

Algorithmique et structures de données

Cours INF TC1

Romain Vuillemot
`romain.vuillemot@ec-lyon.fr`

Département Math-Info
École Centrale de Lyon

2022-2023

Sommaire

Structures de données avancées: graphes

- Graphes

- Arbres couvrants minimaux

- Parcours de graphe

- Dessin d'arbres et graphes

Séances de cours de INF TC1

- ▶ Cours 1: introduction aux algorithmiques et structures de données
Hash maps, Piles, Files, Listes de priorité, Listes chaînées
- ▶ Cours 2: stratégies de programmation
Arbres, Parcours Profondeur/Largeur, Diviser pour Régner, Programmation Dynamique, Glouton
- ▶ Cours 3: structure de données avancées
Graphes, Arbres couvrant, Parcours de Graphe, Recherche de Chemin
- ▶ Cours 4: algorithmes avancés
Arbres de Recherche, Tri, Heuristiques

Sommaire

Structures de données avancées: graphes

- Graphes

- Arbres couvrants minimaux

- Parcours de graphe

- Dessin d'arbres et graphes

Structures de données avancées: graphes

Graphes

Graphes

Un **graphe** est une structure de données abstraite constitué d'un ensemble de sommets reliés par des arêtes.

Connaissez-vous des graphes ?

Graphes

Un **graphe** est une structure de données abstraite constitué d'un ensemble de sommets reliés par des arêtes.

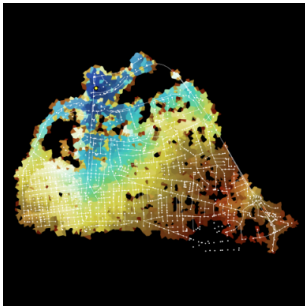
Connaissez-vous des graphes ?

- ▶ Messagerie : le voyageurs de commerce, trajet d'un facteur,
- ▶ Réseaux de communication
- ▶ Gestion du trafic : problème de FLUX, chemins d'encombrement minimum, ..
- ▶ Navigation aérienne (les avions dans des couloirs au ciel !)
- ▶ Le système de transport fermé (circuit fermé) : livraison de marchandises, TSP.
- ▶ Câblage de circuits imprimés
- ▶ ..

Structures de données avancées: graphes

Graphes

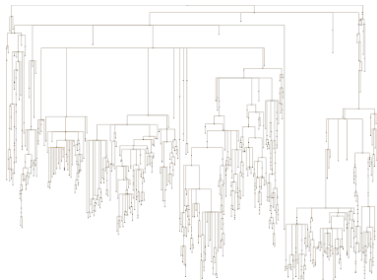
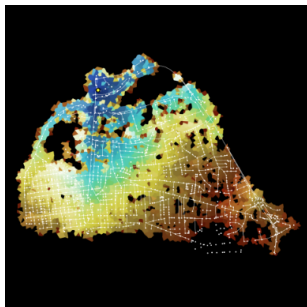
Les plans urbains sont des graphes (mais leur parcours des chemins ou des arbres..)



Structures de données avancées: graphes

Graphes

Les plans urbains sont des graphes (mais leur parcours des chemins ou des arbres..)



<https://observablehq.com/@pierreleripoll/bulle-de-confinement-lyon>
(avec prise en compte de limite temporelle/géographique)

Graphes

Un **graphe** est une structure de données abstraite constitué d'un ensemble de sommets reliés par des arêtes

- ▶ Un graphe $G = (V, E)$ avec
 - ▶ V : ensemble de nœuds (vertex)
 - ▶ $E \in (V \times V)$: ensemble d'arêtes ou arcs (edges)
- ▶ Plusieurs types de graphes : orienté ou non, valué ou non, (fortement) connexe, bi-parties, Graphe dense $|E| = O(|V|^2)$.
- ▶ Graphe **valué** ont une valeur numérique associée à leurs nœuds appelée **poids** (*weight*, pondération).
- ▶ w est **adjacent** de v si $(v, w) \in E$
- ▶ Longueur d'un chemin contenant N nœuds=nombre d'arcs= $n-1$
- ▶ Un arbre : un graphe acyclique connexe

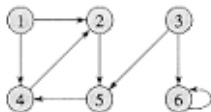
Domaine général du (vaste) domaine de la théorie des graphes¹

¹https://github.com/11Sourcecell/learn_math_fast#graph-theory

Structures de données avancées: graphes

Structure de données Graphe

Structure de graphe : liste de liens vs matrice d'adjacence



(a)



(b)

	1	2	3	4	5	6
1	0	1	0	1	0	0
2	0	0	0	0	1	0
3	0	0	0	0	1	1
4	0	1	0	0	0	0
5	0	0	0	1	0	0
6	0	0	0	0	0	1

(c)

Une matrice d'adjacence :

- ▶ Carrée : il y a autant de lignes que de colonnes.
- ▶ Il n'y a que des zéros sur la diagonale allant du coin supérieur gauche au coin inférieur droit. Un 1 sur la diagonale indiquerait une boucle.
- ▶ Elle est symétrique : $m_{ij} = m_{ji}$ si graphe non-orienté.

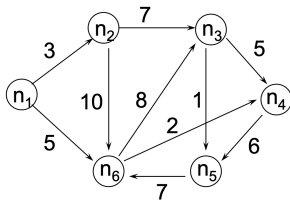
Autres structures (voir cours sur Arbres) : par dictionnaire (dict), ensembles (set), ..

Structures de données avancées: graphes

Structure de données Graphe

Question :

Quelle est la matrice d'adjacence de ce graphe ?

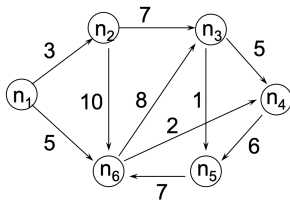


Structures de données avancées: graphes

Structure de données Graphe

Question :

Quelle est la matrice d'adjacence de ce graphe ?



	n ₁	n ₂	n ₃	n ₄	n ₅	n ₆
n ₁	0	3				5
n ₂		0	7			10
n ₃			0	5	1	
n ₄				0	6	
n ₅					0	7
n ₆			8	2		0

Structures de données avancées: graphes

Matrice d'adjacence

Structure de données d'une matrice d'adjacence ?

```
class Graph:

    # Constructor
    def __init__(self, x):
        self.__n = x
        self.__g = [[0 for x in range(10)] for y in range(10)]

        for i in range(0, self.__n):
            for j in range(0, self.__n):
                self.__g[i][j] = 0

    def displayAdjacencyMatrix(self):
        for i in range(0, self.__n):
            print()
            for j in range(0, self.__n):
                print("", self.__g[i][j], end = "")
```

Structures de données avancées: graphes

Matrice d'adjacence

Structure de données d'une matrice d'adjacence ?

```
def addEdge(self, x, y):
    if (x < 0) or (x >= self.__n):
        print("Vertex {} does not exist!".format(x))
    if (y < 0) or (y >= self.__n):
        print("Vertex {} does not exist!".format(y))

    if(x == y):
        print("Same Vertex!")
    else:
        self.__g[y][x] = 1
        self.__g[x][y] = 1

def removeEdge(self, x, y):
    if (x < 0) or (x >= self.__n):
        print("Vertex {} does not exist!".format(x))
    if (y < 0) or (y >= self.__n):
        print("Vertex {} does not exist!".format(y))
    if(x == y):
        print("Same Vertex!")
    else:
        self.__g[y][x] = 0
        self.__g[x][y] = 0
```

Structures de données avancées: graphes

Matrice d'adjacence

Structure de données d'une matrice d'ajacence ?

```
obj = Graph(6);

obj.addEdge(0, 1)
obj.addEdge(0, 2)
obj.addEdge(0, 3)
obj.addEdge(0, 4)
obj.addEdge(1, 3)
obj.addEdge(2, 3)
obj.addEdge(2, 4)
obj.addEdge(2, 5)
obj.addEdge(3, 5)

obj.displayAdjacencyMatrix();

0 1 1 1 1 0
1 0 0 1 0 0
1 0 0 1 1 1
1 1 1 0 0 1
1 0 1 0 0 0
0 0 1 1 0 0

obj.removeEdge(2, 3);
obj.displayAdjacencyMatrix();

0 1 1 1 1 0
1 0 0 1 0 0
1 0 0 0 1 1
1 1 0 0 0 1
```

Structures de données avancées: graphes

Matrice d'adjacence

Structure de données d'une matrice d'ajacence? (module networkx)

```
import networkx as nx

graph = {'1': [{ '2': '15' }, { '4': '7' }, { '5': '10' }],
        '2': [{ '3': '9' }, { '4': '11' }, { '6': '9' }],
        '3': [{ '5': '12' }, { '6': '7' }],
        '4': [{ '5': '8' }, { '6': '14' }],
        '5': [{ '6': '8' }]}

new_graph = nx.Graph()

for source, targets in graph.iteritems():
    for inner_dict in targets:
        assert len(inner_dict) == 1
        new_graph.add_edge(int(source) - 1,
                           int(inner_dict.keys()[0]) - 1,
                           weight=inner_dict.values()[0])

adjacency_matrix = nx.adjacency_matrix(new_graph)

>>> [[ 0., 15.,  0.,  7., 10.,  0.],
      [15.,  0.,  9., 11.,  0.,  9.],
      [ 0.,  9.,  0.,  0., 12.,  7.],
      [ 7., 11.,  0.,  0.,  8., 14.],
      [10.,  0., 12.,  8.,  0.,  8.],
      [ 0.,  9.,  7., 14.,  8.,  0.]])
```

Structures de données avancées: graphes

Structure de données Graphe

Dictionnaire. Structure de graphe en dictionnaire et recherche des noeuds/arêtes

```
graph = { "a" : ["c"],
          "b" : ["c", "e"],
          "c" : ["a", "b", "d", "e"],
          "d" : ["c"],
          "e" : ["c", "b"],
          "f" : []
        }

def genere_arretes(graph):
    edges = []
    for node in graph:
        for neighbour in graph[node]:
            edges.append((node, neighbour))

    return edges

# Affiche les sommets
print(list(graph.keys()))

# Affiche les arrêtes
print(genere_arretes(graph))

# >>> [('a', 'c'), ('c', 'a'), ('c', 'b'), ('c', 'd'), ('c', 'e'),
#       ('b', 'c'), ('b', 'e'), ('e', 'c'), ('e', 'b'), ('d', 'c')]
```

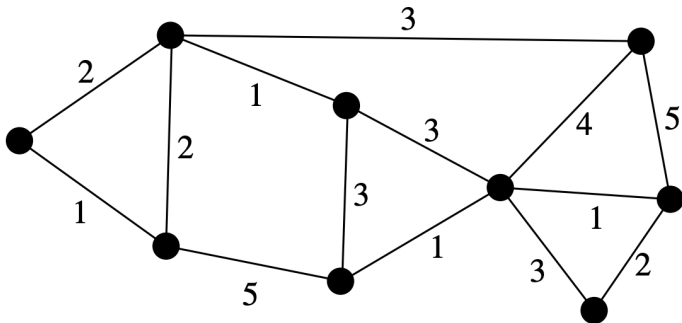
Arbres couvrants minimaux

Un **arbre couvrant minimal** (Minimum spanning tree (MST)) d'un graphe est un sous-ensemble d'arêtes qui connecte tous les sommets, en minimisant la somme totale de la valeur des arêtes.

Arbres couvrants minimaux

Un **arbre couvrant minimal** (Minimum spanning tree (MST)) d'un graphe est un sous-ensemble d'arêtes qui connecte tous les sommets, en minimisant la somme totale de la valeur des arêtes.

Question : Quel est l'arbre couvrant minimal de ce graphe ?

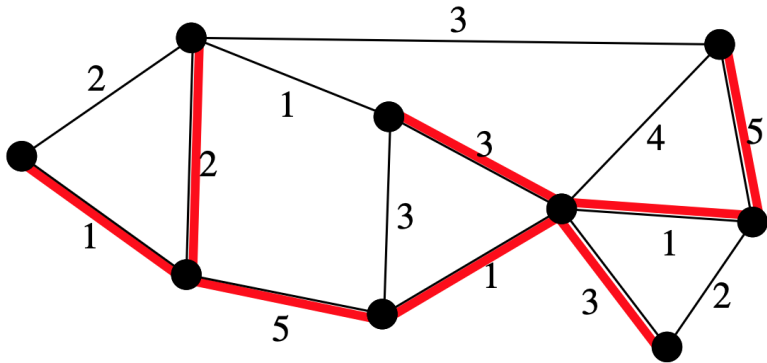


Structures de données avancées: graphes

Arbres couvrants minimaux

Un **arbre couvrant minimal** (Minimum spanning tree (MST)) d'un graphe est un sous-ensemble d'arêtes qui connecte tous les sommets, en minimisant la somme totale de la valeur des arêtes.

Réponse ?

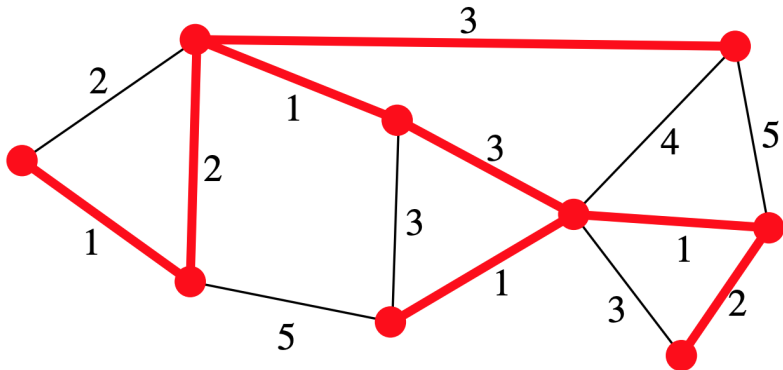


Structures de données avancées: graphes

Arbres couvrants minimaux

Un **arbre couvrant minimal** (Minimum spanning tree (MST)) d'un graphe est un sous-ensemble d'arêtes qui connecte tous les sommets, en minimisant la somme totale de la valeur des arêtes.

Réponse !



Arbres couvrants minimaux

Remarques :

- ▶ Si un graphe a un cycle ou plusieurs chemins, alors on supprime les boucles et doubles arcs (on élève ceux avec poids plus fort)
- ▶ Si un graphe a N sommets, son MST aura $N - 1$ arêtes
- ▶ Un graphe peut avoir plusieurs arbres couvrants, le MST est celui qui a le poids le plus faible
- ▶ Un arbre n'a qu'un seul arbre couvrant : lui-même

Structures de données avancées: graphes

Arbres couvrants minimaux

Remarques :

- ▶ Si un graphe a un cycle ou plusieurs chemins, alors on supprime les boucles et doubles arcs (on élève ceux avec poids plus fort)
- ▶ Si un graphe a N sommets, son MST aura $N - 1$ arêtes
- ▶ Un graphe peut avoir plusieurs arbres couvrants, le MST est celui qui a le poids le plus faible
- ▶ Un arbre n'a qu'un seul arbre couvrant : lui-même

Applications

- ▶ Câblage de réseau téléphonique, internet, etc.
- ▶ Permet de linéariser la navigation dans un graphe

On va voir deux algorithmes **Prim** et de **Kruskal**

Arbres couvrants minimaux

Algorithme de Prim :

1. On part d'un arbre initial réduit à un seul sommet du graphe.
2. À chaque itération, on agrandit l'arbre en lui ajoutant le sommet libre accessible de plus petit poids possible.
3. On stoppe quand l'arbre est recouvrant

Arbres couvrants minimaux

Algorithme de Prim :

1. On part d'un arbre initial réduit à un seul sommet du graphe.
2. À chaque itération, on agrandit l'arbre en lui ajoutant le sommet libre accessible de plus petit poids possible.
3. On stoppe quand l'arbre est recouvrant

Stratégie de programmation ?

Arbres couvrants minimaux

Algorithme de Prim :

1. On part d'un arbre initial réduit à un seul sommet du graphe.
2. À chaque itération, on agrandit l'arbre en lui ajoutant le sommet libre accessible de plus petit poids possible.
3. On stoppe quand l'arbre est recouvrant

Stratégie de programmation ? Glouton

Structures de données avancées: graphes

Arbres couvrants minimaux

Algorithme de Prim (utilise le module pythonds):

```
from pythonds.graphs import PriorityQueue, Graph, Vertex

def prim(G,start):
    pq = PriorityQueue()

    for v in G:
        v.setDistance(sys.maxsize)
        v.setPred(None)

    start.setDistance(0)
    pq.buildHeap([(v.getDistance(),v) for v in G])

    while not pq.isEmpty():
        currentVert = pq.delMin()

        for nextVert in currentVert.getConnections():
            newCost = currentVert.getWeight(nextVert)

            if nextVert in pq and newCost<nextVert.getDistance():
                nextVert.setPred(currentVert)
                nextVert.setDistance(newCost)
                pq.decreaseKey(nextVert,newCost)
```

Arbres couvrants minimaux

Algorithme de Kruskal :

1. On part d'une forêt d'arbres constitués de chacun des sommets isolés du graphe.
2. À chaque itération, on ajoute à cette forêt l'arête de poids le plus faible ne créant pas de cycle avec les arêtes déjà choisies.
3. On stoppe quand on a examiné toutes les arêtes.

Arbres couvrants minimaux

Algorithme de Kruskal :

1. On part d'une forêt d'arbres constitués de chacun des sommets isolés du graphe.
2. À chaque itération, on ajoute à cette forêt l'arête de poids le plus faible ne créant pas de cycle avec les arêtes déjà choisies.
3. On stoppe quand on a examiné toutes les arêtes.

Stratégie de programmation ?

Arbres couvrants minimaux

Algorithme de Kruskal :

1. On part d'une forêt d'arbres constitués de chacun des sommets isolés du graphe.
2. À chaque itération, on ajoute à cette forêt l'arête de poids le plus faible ne créant pas de cycle avec les arêtes déjà choisies.
3. On stoppe quand on a examiné toutes les arêtes.

Stratégie de programmation ? Glouton

Structures de données avancées: graphes

Arbres couvrants minimaux

Algorithme de Kruskal :

```
parent = dict()
rank = dict()

def make_set(vertice):
    parent[vertice] = vertice
    rank[vertice] = 0

def find(vertice):
    if parent[vertice] != vertice:
        parent[vertice] = find(parent[vertice])
    return parent[vertice]

def union(vertice1, vertice2):
    root1 = find(vertice1)
    root2 = find(vertice2)
    if root1 != root2:
        if rank[root1] > rank[root2]:
            parent[root2] = root1
        else:
            parent[root1] = root2
    if rank[root1] == rank[root2]: rank[root2] += 1

def kruskal(graph):
    for vertice in graph['vertices']:
        make_set(vertice)
    minimum_spanning_tree = set()
    edges = list(graph['edges'])
    edges.sort()
    #print edges
    for edge in edges:
        weight, vertice1, vertice2 = edge
        if find(vertice1) != find(vertice2):
            union(vertice1, vertice2)
            minimum_spanning_tree.add(edge)

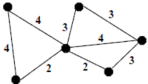





    return sorted(minimum_spanning_tree)
```

Structures de données avancées: graphes

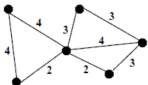


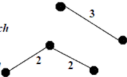
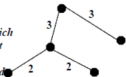
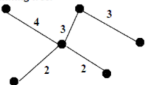
Arbres couvrants minimaux

Prims vs Kruskal :

Prim's Algorithm

<p>1 Given a network.....</p> 	<p>2 Choose a vertex</p> 	<p>3 Choose the shortest edge from this vertex.</p> 
<p>4 Choose the nearest vertex not yet in the solution.</p> 	<p>5 Choose the next nearest vertex not yet in the solution, when there is a choice choose either.</p> 	<p>6 Repeat until you have a minimal spanning tree.</p> 

Kruskal's Algorithm

<p>1 Given a network.....</p> 	<p>2 Choose the shortest edge (if there is more than one, choose any of the shortest).....</p> 	<p>3 Choose the next shortest edge and add it.....</p> 
<p>4 Choose the next shortest edge which wouldn't create a cycle and add it.</p> 	<p>5 Choose the next shortest edge which wouldn't create a cycle and add it.</p> 	<p>6 Repeat until you have a minimal spanning tree.</p> 

Structures de données avancées: graphes

Parcours de graphe

Comment peut-on parcourir les graphes ?

Structures de données avancées: graphes

Parcours de graphe

Comment peut-on parcourir les graphes ?
Comme les arbres (ou presque)

Parcours en profondeur :

- Ce parcours est semblable au parcours en profondeur des arbres.
- Il est en général assez performant mais si le graphe contient un cycle "à gauche", alors aucune réponse ne pourra être produite.
- Pseudo code du parcours en profondeur (version de base, **sans marquage**) :

```
Procédure profondeur ( G : ref Noeud) =  
Début  
  Si Non est_vide(G) alors  
    traiter(noeud_courant(G));    // un traitement quelconque  
    Pour X dans adjacents(noeud_courant(G))  
      profondeur (X);  
    Fin pour;  
  Fin si;  
Fin profondeur ;
```

- Ce parcours ne mémorise rien et peut donc entrer dans une boucle infinie en cas de circuit ou une boucle (loop).

Structures de données avancées: graphes

Parcours de graphe

- Parcours récursif en profondeur **avec marquage**

```
Procédure profonde (G : ref Noeud) =  
Début  
  Si Non est_vide(G) alors  
    marquer(noeud_courant(G));           # on marque tout de suite et ...  
    traiter(noeud_courant(G));           #traitement  quelconque  
    Pour X dans adjacents(noeud_courant(G))  
      Si X n'est pas marqué               # ... on contrôle ici  
      Alors profonde(X);  
    Fin si;  
  Fin pour;  
Fin si;  
Fin profonde ;
```

- Ce parcours marque les noeuds déjà visités pour ne pas les réessayer.
- Les mécanismes de marquage :
 - marquage à l'intérieur du noeud
 - marquage par une structure de données externe au graphe (un tableau, ...)

Structures de données avancées: graphes

Parcours de graphe

- Une autre manière de parcourir le graphe en marquant les noeuds est :

```
Procédure profonde (G : ref Noeud) =  
Début  
  Si Non est_vide(G) ET noeud_courant(G) n'est pas marqué # On contrôle en entrant  
  Alors  
    marquer(noeud_courant(G)); # et on marque ici  
    traiter(noeud_courant(G)); # traitement quelconque  
    Pour X dans adjacents(noeud_courant(G))  
      profonde(X);  
    Fin pour;  
  Fin si;  
Fin profonde ;
```

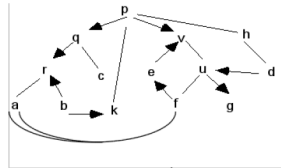
- Trace de parcours en profondeur de **p** à **u** :

profondeur(p) → profondeur(q) →

profondeur(r) → profondeur(a) →

profondeur(f) → profondeur(e) →

profondeur(v) → profondeur(u)



Structures de données avancées: graphes

Parcours de graphe

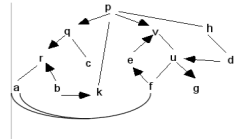
● Principe de l'algorithme itératif en Profondeur :

→ On a besoin d'une pile (LIFO) pour simuler la pile machine (expliquer).

```
Procédure profondeur_iteratif(G)
  Pile=vide
  empiler(noeud_courant(G))
  Tant que NON est_vide(Pile)
    Noeud ← dépiler(Pile)
    Si NON est_marqué(X)          # Utile si un noeud se trouve 2 fois dans la Pile
      marquer(Noeud)
      traiter(Noeud)
      Pour X dans adjacents(Noeud) # A considérer dans l'ordre voulu (p.ex. inversé = de gche à dte, V. Trace)
        Si NON est_marqué(X)
          Alors empiler(Pile, X) # empiler dans le désordre
      fin pour
  Fin Tant que
Fin profondeur_iteratif
```

● Trace de parcours en profondeur de p à d (on souligne si traité) :

[p] → [h,v,k,q] → [h,v,k,c,r] → [h,v,k,c,r] → [h,v,k,c,a] → [h,v,k,c,f]
→ [h,v,k,c,e,u] → [h,v,k,c,e,g,v] → v se trouve 2 fois dans la pile sans être marqué
→ [h,v,k,c,e,g] → [h,v,k,c,e] → [h,v,k,c] → [h,v,k] → [h,v]
→ v : la 2e fois ! déjà traité. puis
→ [h] → [d] → []



Structures de données avancées: graphes

Parcours de graphe

Méthode générale de parcours en profondeur (Depth First Traversal – DTF)

1. Mettre le nœud source dans la **pile**.
2. Retirer le nœud du début de la pile pour le traiter.
3. Mettre tous les voisins non explorés dans la pile (au début).
4. Si la pile n'est pas vide reprendre à l'étape 2.

Structures de données avancées: graphes

Parcours de graphe

Code parcours profondeur (Depth First Traversal – DTF)

```
def dfs(graph, start):
    visited, stack = set(), [start]
    while stack:
        vertex = stack.pop()
        if vertex not in visited:
            visited.add(vertex)
            stack.extend(graph[vertex] - visited)
    return visited

graph = {'A': set(['B', 'C']),
        'B': set(['A', 'D', 'E']),
        'C': set(['A', 'F']),
        'D': set(['B']),
        'E': set(['B', 'F']),
        'F': set(['C', 'E'])}

dfs(graph, 'A') # {'E', 'D', 'F', 'A', 'C', 'B'}
```

Structures de données avancées: graphes

Parcours de graphe

Application du parcours en profondeur : **Tri topologique**

- ▶ Permet de linéariser un graphe acyclique orienté (ou dag, *directed acyclic graph*)
- ▶ Trouver une relation d'ordre total entre les nœuds qui respecte l'ordre partiel, i.e. trouver un ordre des nœud tel qu'un nœud soit toujours visité avant ses successeurs
- ▶ Il suffit pour cela de classer les nœuds dans l'ordre inverse où ils sont coloriés
- ▶ Mais pas unique !

Autres applications du DFS : recherche du plus court chemin, détection de cycles, graphe de dépendance d'un programme, etc.

Exercice : Est-ce qu'un chemin existe entre deux sommets ? Les arrêtes étant $[[0,1], [1,2], [2,0]]$ et les sommets de départ/arrivée $d = 0$ et $a = 2$.

Structures de données avancées: graphes

Parcours de graphe

Exercice : Est-ce qu'un chemin existe entre deux sommets ? Les arrêtes étant $[[0,1], [1,2], [2,0]]$ et les sommets de départ/arrivée $d = 0$ et $a = 2$.

```
voisins = [[] for i in range(n)]
for i, j in edges:
    voisins[i].append(j)
    voisins[j].append(i)

stack = [start]
visited = set(stack)
while stack:
    cur = stack.pop()
    if cur == end:
        return True
    for v in voisins[cur]:
        if v not in visited:
            stack.append(v)
            visited.add(v)
return False
```


Exercice : Existe-t-il un cycle dans un graphe ?

Structures de données avancées: graphes

Parcours de graphe

Exercice : Existe-t-il un cycle dans un graphe ?

Parcours en profondeur (itératif)

```
graph = {0: [1], 1: [2], 2: [3], 3: [4], 4: [1]}
def cycle_existe(G):
    color = { u : "white" for u in G } # Noeuds tous blancs
    trouve_cycle = [False]

    for u in G:                                # On visite tous les noeuds
        if color[u] == "white":
            dfs_visite(G, u, color, trouve_cycle)
            if trouve_cycle[0]:
                break
    return trouve_cycle[0]

def dfs_visite(G, u, color, trouve_cycle):
    if trouve_cycle[0]:
        return
    color[u] = "gray"
    for v in G[u]:
        if color[v] == "gray":
            trouve_cycle[0] = True
            return
        if color[v] == "white":
            dfs_visite(G, v, color, trouve_cycle)
    color[u] = "black"

print(cycle_existe(graph))
```

Parcours de graphe

Exercice : Existe-t-il un cycle dans un graphe ? Etant donnée une liste d'adjacence en entrée $[[0, 1], [0, 2], [0, 3], [1, 4]]$

Structures de données avancées: graphes

Parcours de graphe

Exercice : Existe-t-il un cycle dans un graphe ? Etant donnée une liste d'adjacence en entrée `[[0, 1], [0, 2], [0, 3], [1, 4]]`
Parcours en profondeur (récursif) par coloriage

```
# on construit un graphe sous forme de liste d'adjacence
# l'idée est d'utiliser la valeur du sommet comme index du dict
def constructionGraphe(liste_sommets):

    graphe = {}

    for src, dest in liste_sommets:
        if src not in graphe:
            graphe[src] = []

        graphe[src].append(dest)

        if dest not in graphe:
            graphe[dest] = []

    return graphe
```

Structures de données avancées: graphes

Parcours de graphe

Exercice : Existe-t-il un cycle dans un graphe ? Étant donnée une liste d'adjacence en entrée `[[0, 1], [0, 2], [0, 3], [1, 4]]`

Parcours en profondeur (récursif)

```
def verifieGraphe(graphe, start = 0):
    visited = set()

    def dfs(root):
        visited.add(root)
        for node in graphe[root]:
            if node in visited: # déjà visité
                return False
            if not dfs(node): # on continue dans le graphe
                return False
        return True

    return dfs(start).add(v)
return False

if __name__ == '__main__':

    liste_sommets_acyclique = [[0, 1], [0, 2], [0, 3], [1, 4]]
    g = constructionGraphe(liste_sommets_acyclique)
    print(verifieGraphe(g)) # True

    liste_sommets_cyclique = [[0,1], [1,2], [2,3], [1,3], [1,4]]
    g = constructionGraphe(liste_sommets_cyclique)
    print(verifieGraphe(g)) # False
```

Structures de données avancées: graphes

Parcours de graphe

Le principe de parcours en largeur

- traiter par niveau : traiter chaque noeud
- puis traiter chacun de ses successeurs avant de traiter les successeur du prochain niveau.

☞ **Par contre** : contrairement au parcours en profondeur,

s'il existe un cycle dans le graphe, des réponses seront néanmoins produites.

- Ce parcours s'adapte mieux à une solution itérative.

→ On utilise une file d'attente pour la construction de la liste des noeuds à traiter.

- Les opérations sur une file :

- enfiler(X, File)
- defiler(File)
- premier(File)
- ...

Structures de données avancées: graphes

Parcours de graphe

Pseudo algorithme récursif de parcours en largeur (graphe connexe) :

- Remarque : certains noeuds sont traités plusieurs fois, voir marquage

```
File : file d'attente = File_vide;  
  
Procédure largeur_récuratif (G : ref Noeud) =  
Début  
    Si Non est_vide(G) alors  
        traiter(noeud_courant(G)); //une opération quelconque  
        Pour X dans adjacents(noeud_courant(G))  
            File=Enfiler(X, File);  
        Fin pour;  
        X = Sommet(File); // Respect du TDA File  
        File = Défiler(File);  
        largeur_récuratif (X);  
    Fin si;  
Fin largeur_récuratif ;
```

☞ N.B. : pas de marquage

→ en l'absence de marquage, certains noeuds sont traités plusieurs fois.

Structures de données avancées: graphes

Parcours de graphe

Cas de graphe connexe : tout noeud est connecté aux autres.

→ on peut travailler (se laisser guider) directement par la file d'attente

```
File : file d'attente = File_vide;  
enfiler(la racine du graphe G, File)  
  
Procédure largeur_récuratif (G, File) =  
Début  
  Si Non_est_vide(File) alors  
    X=Sommet(File);  
    File=Défiler(File);           // car défiler ne renvoie pas un élément (cf. TDA File)  
    traiter(noeud_courant(X));    //un traitement quelconque  
    Pour X dans adjacents(noeud_courant(G))  
      File=Enfiler(X, File);  
    Fin pour;  
    largeur_récuratif (G, File);  
  Fin si;  
Fin largeur_récuratif ;
```

- Initialement, il faut enfiler le noeud de départ.
- ☞ Le marquage (pour éviter de re-traiter un noeud) se fait comme pour le parcours en profondeur (voir ci-après).

Structures de données avancées: graphes

Parcours de graphe

- On peut récupérer le chemin par le mécanisme *Coming-From*.
- Pour un graphe connexe, la version suivante récupère le chemin par ce mécanisme.

```
File : file d'attente =File_vide;  
enfiler(la racine du graphe G, File)  
CF : tableau indicé par les noeuds initialisé à 0  
CF[Départ]=Départ  
  
Procédure chemin_largeur_récuratif (File) =  
Début  
  Si Non est_vide(File) alors  
    X=Sommet(File);  
    File=Défiler(File);           // car défiler ne renvoie pas un élément (cf. TDA File)  
    traiter(noeud_courant(X));    //une opération quelconque  
    Pour X dans adjacents(noeud_courant(G))  
      File=Enfiler(X, File);  
      CF[X]=noeud_courant(G)  
    Fin pour;  
    largeur_récuratif (File);  
  Fin si;  
Fin largeur_récuratif ;
```

- Exercice : comment extraire ensuite le chemin ?
- Note : voir la version Python du problème de la monnaie.

Structures de données avancées: graphes

Parcours de graphe

L'algorithme itératif de parcours en largeur

```
File : file d'attente=File_vide;  
  
Procédure largeur_itératif ( G : graphe)  
  File=vide  
  Début  
    Si Non est_vide(G) alors  
      Enfiler(noeud_courant(G), File);  
    Fin si;  
    Tant que Non est_vide(File)  
      N = Sommet(File);  
      File = Défiler(File);  
      traiter(N); // ou visiter(N)  
      Pour X dans adjacents(N)  
        File = Enfiler(X, File);  
      Fin pour;  
    Fin Tant que;  
  Fin largeur_itératif ;
```

☞ N.B. : pas de marquage

Structures de données avancées: graphes

Parcours de graphe

L'algorithme itératif de parcours en largeur avec marquage

Ici, la File peut contenir des doublons qui ne seront pas traités une seconde fois.

```
File : file d'attente=File_vide;  
Procédure largeur_itératif_marquage ( G : graphe)  
Début  
  Si Non est_vide(G)  
    Alors  
      Enfiler(noeud_courant(G), File);  
    Fin si;  
  Tant que Non est_vide(File)  
    N = Sommet(File);  
    File = Défiler(File);  
    Si est_marqué(N)  
      Alors passer à l'itération suivante ;  
    Sinon marquer(N);  
    Fin si;  
    traiter(N); // ou visiter(N)  
    Pour X dans adjacents(N);  
      File=Enfiler(X, File);  
    Fin pour;  
  Fin Tant que;  
Fin largeur_itératif_marquage ;
```

Structures de données avancées: graphes

Parcours de graphe

Une variante parcours en largeur itératif (avec marquage)

- Une seconde version avec marquage :
 - on marque les noeuds non marqués placés dans la file.
- La file ne contiendra pas de doublon.

```
File : file d'attente=File_vide;  
Procédure premier_chemin_largeur_itératif ( G : graphe)  
Début  
  Si Non_est_vide(G) alors  
    Enfiler(noeud_courant(G), File);  
    marquer(noeud_courant(G));  
  Fin si;  
  Tant que Non_est_vide(File)  
    N = Sommet(File);  
    File = Défiler(File);  
    traiter(N); // ou visiter(N)  
    Pour X dans adjacents(N) non marqués  
      File = Enfiler(X, File);  
      marquer(X);  
    Fin pour;  
  Fin Tant que;  
Fin premier_chemin_largeur_itératif ;
```

Parcours de graphe

Méthode générale de parcours en largeur (Breadth First Traversal – BFT)

1. Mettre le nœud source dans la **file**.
2. Retirer le nœud du début de la file pour le traiter.
3. Mettre tous les voisins non explorés dans la file (à la fin).
4. Si la file n'est pas vide reprendre à l'étape 2.

Structures de données avancées: graphes

Parcours de graphe

Code. Parcours largeur (Breadth First Traversal – BFT)

```
import collections
class graph:
    def __init__(self, gdict=None):
        if gdict is None:
            gdict = {}
        self.gdict = gdict

    def bfs(graph, startnode):
        # Track the visited and unvisited nodes using queue
        seen, queue = set([startnode]), collections.deque([startnode])
        while queue:
            vertex = queue.popleft()
            marked(vertex)
            for node in graph[vertex]:
                if node not in seen:
                    seen.add(node)
                    queue.append(node)

    def marked(n):
        print(n)

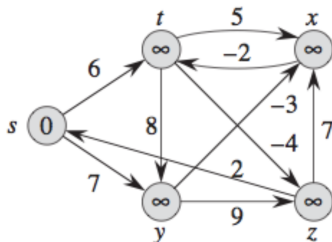
# The graph dictionary
gdict = { "a" : set(["b", "c"]),
          "b" : set(["a", "d"]),
          "c" : set(["a", "d"]),
          "d" : set(["e"]),
          "e" : set(["a"])
        }

bfs(gdict, "a")
```

Structures de données avancées: graphes

Parcours de graphe

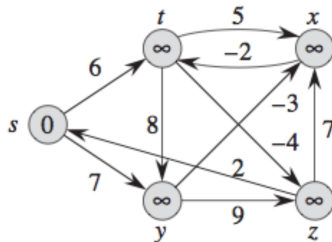
Application du parcours en largeur : recherche de plus court chemin ($s \rightarrow z$) ?



Structures de données avancées: graphes

Parcours de graphe

Application du parcours en largeur : recherche de plus court chemin ($s \rightarrow z$) ?



Approches (naïve) :

1. BFS avec minimum local (glouton)
2. BFS avec minimum global (programmation dynamique)

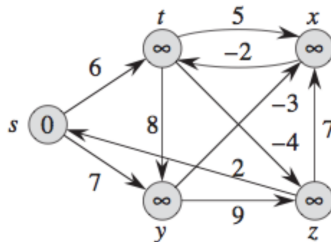
Structures de données avancées: graphes

Parcours de graphe

Algorithme de Bellman-Ford (programmation dynamique)

Algorithme 1 Bellman-Ford(G, w, s)

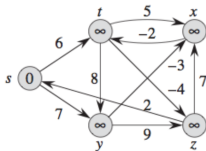
```
1: pour tout sommet  $v \in V$  faire // Initialisation
2:    $d[v] \leftarrow \infty$ ,  $\pi[v] \leftarrow \text{NIL}$ 
3:  $d[s] \leftarrow 0$ 
4: pour  $i$  de 1 à  $|V| - 1$  faire
5:   pour tout arc  $(u, v) \in E$  faire // relâchement de l'arc  $(u, v)$ 
6:     si  $d[v] > d[u] + w(u, v)$  alors
7:        $d[v] \leftarrow d[u] + w(u, v)$ ,  $\pi[v] \leftarrow u$ 
8:   pour tout arc  $(u, v) \in E$  faire // détection des circuits négatifs
9:     si  $d[v] > d[u] + w(u, v)$  alors
10:      retourner Faux
11: retourner Vrai
```



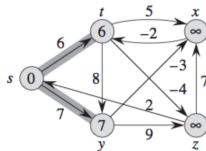
Structures de données avancées: graphes

Parcours de graphe

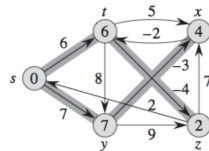
Algorithme de Bellman-Ford



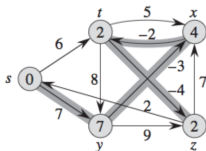
(a)



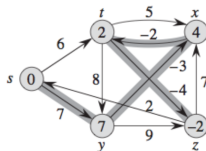
(b)



(c)



(d)



(e)

- Complexité $O(VE)$ car initialisation en $O(V)$, relâchement $O(E)$, et recherche circuit $O(E)$. Fonctionne avec poids de noeuds négatifs.

Structures de données avancées: graphes

Parcours de graphe

Algorithme de Bellman-Ford

```
def BellmanFord(self, src):  
  
    # Distances infinies  
    dist = [float("Inf")] * self.V  
    dist[src] = 0  
  
    # Relache les sommets - 1  
    for i in range(self.V - 1):  
  
        # Met a jour noeud et parents  
        for u, v, w in self.graph:  
            if dist[u] != float("Inf") and dist[u] + w < dist[v]:  
                dist[v] = dist[u] + w  
  
    # Verifie si cycle  
    for u, v, w in self.graph:  
        if dist[u] != float("Inf") and dist[u] + w < dist[v]:  
            print "Le graphe contient des cycles négatifs"  
            return
```

Structures de données avancées: graphes

Parcours de graphe

Algorithme de Dijkstra (intuition)

- ▶ Objectif connaître les plus courts chemins entre S sources et les nœuds du graphe accessibles depuis S
- ▶ Construction incrémentale et gloutonne d'un ensemble de nœuds E parcourus accessibles depuis sommet initial S
- ▶ Initialisation : E_0 liste vide et $G = \{S\}$
- ▶ Passage à l'étape suivante :
 - ▶ $E_{i+1} = E_i \cup \{ \text{nœud de } G \text{ hors de } E_i \text{ le plus proche de } S \text{ en empruntant un chemin qui ne traverse que les nœuds de } E_i \}$
 - ▶ Les sommets entrant dans E par ordre croissant de distance à S

Complexité dépend de l'implémentation de $O(V^2)$ à $O(E \log(V))$ si utilise files.

Structures de données avancées: graphes

Parcours de graphe

Algorithme de Dijkstra :

```
def dijkstra(graph, initial):
    visited = {initial: 0}
    path = {}

    nodes = set(graph.nodes)

    while nodes:
        min_node = None
        for node in nodes:
            if node in visited:
                if min_node is None:
                    min_node = node
                elif visited[node] < visited[min_node]:
                    min_node = node

        if min_node is None:
            break

        nodes.remove(min_node)
        current_weight = visited[min_node]

        for edge in graph.edges[min_node]:
            weight = current_weight + graph.distance[(min_node, edge)]
            if edge not in visited or weight < visited[edge]:
                visited[edge] = weight
                path[edge] = min_node

    return visited, path
```

Structures de données avancées: graphes

Parcours de graphe

Algorithme de Dijkstra :

Algorithme 7: Algorithme de Dijkstra

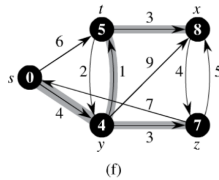
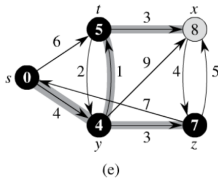
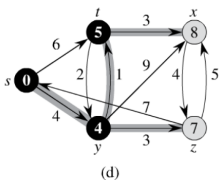
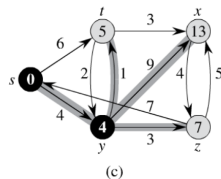
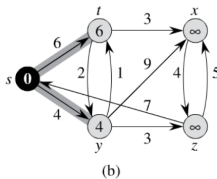
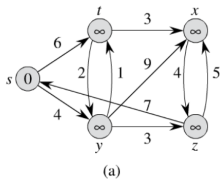
Données : Un graphe orienté pondéré $G = (X, A, W)$ et un sommet $s \in X$
Résultat : Le plus court chemin de s vers tous les autres sommets de G
// V : Tableau stockant les étiquettes des sommets de G

```
1 Initialiser  $V$  à  $+\infty$ 
2  $V[s] = 0$ 
  //  $P$  : Tableau permettant de retrouver la composition des chemins
3 Initialiser  $P$  à 0
4  $P[s] = s$ 
5 répéter
    // Recherche du sommet  $x$  non fixé de plus petite étiquette
6    $V_{min} = +\infty$ 
7   pour  $y$  allant de 1 à  $N$  faire
8     si  $y$  non marqué et  $V[y] < V_{min}$  alors
9        $x \leftarrow y$ 
10       $V_{min} \leftarrow V[y]$ 
    // Mise à jour des successeurs non fixés de  $x$ 
11   si  $V_{min} < +\infty$  alors
12     Marquer  $x$ 
13     pour tout successeur  $y$  de  $x$  faire
14       si  $y$  non marqué et  $V[x] + W[x, y] < V[y]$  alors
15          $V[y] = V[x] + W[x, y]$ 
16          $P[y] = x$ 
17 jusqu'à  $V_{min} = +\infty$ 
```

Structures de données avancées: graphes

Parcours de graphe

Algorithme de Dijkstra :



- Complexité $O(|E| + |V|\log|V|)$ si implémenté avec file de priorité.

Structures de données avancées: graphes

Parcours de graphe

Algorithme de Dijkstra (avec file de priorité) :

```
from queue import PriorityQueue

class Graph:

    def __init__(self, num_of_vertices):
        self.v = num_of_vertices
        self.edges = [[-1 for i in range(num_of_vertices)] for j in range(num_of_vertices)]
        self.visited = []

    def add_edge(self, u, v, weight):
        self.edges[u][v] = weight
        self.edges[v][u] = weight

    def dijkstra(self, start_vertex):
        D = {v: float('inf') for v in range(self.v)}
        D[start_vertex] = 0

        pq = PriorityQueue()
        pq.put((0, start_vertex))

        while not pq.empty():
            (dist, current_vertex) = pq.get()
            self.visited.append(current_vertex)

            for neighbor in range(self.v):
                if self.edges[current_vertex][neighbor] != -1:
                    distance = self.edges[current_vertex][neighbor]
                    if neighbor not in self.visited:
                        old_cost = D[neighbor]
                        new_cost = D[current_vertex] + distance
                        if new_cost < old_cost:
                            pq.put((new_cost, neighbor))
                            D[neighbor] = new_cost

        return D
```


Structures de données avancées: graphes

Parcours de graphe

Algorithme de Dijkstra (avec file de priorité) :

```
g = Graph(5)
g.add_edge(0, 1, 6)
g.add_edge(0, 2, 4)
g.add_edge(1, 2, 2)
g.add_edge(1, 3, 3)
g.add_edge(2, 1, 1)
g.add_edge(2, 3, 9)
g.add_edge(3, 4, 4)
g.add_edge(4, 3, 5)
g.add_edge(4, 0, 7)

D = g.dijkstra(0)

for vertex in range(len(D)):
    print("La distance du sommet 0 au sommet", vertex, "est de", D[vertex])
```

Structures de données avancées: graphes

Parcours de graphe

Bilan des parcours :

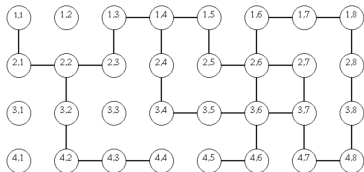
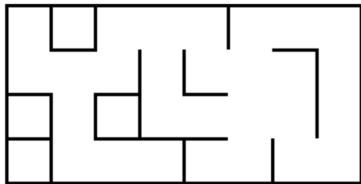
- ▶ Principe de minimiser un coût (sous-probleme optimal)
- ▶ Principe des algorithmes (Bellman-Ford, Dijkstra, Floyd-Warshall) est de sur-estimer le poids des sommets et d'en ajuster le coût avec une méthode de *relâchement*.
- ▶ L'algorithme de Bellman-Ford est proche de celui de Dijkstra. On y retrouve la notion de relaxation : $d(j) \rightarrow \min(d(j), d(x) + G(x, j))$.
- ▶ Dijkstra ne tolère pas les coûts négatifs et utilise une file de priorités pour traiter les arêtes dans le bon ordre et ne relaxer qu'une fois chaque arête.
- ▶ Bellman-Ford traite les arêtes dans un ordre arbitraire. Il tolère les coûts négatifs.
- ▶ Dijkstra avec graphe de coût 1 s'apparente au parcours en largeur (la file d'attente devient une pile).

Autres algorithmes : Floyd-Warshall, recherche bi-directionnelle, ..

Structures de données avancées: graphes

Parcours d'un labyrinthe

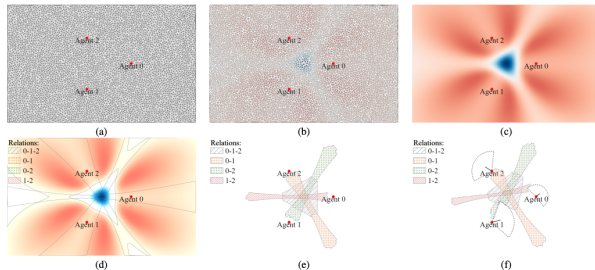
Application du parcours : sortir d'un labyrinthe peut être reformulé sous forme de parcours d'un graphe



Structures de données avancées: graphes

Parcours d'un labyrinthe

Dans le cas d'environnement continu, création d'un *pathfinding grid* :



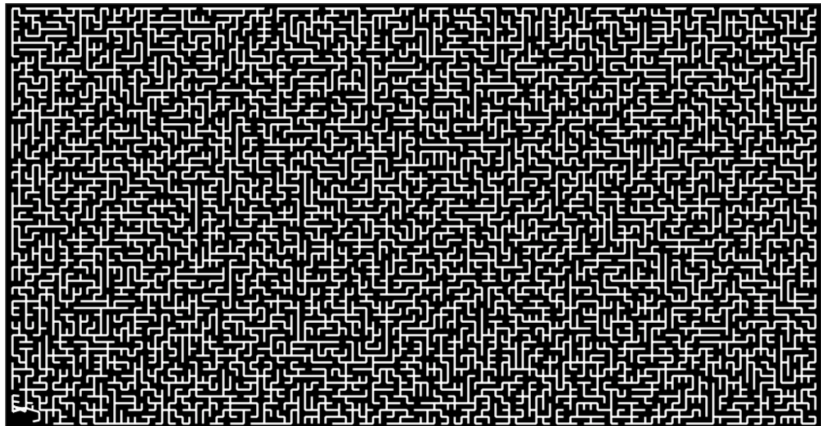
Jules Allegre (ECL'19), Romain Vuillemot. Visualizing and Analyzing Disputed Areas in Soccer. Visualization in Data Science, Oct 2020, Salt Lake City (Virtual Conference), United States.

<https://hal.archives-ouvertes.fr/hal-02951454/document>

<http://theory.stanford.edu/~amitp/GameProgramming/>

[MapRepresentations.html](http://theory.stanford.edu/~amitp/GameProgramming/MapRepresentations.html)

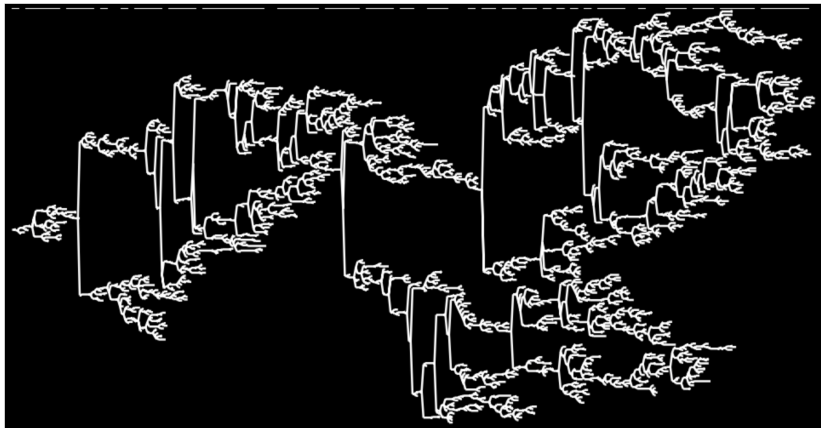
Structures de données avancées: graphes
Parcours d'un labyrinthe



<https://bl.ocks.org/mbostock/061b3929ba0f3964d335>

Structures de données avancées: graphes

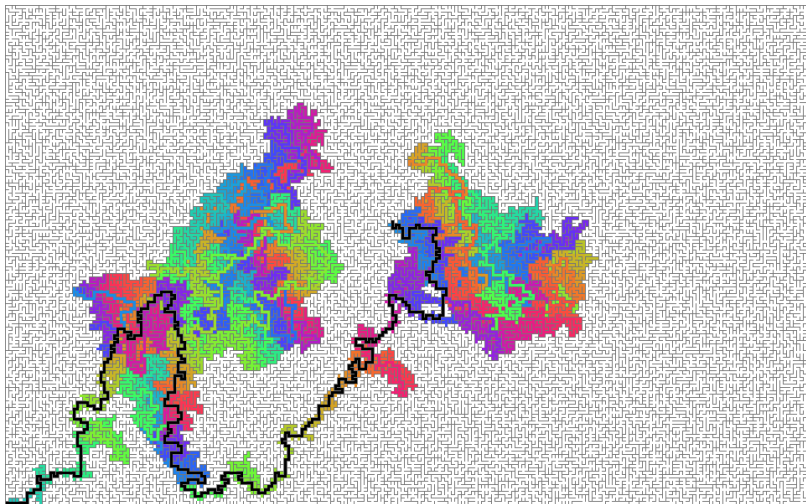
Parcours d'un labyrinthe



<https://bl.ocks.org/mbostock/061b3929ba0f3964d335>

Structures de données avancées: graphes

Parcours d'un labyrinthe



<https://beta.observablehq.com/@mbostock/best-first-search>

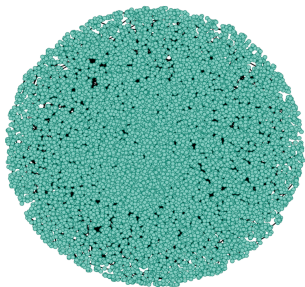
Structures de données avancées: graphes

Parcours de graphe

- ▶ GraphViz <https://graphviz.org/>
- ▶ Bibliothèques Python
- ▶ NetworkX <https://networkx.github.io/>
- ▶ Bokeh https://bokeh.pydata.org/en/latest/docs/user_guide/graph.html
- ▶ Altair <https://altair-viz.github.io/>
- ▶ D3.JS <https://d3js.org/>
- ▶ Observable <https://beta.observablehq.com/>

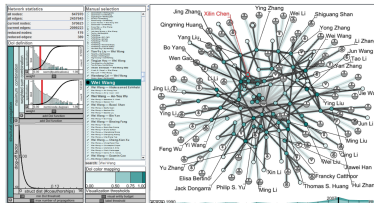
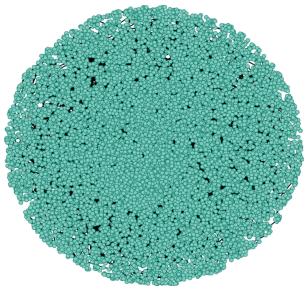
Structures de données avancées: graphes

Dessin d'arbres et graphes



Structures de données avancées: graphes

Dessin d'arbres et graphes

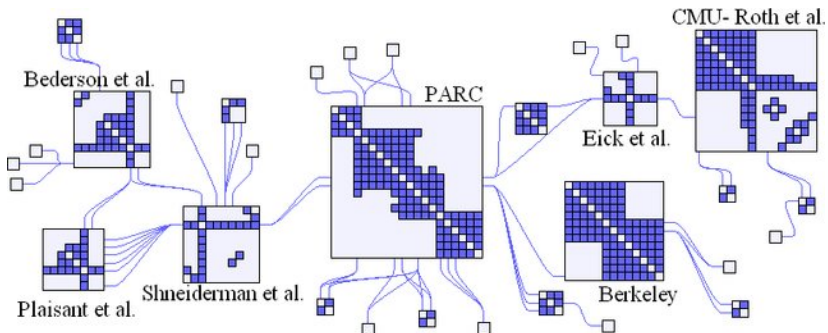


http://vcg.informatik.uni-rostock.de/~hadlak/pub_files/2014Abello-DoiGraph-Talk.pdf

Structures de données avancées: graphes

Dessin d'arbres et graphes

Les graphes peuvent être clusterisés afin d'avoir une représentation adaptée au cluster tout en gardant la représentation nœud-lien.

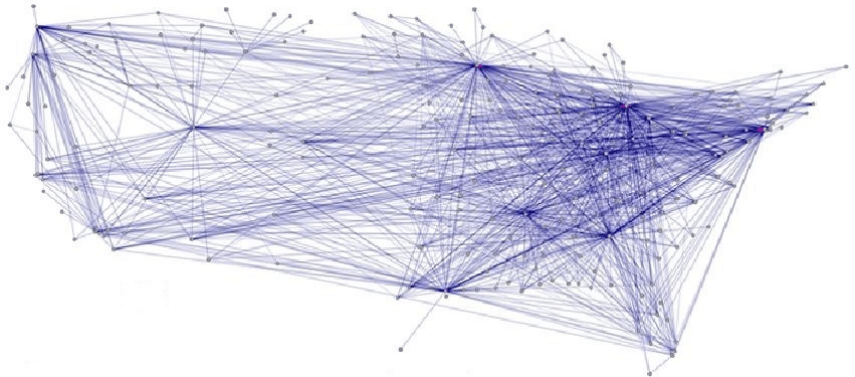


<https://aviz.fr/Research/Nodetrix>

Structures de données avancées: graphes

Dessin d'arbres et graphes

Les données brutes de graphe sous forme nœud lien sont difficiles à visualiser (pour identifier flots, ..)



Structures de données avancées: graphes

Dessin d'arbres et graphes

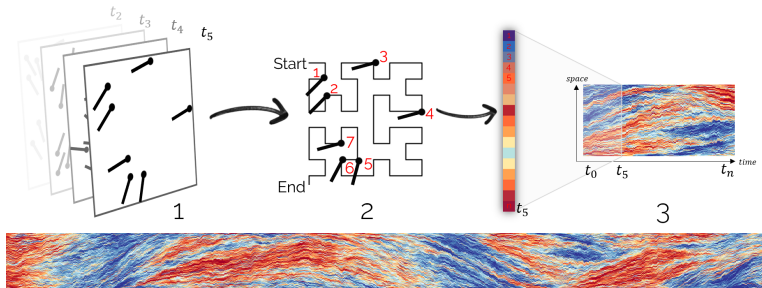
Les techniques de *edge bundling* permettent de faire émerger des routes et refléter la structure hiérarchique du réseau.



Structures de données avancées: graphes

Dessin d'arbres et graphes

Utilisation d'un index spatial pour linéariser un graphe dynamique



<https://github.com/jbuchmueller/motionrugs>

Structures de données avancées: graphes

Dessin d'arbres et graphes

Challenges liés à la visualisation de graphes

- ▶ Méthode générique quel que soit le graphe (taille, distribution, ..)
- ▶ Graphes dynamiques, dont les attributs et la topologie peuvent changer au fil du temps
- ▶ La visualisation est relative aux tâches que l'utilisateur doit réaliser avec : identifier une communauté, suivre un chemin, etc.
- ▶ Comme il s'agit d'une structure de donnée *abstraite* l'ordre des nœuds joue un rôle important
- ▶ Passage à l'échelle (en nombre de noeuds et d'arêtes)
- ▶ ...

<https://hal.inria.fr/hal-00712779/document>