Université Clermont Auvergne
BUT Info 1A
R207 Graphes
Année 2021-2022

Florent Foucaud
Malika More
Thibault Ralet
Carine Simon
Thierry Trévisan

Distances dans les graphes valués

1 Introduction

Une impression de déjà vu

Ce que nous avons vu jusqu'ici concernant les graphes :

- Détecter si un graphe est *planaire ou pas* (avec les *mineurs* interdits)
- *Arbre couvrant de* poids minimal (2 méthodes : Prim, Kruskal)
- Parcours d'un graphe connexe + *File* : *arbre de parcours en largeur* (*distance à la source*)
- Parcours d'un graphe connexe + ***P**ile* : *arbre de parcours en **p**rofondeur* (*ordre topologique*)
- L'*algorithme de Ford-Fulkerson* pour calculer un *flot maximal* dans un réseau (+ lien avec la notion de *coupe minimale*)

Trouver des chemins courts et calculer des distances.

Il s'agit d'un problème récurrent en informatique, que vous allez rencontrer très souvent :

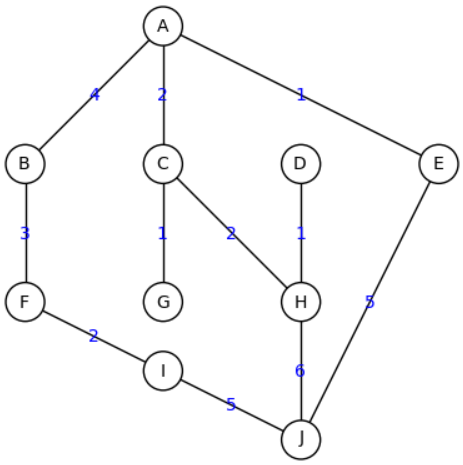
- En réseaux, pour le routage avec le protocole open shortest path first
- Pour modéliser par exemple le calcul des correspondances optimales dans un réseau de transport multi-modal
- En robotique, pour décider de la trajectoire d'un robot
- Dans l'industrie des jeux, comme sous-routine de l'IA gérant un bot

Plusieurs variantes assez semblables

- Toutes les distances / tous les chemins plus courts entre toutes les paires de sommets
- Plusieurs chemins (les 10 chemins les plus courts par exemple, en cas de panne)
- *Les chemins les plus courts depuis un sommet vers tous les autres*
- etc.

2 Algorithme de Dijkstra

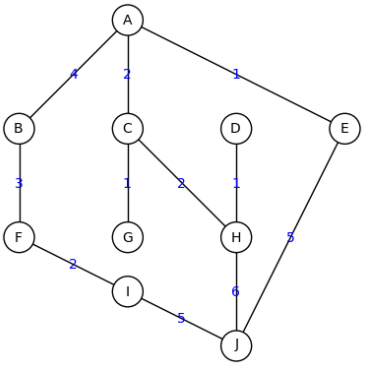
Le problème des distances à un sommet



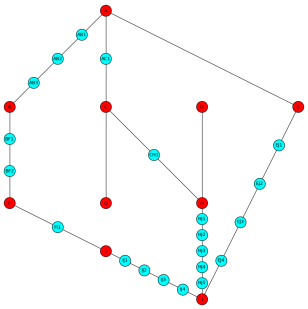
- On se donne un graphe avec des valeurs *positives* sur les arêtes (représentant une distance ou un temps de parcours)
- On veut calculer la distance de A à tous les autres sommets

Première idée : avec un parcours en largeur ?

Graphe avec des distances



⟶ On le transforme en un graphe usuel (= sans distances), en insérant des "sommets intermédiaires".



Puis on effectue un parcours en largeur.

Inconvénients

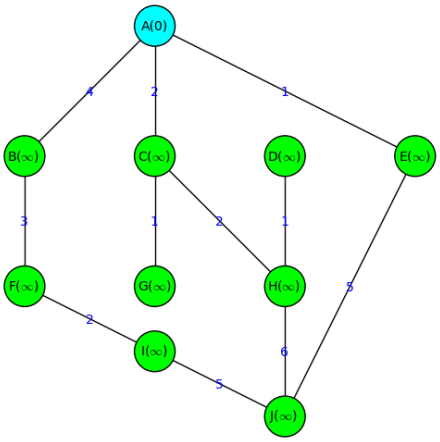
- Il faut distinguer entre les anciens sommets (qui nous intéressent) et les nouveaux sommets (uniquement là pour calculer les distances vers les autres)
- *Si les distances sont grandes, le graphe devient grand* : pour notre petit graphe à 10 sommets, quelques distances de l'ordre de 100 sur certaines arêtes donnent un graphe à plusieurs centaines de sommets !
- Or le temps nécessaire au parcours en largeur *dépend du nombre de sommets :* il est proportionnel à $v + e$ si v est le nombre de sommets et e le nombre d'arêtes

Méthode sans ces inconvénients

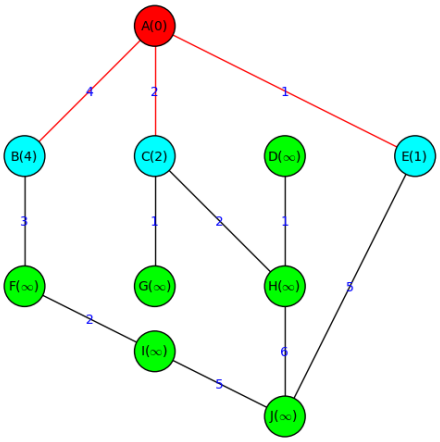
- On simule le principe du parcours en largeur
- Mais on travaille directement sur le graphe avec distance
- Le temps dépend du nombre de sommets
- Il dépend aussi de la taille des distances manipulées mais beaucoup moins qu'avant (plus de détails à ce sujet plus tard)
- C'est le célèbre *algorithme de Dijkstra* (1959)

Un premier exemple de l'algorithme de Dijkstra

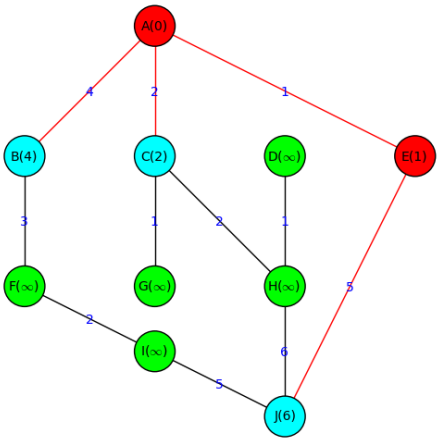
Au début *on ne connaît que le sommet* A de distance 0 à lui-même, et les autres sommets sont à distance ∞



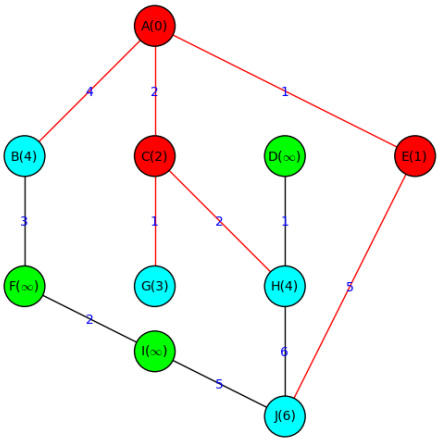
On met à jour les distances vers les sommets B, C et E *voisins de* A, qu'on ajoute à la frontière et on marque A comme traité



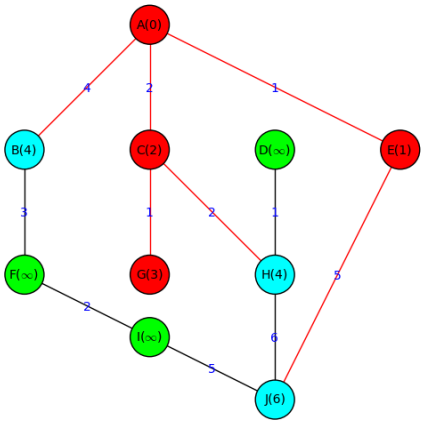
- On choisit comme nouveau sommet courant *celui dont la distance est la plus proche de* A ici le sommet E.
  (cela correspond bien à travailler par niveau dans le parcours en largeur)
- On découvre un voisin de E jusque là inconnu (sommet J) et on met à jour sa distance à A
- On marque E comme traité



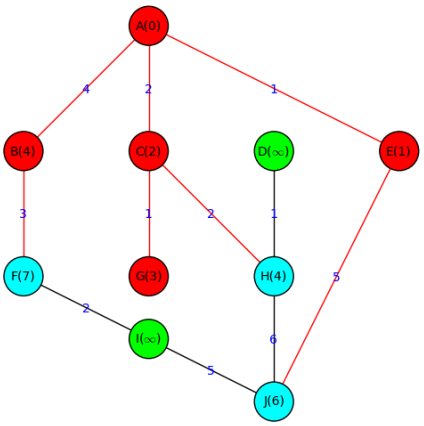
On continue sur le même principe avec le prochain sommet non marqué (c-à-d. de la frontière) le plus proche de A , ici C, qui permet de découvrir G et H et de leur attribuer une distance à A



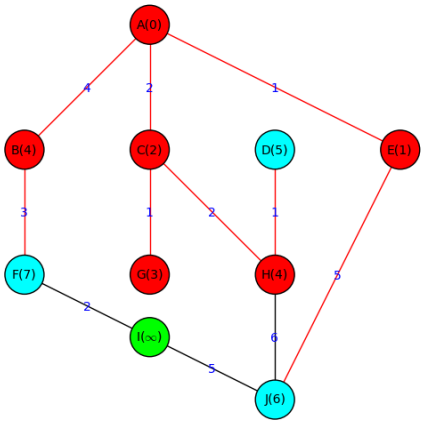
Puis G



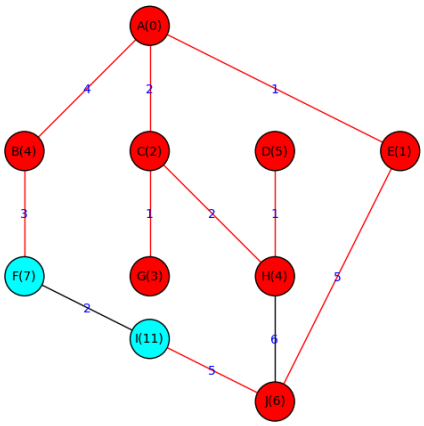
Puis B, qui permet de découvrir F



Puis H, qui permet de découvrir D (notons qu'on aurait pu aussi faire H puis B puisqu'ils étaient tous les deux à distance 4)

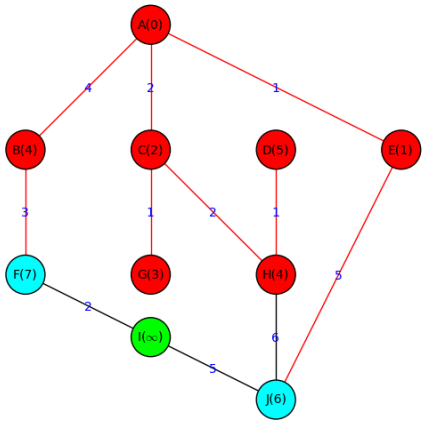


Puis J : on découvre depuis J le sommet I à distance 11

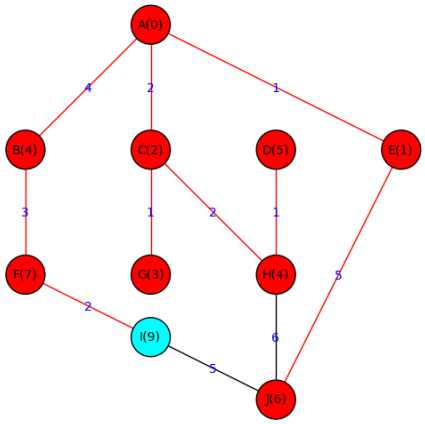


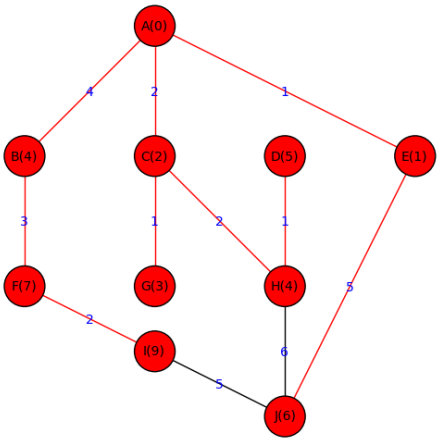
Et enfin I

Puis D



Puis F. **Attention :** notez la *mise à jour* de la distance de I qu'on redécouvre depuis F, pour un chemin de longueur 9





Arbre recouvrant et distances

- Puisqu'on simule le principe d'un parcours en largeur sur le graphe sans distance vu précédent, on obtient évidemment un *arbre recouvrant* apparenté à un parcours en largeur
- Un *plus court chemin de A à un sommet X*, correspond à l'unique chemin dans l'arbre recouvrant de A à X.

L'algorithme de Dijkstra point par point

Initialisation 0 pour le premier sommet, ∞ pour les autres

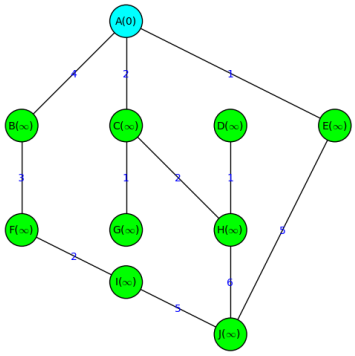
Traitement On procède par marquage progressif des sommets :
- sommet courant = choisir X non marqué de valeur minimale
- pour tout Y non marqué adjacent à X, mettre à jour la distance / le prédécesseur Y si plus court via X
- marquer X

Fin
- quand tous les sommets sont marqués
- distance à la source = valeur trouvée pour chaque sommet
- pour connaître le chemin correspondant à cette distance, il faut noter le prédécesseur

3 Méthode de résolution par tableau

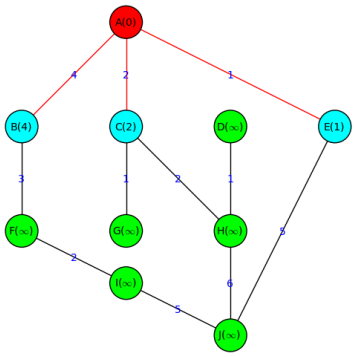
Algorithme de Dijkstra pour les humains en version tableau

Etape 1 **(Initialisation)**
- Placer tous les sommets du graphe dans la 1ère ligne d'un tableau
- Sur la 2ème ligne, écrire la valeur 0 pour le sommet de départ et les valeurs ∞ pour tous les autres sommets



A	B	C	D	E	F	G	H	I	J
0	∞	∞	∞	∞	∞	∞	∞	∞	∞

Etape 2 **(Traitement)**
- Repérer le sommet X de valeur minimale : rayer la colonne sous ce sommet
- Pour tous les sommets Y non rayés :
  - Si Y est adjacent à X alors :
    - p = valeur(X)+poids(arête X-Y)
    - Si p < valeur(Y) alors remplacer la valeur de Y par p ; sinon recopier la valeur(Y).
  - Si Y n'est pas adjacent à X alors recopier la valeur(Y)
- Tant qu'il reste des colonnes non rayées, reprendre l'étape 2.



A	B	C	D	E	F	G	H	I	J
0	∞	∞	∞	∞	∞	∞	∞	∞	∞
•	4	2	∞	1	∞	∞	∞	∞	∞
||	4	2	∞	•	∞	∞	∞	∞	6
||	4	•	∞	||	∞	3	4	∞	6
||	4	||	∞	||	∞	•	4	∞	6
||	•	||	∞	||	7	||	4	∞	6
||	||	||	5	||	7	||	•	∞	6
||	||	||	•	||	7	||	||	∞	6
||	||	||	||	||	7	||	||	11	•
||	||	||	||	||	•	||	||	9	||
||	||	||	||	||	||	||	||	•	||

Etape 3 **(Fin)**

- Le poids de la chaîne la plus courte depuis le sommet de départ est le dernier nombre dans la colonne du sommet d'arrivée
- Pour connaître cette chaîne, on remonte à l'envers : partir du dernier nombre de la colonne du sommet d'arrivée. Rechercher l'étape de sa modification : prendre le sommet qui a été barré à cette étape. Procéder de même jusqu'à remonter à la valeur 0

Retrouver un plus court chemin avec le tableau

Exemple. Distance de A à $I = 9$
Chemin le plus court de A à I (à l'envers) : I–F–B–A

4 Pourquoi ça marche ?

Pourquoi l'algorithme de Dijkstra calcule-t-il les distances avec le sommet de départ ?

- L'algorithme de Dijkstra s'arrête toujours
- La distance des sommets marqués ne change plus
- La distance des sommets marqués est correcte

Preuve (idée) :

À chaque étape, on marque un sommet non marqué. Quand on marque un sommet, on décide qu'une chaîne de *longueur minimum* a été trouvée pour ce sommet. C'est cela qu'il faut vérifier.

Hypothèse de récurrence : *la valeur des sommets déjà marqués est la distance minimum au sommet de départ.*

- Initialisation : (0) est bien la distance entre le sommet de départ et lui-même. OK.
- À chaque nouvelle étape, on compare des valeurs du type $Val(Y) = d(A, X) + Poids(X, Y)$ pour X un sommet marqué et Y un sommet non marqué, et on marque le sommet Y^* de plus petite valeur. Puisque $d(A, X)$ est la plus petite distance de A à X, on est certain d'avoir trouvé la plus petite distance possible entre Y^* et le sommet de départ. La propriété de récurrence reste donc vraie pour le nouveau sommet marqué, Y^*.

Par récurrence, la propriété est donc vraie pour tous les sommets du graphe, CQFD.

Remarque : Si on faisait tourner l'algorithme de Dijkstra sur un graphe non connexe, les sommets inatteignables à partir du sommet de départ resteraient à distance (∞), ce qui est naturel. Mais il faudrait changer la condition d'arrêt en "S'arrêter lorsqu'on ne peut plus marquer de nouveau sommet".

Efficacité de cet algorithme (temps de calcul) ?

- On effectue autant de passages dans la boucle de marquage qu'il y a de sommets (v)
  - à chaque passage, on scanne tous les sommets restants (au plus v) afin de mettre à jour leur distance et de sélectionner le meilleur pour le prochain marquage.
- Le **nombre d'étapes** requises par notre version est donc de l'ordre de $v \times v = v^2$.
- Le **temps de calcul** est donc, lui aussi, proportionnel à v^2.

Version améliorée de Dijkstra

- Il est possible d'améliorer le temps de calcul, dans le cas particulier où le graphe n'est *pas très dense* (nombre d'arêtes e beaucoup plus petit que v^2).
- L'idée est d'utiliser une structure de données particulière appelée le *tas de Fibonacci*, qui permet de gérer efficacement la recherche du sommet courant (sommet min non marqué), l'insertion de nouveaux sommets, et la mise à jour des distances.
- Dans ce cas, nombre d'étapes : $e + v \log(v)$ où v est le nombre de sommets et e le nombre d'arêtes. Ce qui est mieux (plus rapide) que notre complexité précédente en v^2.
- Dans le cas du routage, les graphes sont en effet peu denses, ce qui explique le choix de l'algorithme de Dijkstra (dans sa version optimisée).

5 L'algorithme de Floyd-Warshall

C'est un algorithme conceptuellement plus simple que celui de Dijkstra conçu en 1962. Complexité : v^3 où v est le nombre de sommets. Il fonctionne aussi pour les poids négatifs.

Principe On ordonne les sommets : $s_1, \ldots, s_n$. Soit $d_k(i,j)$ la longueur d'un plus court chemin de s_i à s_j qui a le droit d'utiliser les sommets $s_1, \ldots, s_k$ comme *sommets internes*.

On peut écrire la formule récursive :

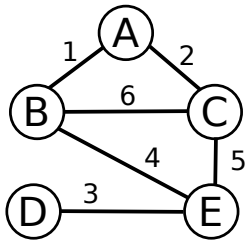
$$d_k(i,j) = \min\{d_{k-1}(i,j), d_{k-1}(i,k) + d_{k-1}(k,j)\}.$$

L'algorithme

Algorithme de Floyd-Warshall pour le graphe G

- Les sommets sont ordonnés : s_1 ... s_n
- On initialise une matrice D de taille $n \times n$, où $D(i,j)$ devra contenir la distance entre le sommet s_i et le sommet s_j. Pour toute arête entre s_i et s_j de poids p_{ij}, on fixe $D(i,j) = p_{ij}$, et pour tout i on fixe $D(i,i) = 0$. Dans les autres cas, on fixe $D(i,j) = \infty$.
- Pour k allant de 1 à n :
  - Pour i allant de 1 à n :
    - Pour j allant de 1 à n : $D(i,j) = \min\{D(i,j), D(i,k) + D(k,j)\}$
- Renvoyer D

Exemple



D_0	A	B	C	D	E
A	0	1	2	∞	∞
B	1	0	6	∞	4
C	2	6	0	∞	5
D	∞	∞	∞	0	3
E	∞	4	5	3	0

D_1	A	B	C	D	E
A	0	1	2	∞	∞
B	1	0	**3**	∞	4
C	2	**3**	0	∞	5
D	∞	∞	∞	0	3
E	∞	4	5	3	0

D_2	A	B	C	D	E
A	0	1	2	∞	**5**
B	1	0	3	∞	4
C	2	3	0	∞	5
D	∞	∞	∞	0	3
E	**5**	4	5	3	0

D_3	A	B	C	D	E
A	0	1	2	∞	5
B	1	0	3	∞	4
C	2	3	0	∞	5
D	∞	∞	∞	0	3
E	5	4	5	3	0

D_4	A	B	C	D	E
A	0	1	2	∞	5
B	1	0	3	∞	4
C	2	3	0	∞	5
D	∞	∞	∞	0	3
E	5	4	5	3	0

D_5	A	B	C	D	E
A	0	1	2	**8**	5
B	1	0	3	**7**	4
C	2	3	0	**8**	5
D	**8**	**7**	**8**	0	3
E	5	4	5	3	0

6 Conclusion

- L'*algorithme de Dijkstra* pour calculer des plus courts chemins depuis un sommet source vers tous les autres sommets d'un graphe (dont les arêtes comportent des poids positifs modélisant des distances)
- Une manière simple (pour un humain) de *présenter l'exécution de Dijkstra avec un tableau.*
- L'*algorithme de Floyd-Warshall* permet de calculer les distances entre toutes les paires de sommets. Il fonctionne aussi avec des poids négatifs.