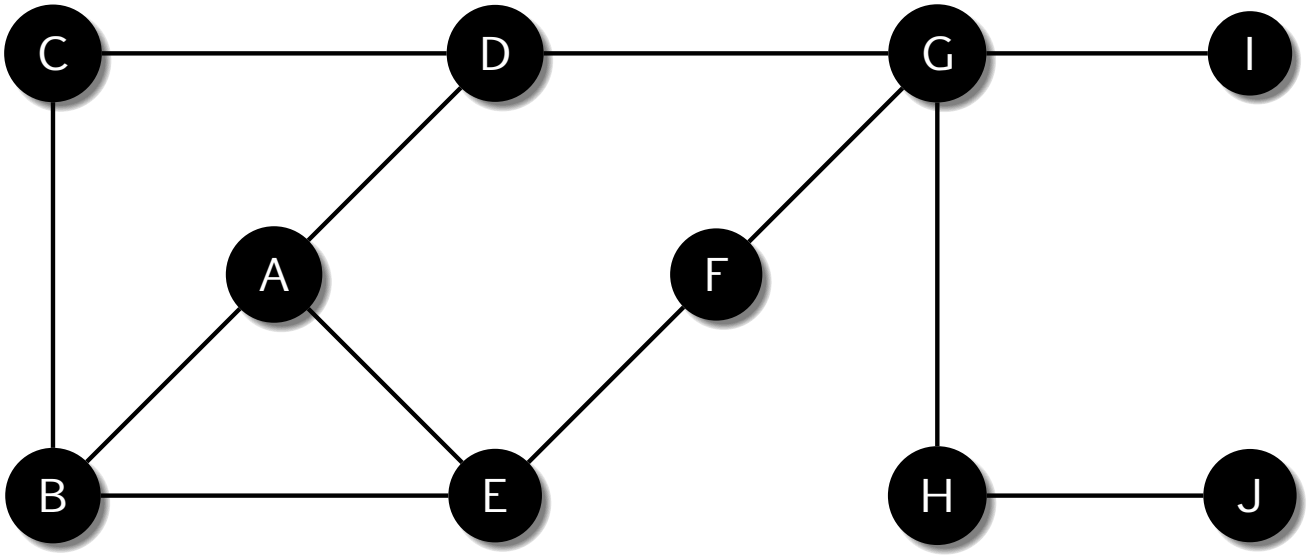
parcours : A B C D G F E H J

Attente issue de A : D E

Attente issue de B : E

Attente issue de G : I

Version récursive



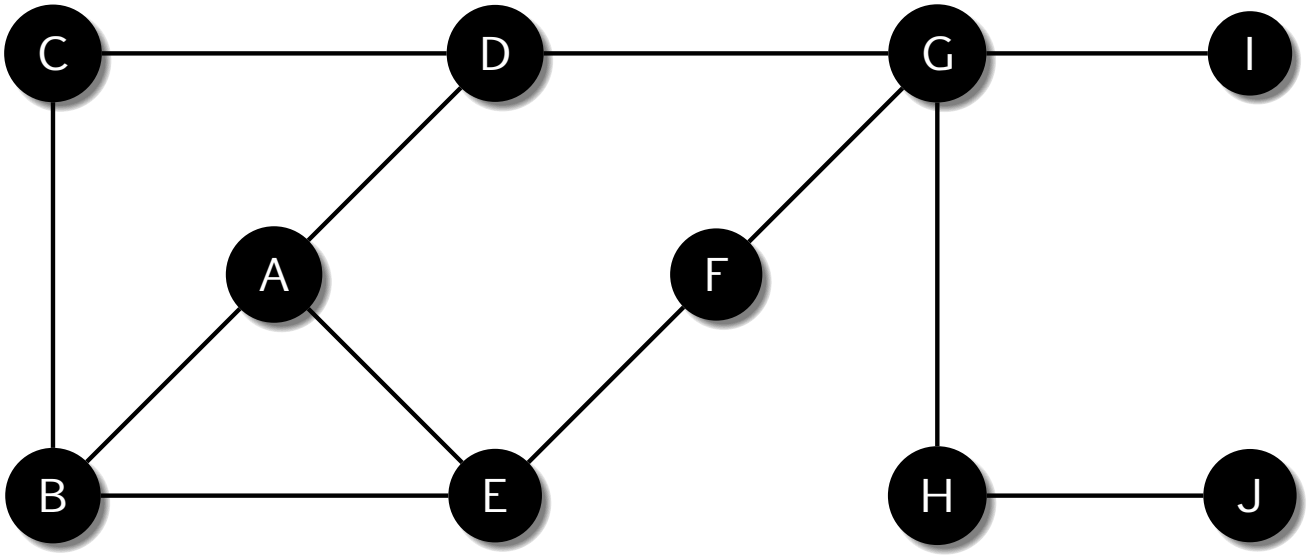
parcours : A B C D G F E H J I

Attente issue de A : D E

Attente issue de B : E

Attente issue de I :

Version récursive

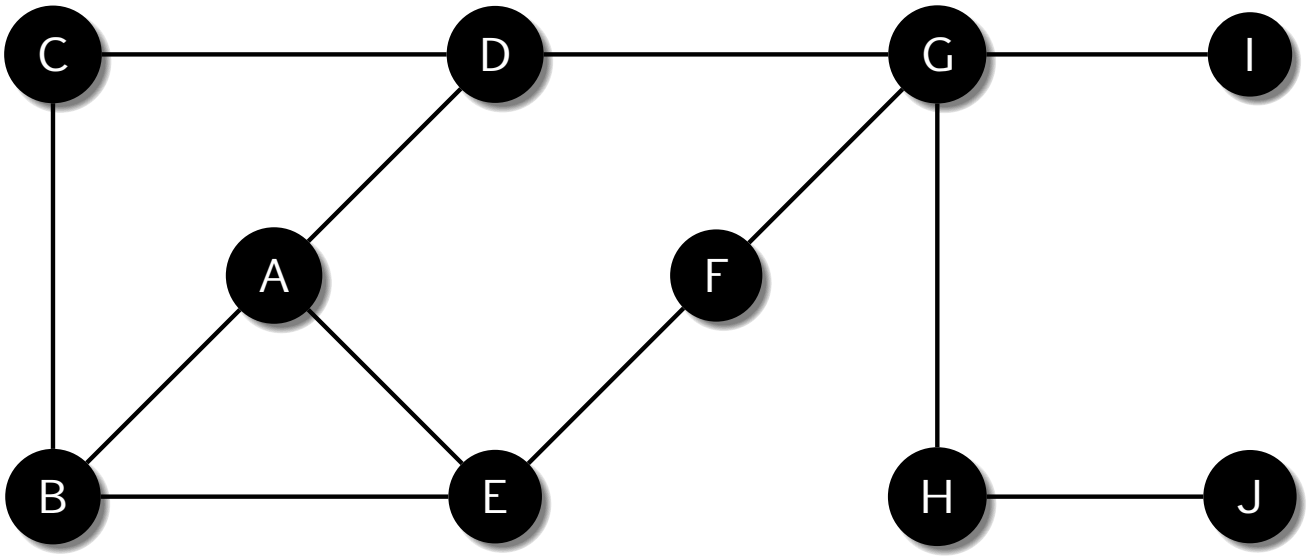


parcours : A B C D G F E H J I

Attente issue de A : D E

Attente issue de B : E

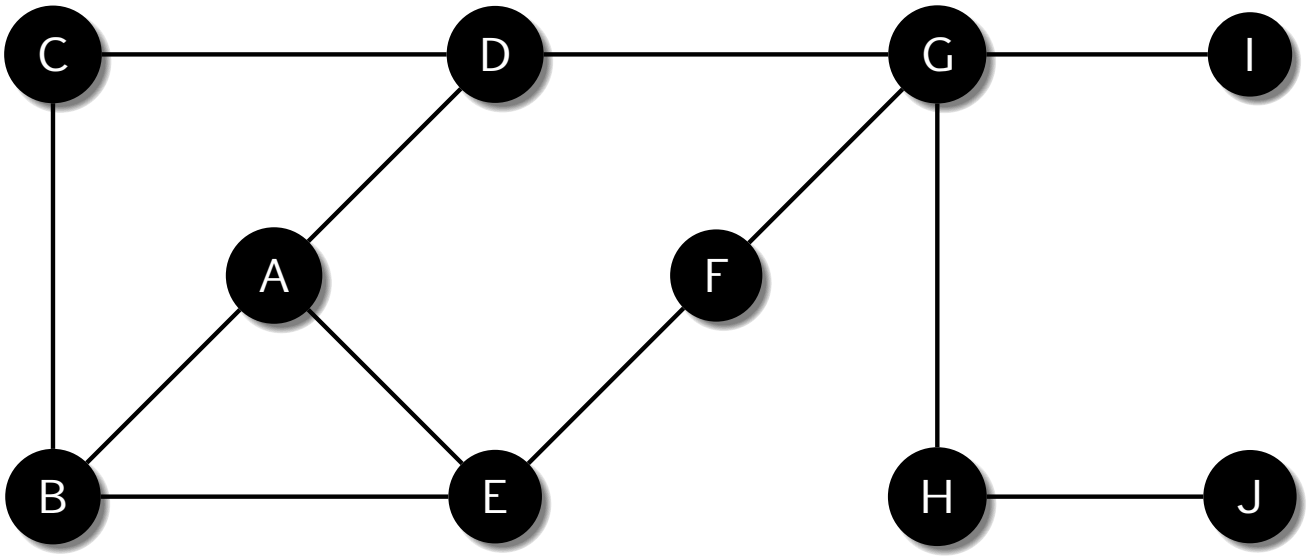
Version récursive



`parcours :` A B C D G F E H J I

`Attente issue de A :` D E

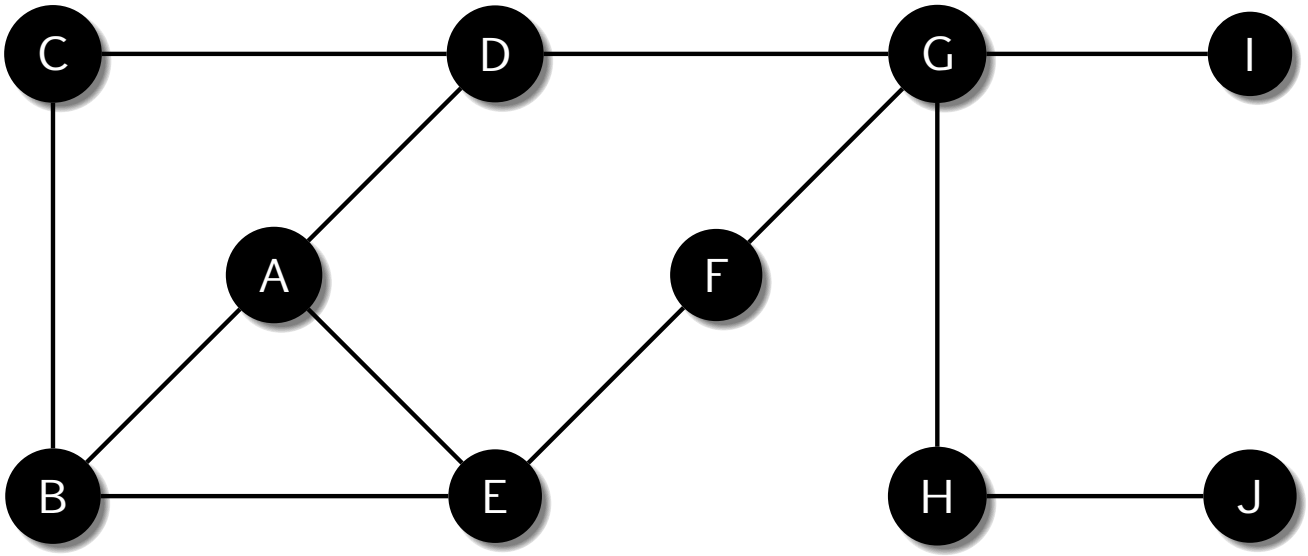
Version récursive



`parcours` : A B C D G F E H J I

`Attente issue de A` : E

Version récursive



`parcours` : A B C D G F E H J I

Version récursive

Pour le graphe support de l'étude, la transcription de ALGO 1 en langage **Python** renvoie :

```
['A', 'B', 'C', 'D', 'G', 'F', 'E', 'H', 'J', 'I']
```

Version itérative

La version itérative adopte les mêmes variables que pour la version récursive. La liste `attente` prend l'allure d'une pile qu'on empile à chaque découverte de sommet. L'exploration d'un sommet dépile la pile.

- Les sommets explorés sont stockés dans une liste `parcours`.
- Une pile `attente` stocke tous les sommets découverts mais non explorés.
- Pour éviter de chercher dans la liste `parcours` si un sommet a déjà été exploré, le dictionnaire `explore` a pour clés les noms des sommets et pour valeur `True` si le sommet a été exploré, `False` sinon.
- La fonction `dfs_itératif` prend en argument :
  - `graphe`, un graphe sous la forme d'un dictionnaire (clé = nom d'un sommet, valeur = noms des voisins du sommet clé)
  - `sommet`, le nom du sommet de départ

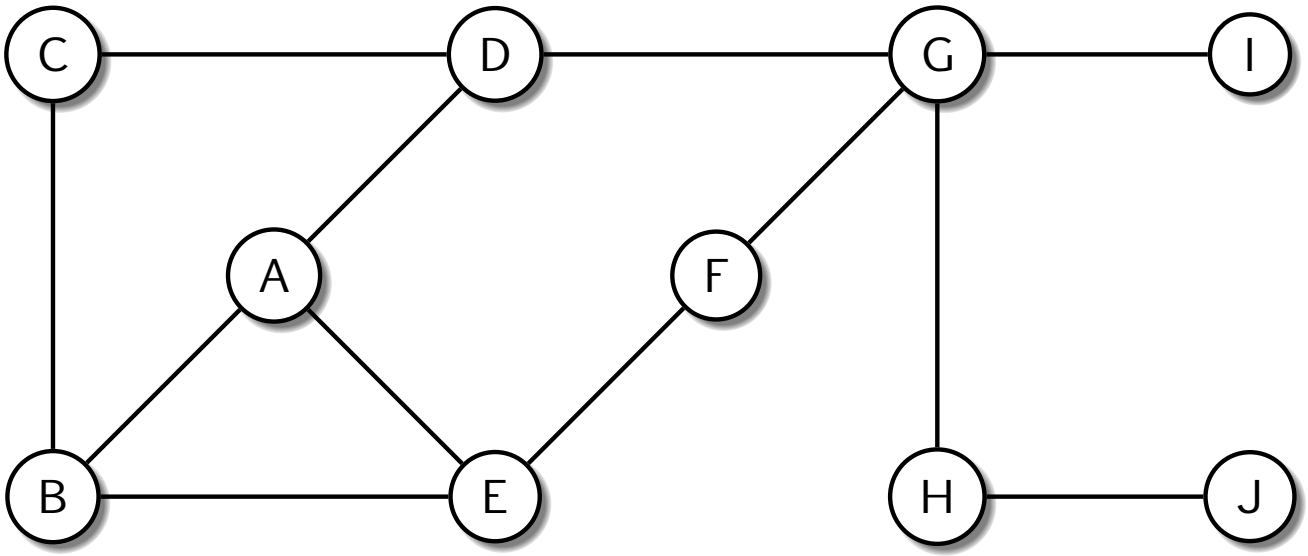
Version itérative

Algorithm 2 Parcours en profondeur itératif

entrée: `graphe`, `sommet`
résultat: `parcours`, liste des sommets explorés
dfs_ieratif(`graphe`, `sommet`)

```
1:  parcours, explore = [], {}
2:  pour s ∈ graphe faire
3:      explore[s] = Faux
4:  fin pour
5:  attente = [sommet]
6:  tant que taille(attente) > 0 faire
7:      sommet = depiler(attente)
8:      si non explore[sommet] alors
9:          ajouter sommet à parcours
10:         explore[sommet] = Vrai
11:         pour s ∈ graphe[sommet] faire
12:             si non explore[s] alors
13:                 ajouter s à attente
14:             fin si
15:         fin pour
16:     fin si
17: fin tant que
18: renvoi: parcours
```

Version itérative

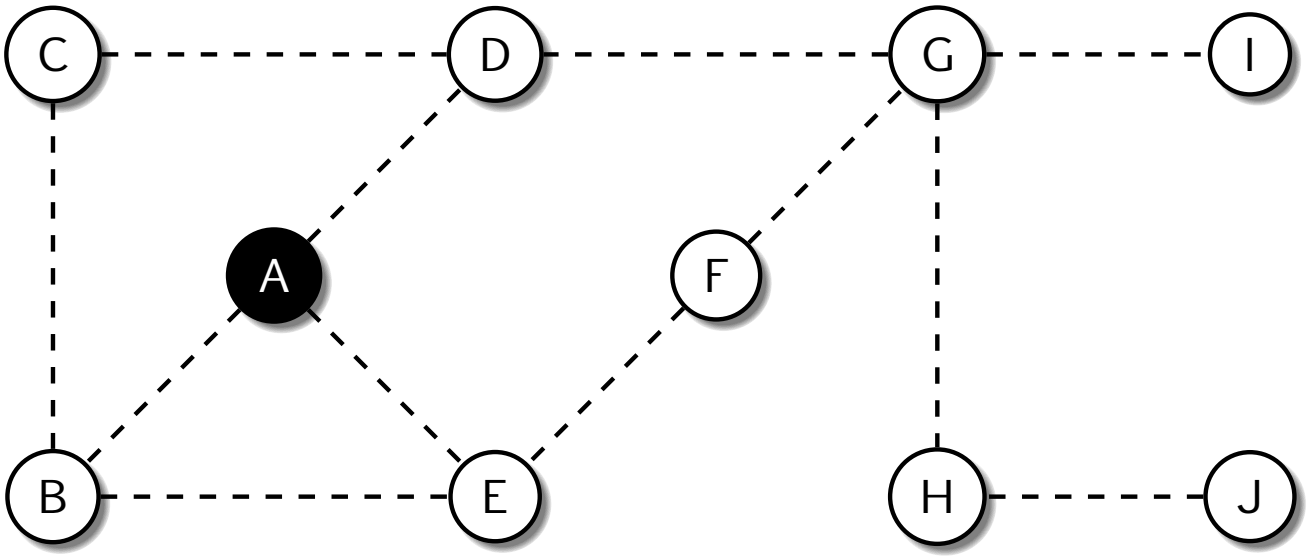



```
parcours :
  attente :
```

Version itérative

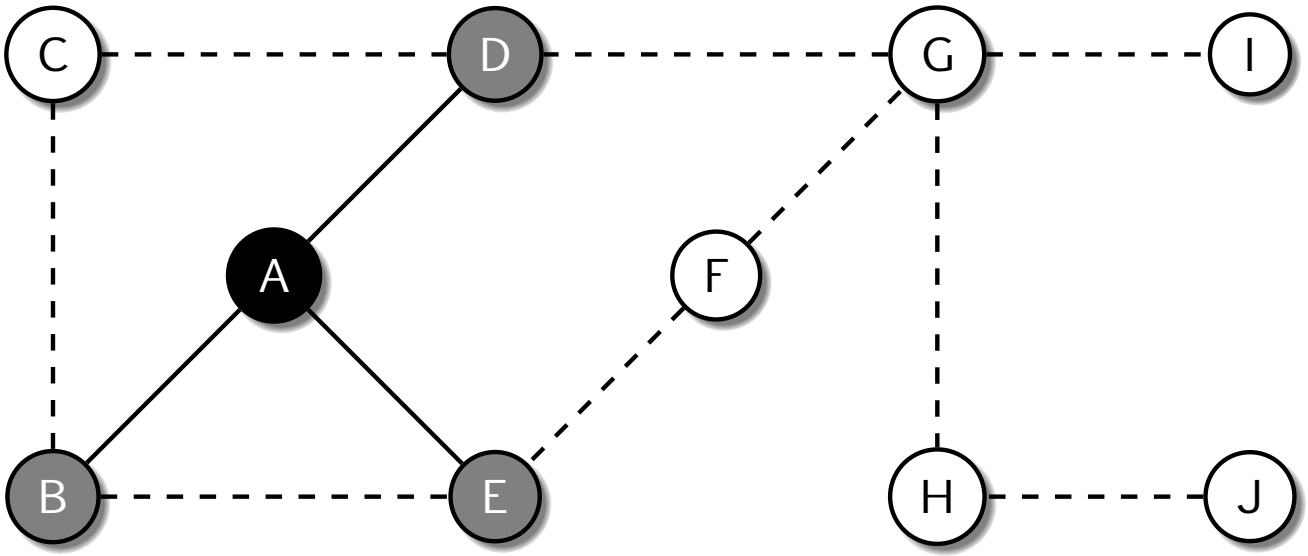
entrée: `graphe`, `sommet`
résultat: `parcours`, liste des sommets explorés
dfs_ieratif(`graphe`, `sommet`)
`parcours`, `explore` = `[]`, {}
pour `s` ∈ `graphe` **faire**
    `explore[s]` = Faux
fin pour
`attente = [sommet]`
tant que taille(`attente`) > 0 **faire**
    `sommet` = depiler(`attente`)
    si non `explore[sommet]` **alors**
        ajouter `sommet` à `parcours`
        `explore[sommet] = Vrai`
        pour `s` ∈ `graphe[sommet]` **faire**
            si non `explore[s]` **alors**
                ajouter `s` à `attente`
            fin si
        fin pour
    fin si
fin tant que
renvoi: `parcours`

Version itérative




```
parcours : A
 attente :
```

Version itérative



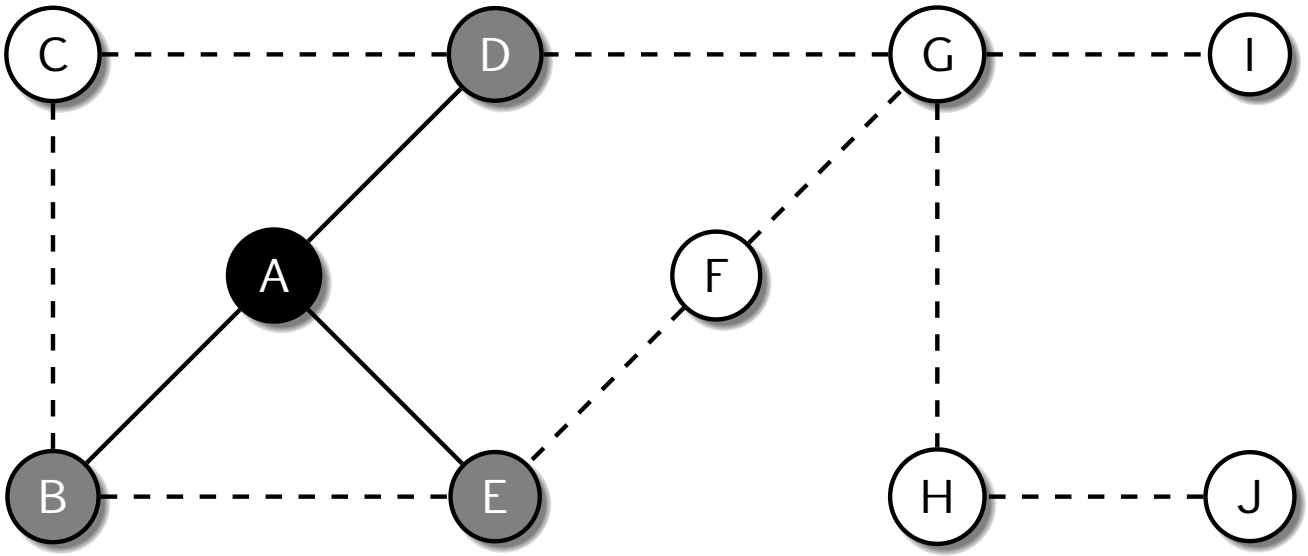
`parcours` : A

`attente` : B D E

Version itérative

entrée: `graphe`, `sommet`
résultat: `parcours`, liste des sommets explorés
dfs_ieratif(`graphe`, `sommet`)
`parcours`, `explore` = `[]`, {}
pour `s` ∈ `graphe` **faire**
    `explore[s]` = Faux
fin pour
`attente = [sommet]`
tant que taille(`attente`) > 0 **faire**
    `sommet` = depiler(`attente`)
    si non `explore[sommet]` **alors**
        ajouter `sommet` à `parcours`
        `explore[sommet] = Vrai`
        pour `s` ∈ `graphe[sommet]` **faire**
            si non `explore[s]` **alors**
                ajouter `s` à `attente`
            fin si
        fin pour
    fin si
fin tant que
renvoi: `parcours`

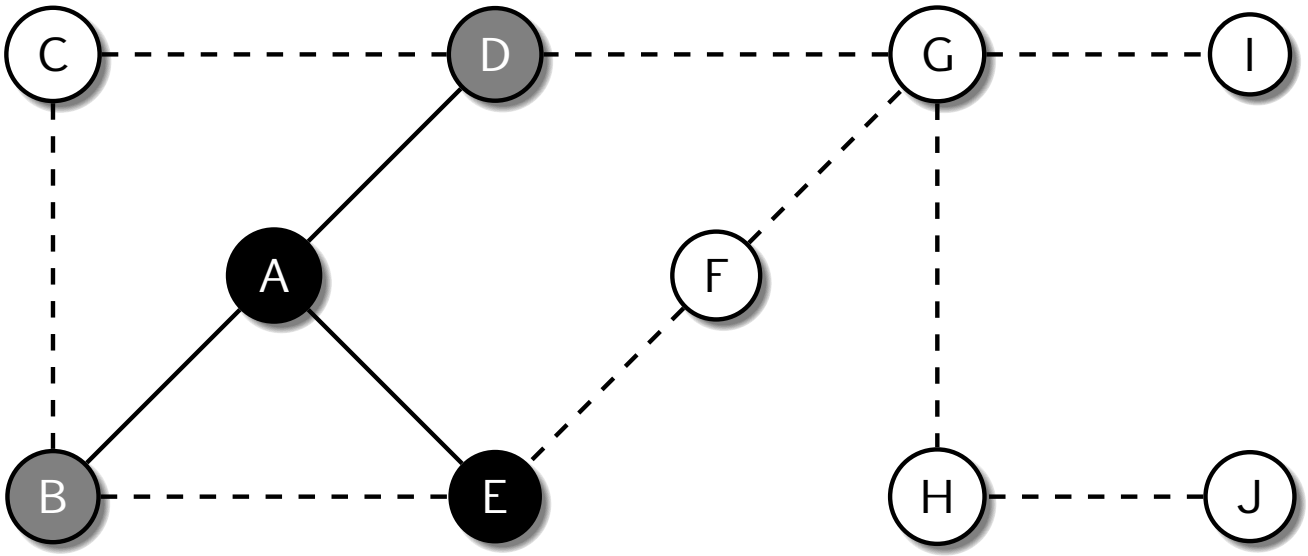
Version itérative



`parcours` : A

`attente` : B D E

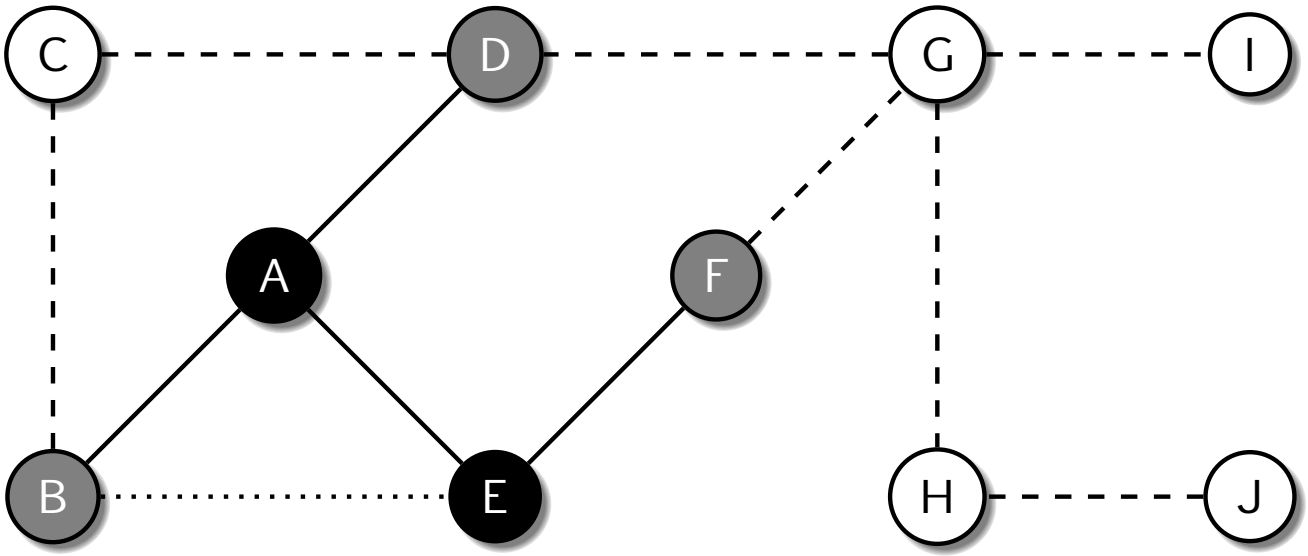
Version itérative



`parcours` : A E

`attente` : B D

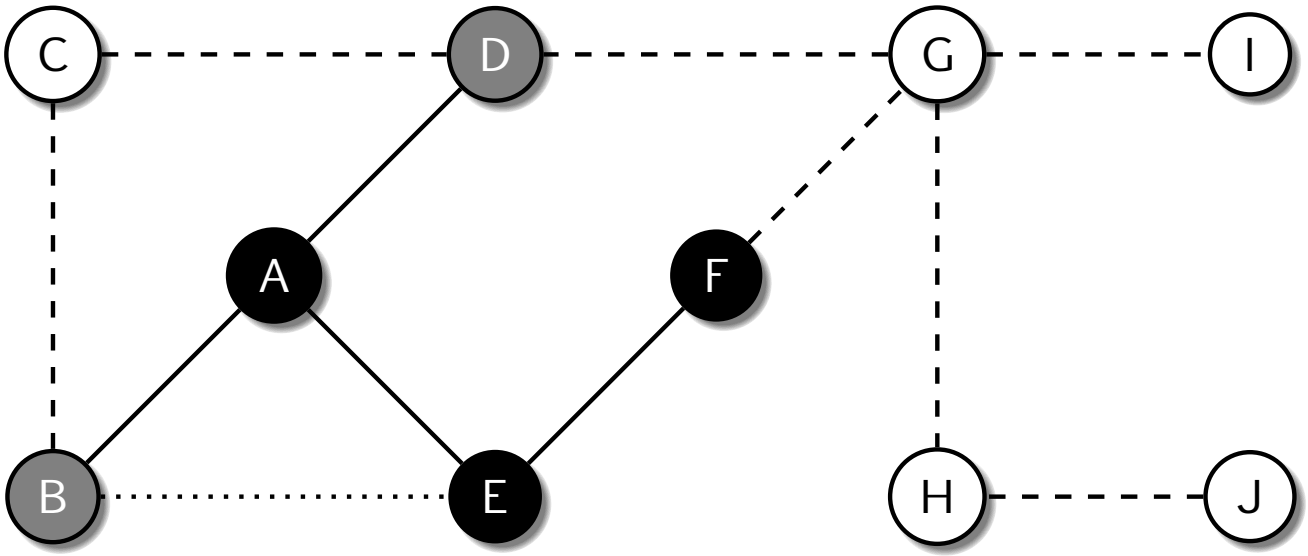
Version itérative



`parcours` : A E

`attente` : B D F

Version itérative



parcours : A E F

attente : B D

Version itérative

Pour le graphe support de l'étude, la transcription de Algo 2 en langage **Python** renvoie :

```
['A', 'E', 'F', 'G', 'I', 'H', 'J', 'D', 'C', 'B']
```

Remarques

On constate que les résultats donnés par l'Algo 1 et par l'Algo 2 sont différents. En effet, la version récursive va choisir le premier voisin de `A`, donc `B`. La version itérative emplie les voisins de `A` qui sont, dans l'ordre de stockage du graphe, `B`, `D` puis `E`. En dépilant, pour choisir le sommet après `A`, on tombe donc sur `E`.

Dans les deux cas, il s'agit bien de parcours en profondeur car on s'éloigne du sommet initial à chaque sommet exploré.

Si en partant d'un sommet s_0, il est possible d'atteindre tous les sommets du graphe, alors le graphe est connexe. Sinon, on appelle classe de connexité de s_0, l'ensemble des sommets atteints.

Recherche d'un plus court chemin

En stockant pour chaque sommet, son prédécesseur (ou père), il est possible d'établir, à rebours, le chemin parcouru pour aller d'un sommet s_0 au sommet s_f.

Pour un graphe non orienté et non pondéré, pour trouver le chemin le plus court entre deux sommets, c'est à dire celui minimisant le nombre de sommets, on utilise une liste `chemin` qui mémorise le chemin en cours et une liste `plus_court` qui mémorise le plus court des chemins étudiés. Il s'agit là encore d'un parcours en profondeur récursif d'un graphe.

Pour le graphe support de l'étude, la transcription de ALGO 3 en langage **Python** renvoie :

```
>>> dfs_pcc(graf, 'A', 'J')
['A', 'D', 'G', 'H', 'J']
>>> dfs_pcc(graf, 'F', 'I')
['F', 'G', 'I']
```

Recherche d'un plus court chemin

Algorithm 3 Recherche d'un plus court chemin

entrée: `graphe, depart, arrivee, chemin = [], plus_court = None`
résultat: `plus_court` le chemin le plus court entre `depart` et `arrivee`
dfs_pcc(`graphe, depart, arrivee, chemin = [], plus_court = None`)

```
1:  on_avance = chemin + [depart]
2:  si depart = arrivee alors
3:      renvoi: on_avance
4:  fin si
5:  pour s ∈ graphe[depart] faire
6:      si s ∉ on_avance alors
7:          si plus_court = None ou taille(on_avance) < taille(plus_court) alors
8:              nouveau = fs_pcc(graphe, s, arrivee, on_avance, plus_court)
9:              si nouveau n'est pas None alors
10:                 plus_court = nouveau
11:             fin si
12:         fin si
13:     fin si
14: fin pour
15: renvoi: plus_court
```

Version itérative

Principe :

> *A partir du sommet de départ, on explore tous ses sommets voisins immédiats, puis tous les voisins immédiats non exploré des voisins immédiats et ainsi de suite jusqu'à ce qu'il n'y ait plus de sommet non exploré.*

Dans l'Algo 2 la profondeur de parcours était obtenue en prenant le sommet de la pile `attente` pour choisir le prochain sommet à explorer. Or en prenant les éléments en début de la liste `attente`, on retrouve les premiers voisins du sommet de départ.

Une file `attente` permet donc un parcours en largeur (ou bds `breadth first search`) du graphe quant une pile `attente` offre un parcours en profondeur (dfs).

Pour le graphe support de l'étude, la transcription de Algo 4 en langage **Python** renvoie :

```
['A', 'B', 'D', 'E', 'C',
       'G', 'F', 'H', 'I', 'J']
```

Version itérative

Algorithm 4 Parcours en largeur itératif

entrée: `graphe`, `sommet`
résultat: `parcours`, liste des sommets explorés
dfs_ieratif(`graphe`, `sommet`)

```
1:  parcours, explore = [], {}
2:  pour s ∈ graphe faire
3:      explore[s] = Faux
4:  fin pour
5:  attente = [sommet]
6:  tant que taille(attente) > 0 faire
7:      sommet = defiler(attente)
8:      si non explore[sommet] alors
9:          ajouter sommet à parcours
10:         explore[sommet] = Vrai
11:         pour s ∈ graphe[sommet] faire
12:             si non explore[s] alors
13:                 ajouter s à attente
14:             fin si
15:         fin pour
16:     fin si
17: fin tant que
18: renvoi: parcours
```

Recherche d'un cycle

Pour rechercher si un graphe présente un cycle, on peut adopter une stratégie de parcours en largeur. Si en explorant un voisin s_j d'un sommet s_i on constate qu'il a déjà été exploré mais qu'il n'est pas le prédécesseur de s_j, c'est qu'on est en présence d'un cycle.

On utilise donc un parcours en largeur en attribuant à chaque sommet parcouru sa distance (en nombre d'arêtes) par rapport au sommet initial s_0.

Recherche d'un cycle

- On attribue à chaque sommet une distance infinie via un dictionnaire :

```
distance = {s : float('inf') for s in graphe}
```

- On attribue une distance de 0 au sommet de départ qu'on place dans une file `file`
- Tant que la file n'est pas vide
  - On appelle `sommet` l'élément qu'on défile de `file`
  - On explore les voisins de `sommet` et pour chaque voisin :
    - ★ s'il est à une distance infinie, on lui donne pour distance celle de `sommet` additionnée de 1
    - ★ si son niveau est supérieur ou égal à celui de `sommet`, c'est qu'on a trouvé un cycle et on renvoie `True`
  - Si la file devient vide, c'est qu'il n'y a pas de cycle et on renvoie `False`

Recherche d'un cycle

Algorithm 5 Recherche d'un cycle

entrée: `graphe`, `sommet`
résultat: `True` si le sous graphe connexe auquel appartient `sommet` possède un cycle, `False` sinon
bfs_cycle(`graphe`, `sommet`)

```
1:  niveau = {s : float('inf') for s in graphe}
2:  niveau[sommet], file = 0, [sommet]
3:  tant que taille(file) > 0 faire
4:      sommet = defile(file)
5:      pour s ∈ graphe[sommet] faire
6:          si niveaux[s] est ∞ alors
7:              niveaux[s] = niveaux[sommet] + 1
8:              file.enfile(s)
9:          sinon si niveaux[s] >= niveaux[sommet] alors
10:             renvoi: True
11:         fin si
12:     fin pour
13: fin tant que
14: renvoi: False
```

Sommaire

Algorithme de Dijkstra et algorithme A*

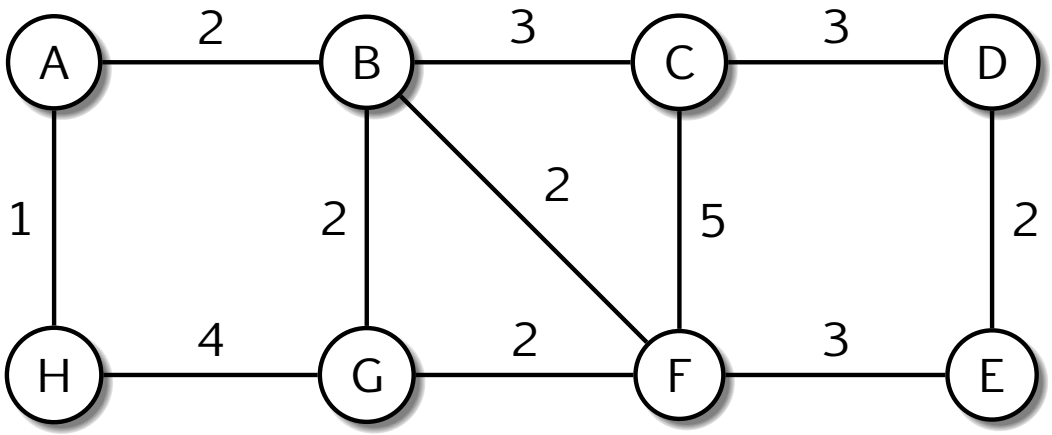
Dans les parties précédentes, le parcours des graphes ne tenait pas compte du poids associé aux arrêtes.

Si le temps entre deux stations de métro consécutives est sensiblement le même quelques soient ces stations, pour la recherche d'un plus court chemin sur une carte routière, un poids lié à la distance ou lié au temps de parcours peut donner des résultats différents. La Beauce et la Haute Savoie ne se parcours pas de la même manière !

Dans cette partie, on s'intéresse à des graphes connexes non orientés à poids positifs.

Algorithme de Dijkstra et algorithme A*

Pour les exemples, le graphe suivant sera utilisé.



Algorithme de Dijkstra

Principe

PRINCIPE :

> *Tant que le sommet d'arrivé n'est pas découvert, on explore les voisins non explorés du sommet, le plus proche du sommet initial, dont il existe au moins un voisin non exploré.*

L'algorithme Dijkstra s'apparente à un algorithme de parcours en largeur mais où la file d'attente est remplacée par une file de priorité (le plus proche). On peut stopper l'algorithme dès que le sommet d'arrivé est trouvé ou bien le laisser parcourir le graphe pour déterminer la distance de chacun des sommets au sommet initial.

Détail des variables

Pour l'implémentation de l'algorithme, on utilise :

- un graphe `graphe`, dictionnaire de dictionnaires. La clé du dictionnaire est un sommet s et sa valeur un dictionnaire dont les clés sont les sommets voisins de s et les valeurs leur distance à s.
- `depart`, le sommet de départ
- `arrivee`, le sommet d'arrivé si on cherche un plus court chemin jusqu'à un sommet
- `d_ini` un dictionnaire dont les clés sont les sommets du graphe et les valeurs leur distance initiale au sommet de départ (au début tous les sommets sont à une distance infinie `float('inf')`
- `d_fin` un dictionnaire dont les clés sont les sommets du graphe et les valeurs leur distance réel au sommet de départ
- `pere`, si on cherche le plus court chemin, un dictionnaire dont les clés sont les sommets du graphe et les valeurs le prédécesseur du sommet

Détail de la stratégie

La structure de l'algorithme de Dijkstra est :

- initialisation de `d_fin` (dictionnaire vide) et `d_ini` (tous les sommets sont à l'infini sauf le sommet `depart`
- tant qu'il reste des sommets non exploré (ou que le sommet d'arrivé n'a pas été trouvé)
  - on sélectionne le sommet `s_k` le plus proche parmi les sommets contenus dans `d_ini`.
  - pour chacun des des sommets voisins `s_v` de `s_k`, on regarde s'il n'est pas dans `d_fin`. Auquel cas, on affecte à `d_ini[s_k]` le minimum entre sa valeur actuelle et la valeur qu'il aurait si son prédécesseur était `s_k`. Cette étape permet de mettre à jour les distances des voisins du sommet visité pour la prochaine étape et déterminer qui est le père.
- on ajoute à `d_fin`, le sommet `s_k` avec la distance `d_ini[s_k]`
- on supprime la clé `s_k` pour ne pas la choisir à nouveau.

Remarque : cet algorithme donne la distance de tous les sommets du graphe au sommet départ. Cependant, il ne donne pas le chemin. Avec le dictionnaire `pere`, il est possible de retrouver le chemin en partant du sommet `arrivee` et en remontant la liste des prédécesseurs.

Pseudocodes

Pour sélectionner le sommet `s_k` le plus proche parmi les sommets contenus dans `d_ini`, on a besoin d'un algorithme Algo 6 qui permet de trouver la clé dont la valeur est la plus petite.

L'Algo 7 donne, en pseudocode, une version de l'algorithme permettant de déterminer la distance de tous les sommets au sommet départ.

Algorithm 6 Minimum dans un dictionnaire

entrée: `dic` un dictionnaire
résultat: `c`, la clé dont la valeur est la plus faible des valeurs de `dic`
mini_dic(`dic`)

```
1: val = float('inf')
2: pour k dans dic faire
3:     si dic[k] < val alors
4:         c, val = k, dic[k]
5:     fin si
6: fin pour
7: renvoi: c
```

Algorithm 7 Algorithme de Dijkstra

```
    entrée: graphe, sommet
    résultat: d_fin, pere
    dijkstra(graphe, sommet)
 1: d_ini = {s : float('inf') for s in graphe}
 2: d_ini[sommet] = 0
 3: d_fin, pere = {}, {}
 4: tant que taille(d_ini) >0 faire
 5:     s_k = mini_dic(d_ini)
 6:     pour s_v dans graphe[s_k] faire
 7:         w = graphe[s_k][s_v]
 8:         si s_v n'est pas dans d_fin et d_ini[s_v] > d_ini[s_k] + w alors
 9:             d_ini[s_v] = d_ini[s_k] + w
10:             pere[s_v] = s_k
11:         fin si
12:     fin pour
13:     d_fin[s_k] = d_ini[s_k]
14:     supprime(d_ini[s_k])
15: fin tant que
16: renvoi: d_fin, pere
```

Pour déterminer le chemin, l'Algo 8 remonte la *liste* des père depuis le sommet d'arrivée `s_fin` jusqu'au sommet départ `s_dep`.

On utilise alors `pere` le dictionnaire des pères de chaque sommets du graphe explorer depuis `s_dep` et une liste `chemin` des sommets classés par ordre de parcours depuis le sommet de départ jusqu'au sommet d'arrivée.

Partant du sommet d'arrivée, la liste `chemin` doit être inversée avant d'être renvoyée.

Algorithm 8 Obtention du chemin

entrée: `pere`, `depart` et `arrivee`
résultat: `chemin`
trajet(`pere`, `depart`, `arrivee`)
1: `chemin = [arrivee]`
2: **tant que** `arrivee` différent de `depart` **faire**
3: `arrivee = pere[chemin[-1]]`
4: ajouter `arrivee` à `chemin`
5: **fin tant que**
6: inverser `chemin`
7: **renvoi:** `chemin`

Si on souhaite le plus court chemin entre les sommets `depart` et `arrivee` sans calculer les distances de tous les sommets du graphe au sommet de départ, on peut utiliser l'Algo 9 :

Algorithm 9 Plus court chemin entre deux sommets

entrée: `graphe`, `depart` et `arrivee`
résultat: `chemin`
dijkstra_pcc(`graphe`, `depart`, `arrivee`)

```
1:  d_ini = {s : float('inf') for s in graphe}
2:  d_ini[depart] = 0
3:  d_fin, pere = {}, {}
4:  tant que taille(d_ini) >0 et arrivee n'est pas dans d_fin faire
5:      s_k = mini_dic(d_ini)
6:      pour s_v dans graphe[s_k] faire
7:          w = graphe[s_k][s_v]
8:          si s_v n'est pas dans d_fin et d_ini[s_v] > d_ini[s_k] + w alors
9:              d_ini[s_v] = d_ini[s_k] + w
10:             pere[s_v] = s_k
11:         fin si
12:     fin pour
13:     d_fin[s_k] = d_ini[s_k]
14:     supprime(d[s_k])
15: fin tant que
16: renvoi: trajet(pere, depart, arrivee)
```

Algorithme A*

Heuristique

L'algorithme de Dijkstra permet de trouver le plus court chemin entre deux sommets. Cependant, l'exploration s'apparente à un parcours en largeur, faisant *pousser la recherche sous forme d'un disque croissant centrée sur le sommet de départ*. Or, il est regrettable de chercher des trajets en direction de Lille quand on souhaite faire un Paris - Marseille.

L'algorithme nommé `Algorithme A*`, proposé en 1968 par Peter E. Hart, Nils John Nilson et Bertram Raphael, permet d'améliorer l'algorithme de Dijkstra. En effet, au lieu d'explorer tous les sommets consciencieusement en fonction de leur distance par rapport au sommet, on peut privilégier ceux qui ont *une bonne tête*.

C'est ce qu'on appelle une évaluation heuristique, par exemple basée sur la distance à vol d'oiseau de l'arrivée. Pour un Paris - Marseille, mieux vaut regarder du côté d'Evry que de Chantilly.

Dans le cas des jeux vidéos où le temps de calcul doit être minimal, l'algorithme A* propose une des meilleurs solution, plus rapidement que l'algorithme de Dijkstra.

Pseudocodes

Pour `orienter` la recherche du plus court chemin vers le sommet d'arrivé, on propose ici une évaluation heuristique basée sur la distance à vol d'oiseau d'un sommet vers le sommet recherché.

On suppose qu'on dispose :

- d'un dictionnaire `dico_coords` dont les clés sont les sommets du graphes et les valeurs, leurs coordonnées cartésiennes
- une fonction `heuristique` prend en argument un graphe `graphe` sous forme de dictionnaire de dictionnaires, le sommet arrivée `arrivee` et le dictionnaire des coordonnées `dico_coords`. Cette fonction renvoie un dictionnaire `d_heu` dont les clés sont les sommets du graphe et les valeurs, leur distance à vol d'oiseau au sommet `arrivee`
- l'Algo 6 est modifiée pour ajouter à la distance initiale, la valeur de l'évaluation heuristique, comme le présente l'Algo 10

Pseudocodes

Algorithm 10 Recherche heuristique

entrée: `dist` un dictionnaire des distances initiales et `heu` un dictionnaire des évaluations heuristiques
résultat: `c`, la clé dont la valeur est la meilleure
mini_dic_heu(`dist`, `heu`)

```
1: val = float('inf')
2: pour k dans dic faire
3:     si dic[k] + heu[k] < val alors
4:         c, val = k, dic[k] + heu[k]
5:     fin si
6: fin pour
7: renvoi: c
```

Pseudocodes

Lorsque le sommet d'arrivée `arrivee` est déterminé, on peut construire le dictionnaire des valeurs heuristiques, ici basé sur un calcul de la distance à vol d'oiseau d'un sommet vers le sommet `arrivee`.

L'Algo 11 A* ne diffère donc de l'Algo 9 Dijkstra que dans la file de priorités des sommets à explorer, modifiée par l'évaluation heuristique.
Remarque : tout l'art est de trouver l'évaluation heuristique en fonction du problème. Quelles sont les raisons de privilégier tel sommet par rapport à tel autre.

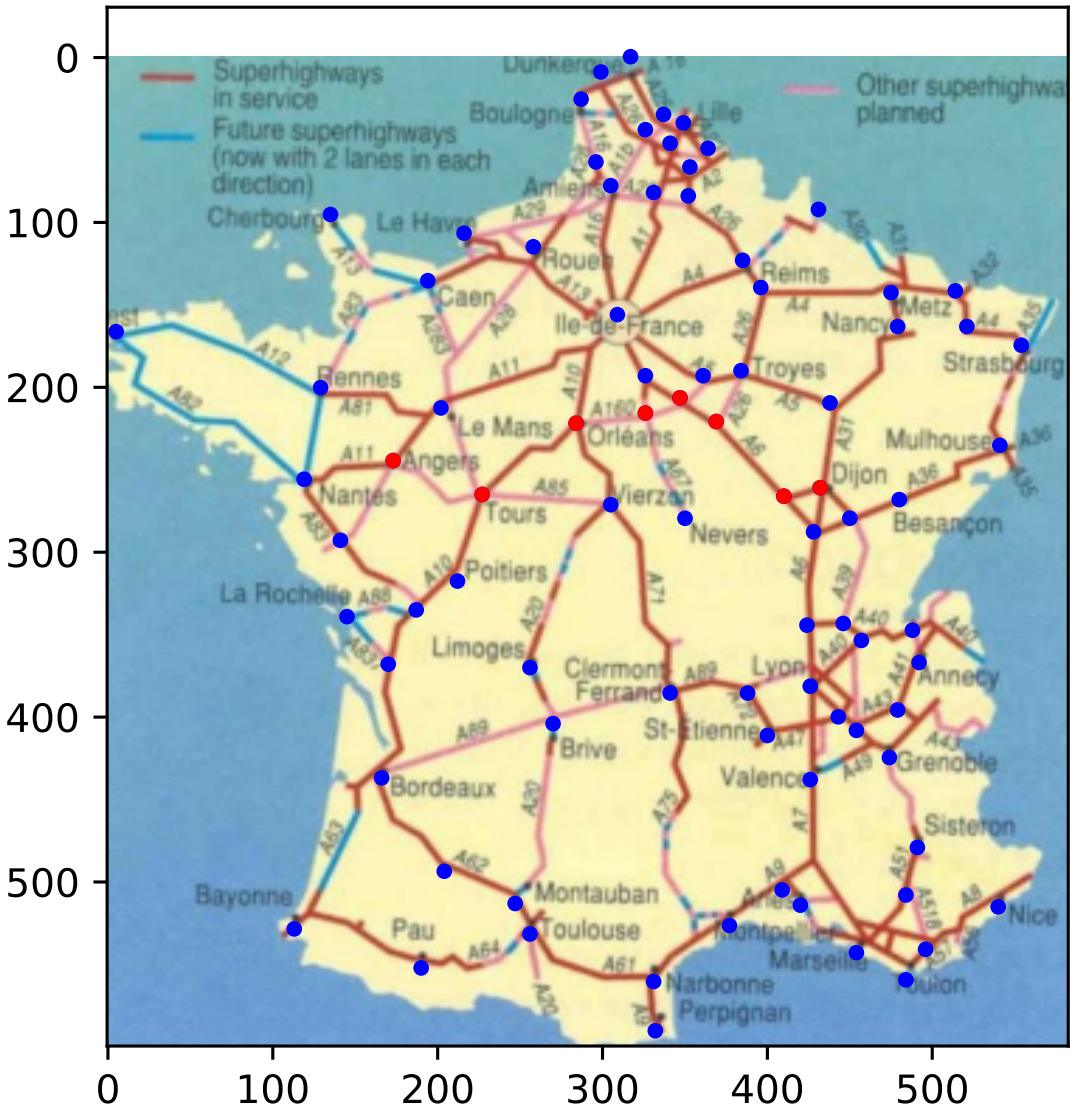
Pseudocodes

Algorithm 11 Algorithme A*

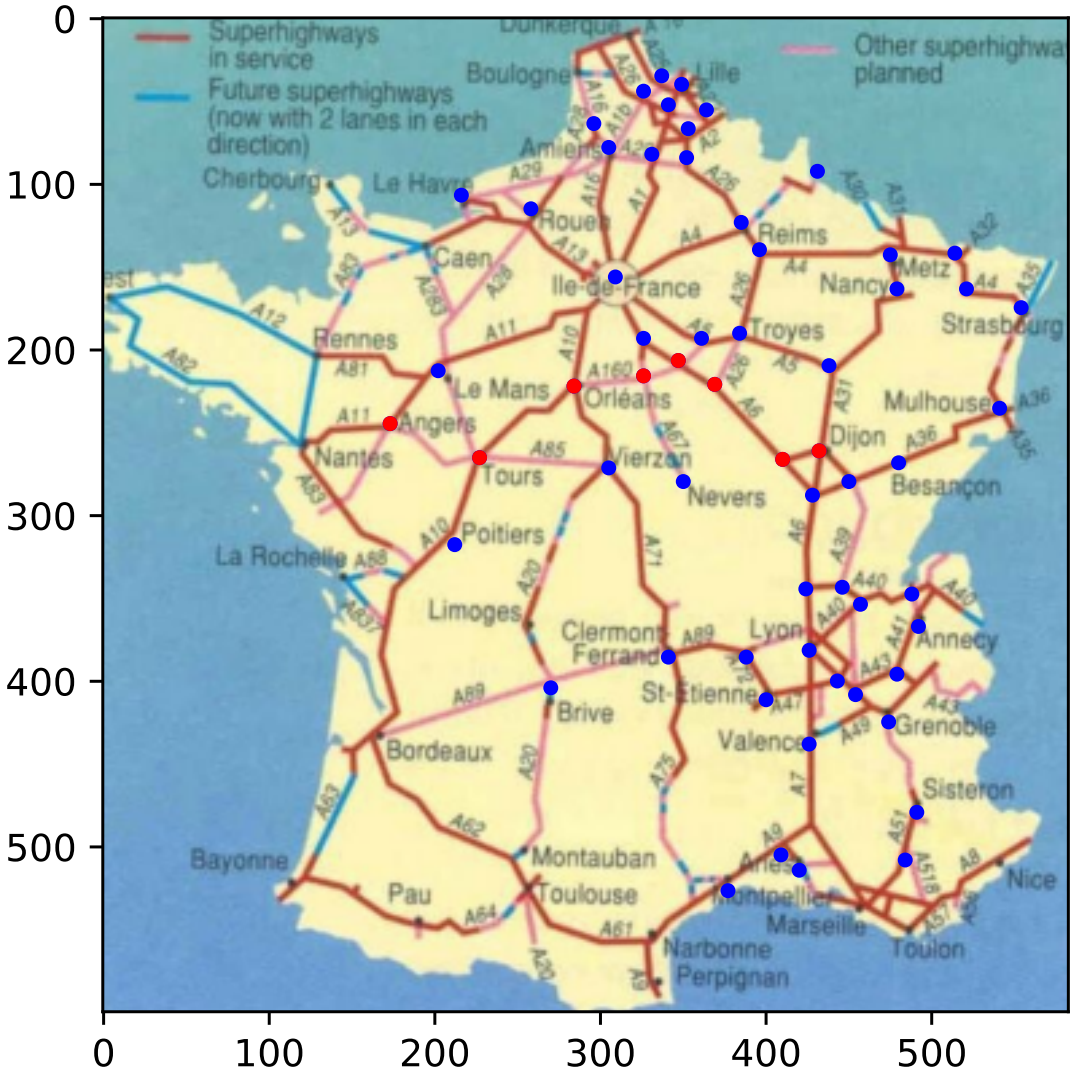
entrée: `graphe, depart, arrivee, dico_coords`
résultat: `chemin`
A_star(`graphe, depart, arrivee, dico_coords`)

```
1:  d_ini = {s : float('inf') for s in graphe}
2:  d_ini[depart] = 0
3:  d_fin, pere = {}, {}
4:  d_heu = heuristique(graphe, arrivee, dico_coords)
5:  tant que taille(d_ini) >0 et arrivee n'est pas dans d_fin faire
6:      s_k = mini_dic_heu(d_ini, d_heu)
7:      pour s_v dans graphe[s_k] faire
8:          w = graphe[s_k][s_v]
9:          si s_v n'est pas dans d_fin et d_ini[s_v] > d_ini[s_k] + w alors
10:             d_ini[s_v] = d_ini[s_k] + w
11:             pere[s_v] = s_k
12:         fin si
13:     fin pour
14:     d_fin[s_k] = d_ini[s_k]
15:     supprime(d[s_k])
16: fin tant que
17: renvoi: trajet(pere, depart, arrivee)
```

Exploration complète avec Dijkstra



Exploration avec Dijkstra



Exploration avec A*