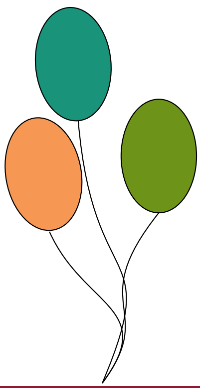
SWERC NoteBook

Équipe SaintGermainDesPrés

Mathilde BONIN, Eyal COHEN, Hugo DEMARET

Avril 2022

Ensemble d'algorithmes et techniques de programmation



Université Paris Cité
UFR de Mathématiques-Informatique
2021-2022

1 Configuration

1.1 C/C++

2 Chaînes de caractères

3 Séquences

4 Parcours de graphes

4.1 DFS - Depth First Search

```
#version iterative pour eviter la recursion limit de python
def dfs_iterative(graph,start,seen):
    seen[start] = True
    to_visit = [start]
    while to_visit:
        node = to_visit.pop()
        for neighbour in graph[node]:
            if not seen[neighbour]:
                seen[neighbour] = True
                to_visit.append(neighbour)
```

4.2 BFS - Breadth First Search

```
from collections import deque
def bfs(graph, start=0):
    to_visit = deque()
    dist = [float('inf')] * len(graph)
    prec = [none] * len(graph)
    dist[start] = 0
    to_visit.appendleft(start)
    while to_visit: #evalue a faux si vide
        node = to_visit.pop()
        for neighbour in graph[node]:
            if dist[neighbour] == float('inf'):
                dist[neighbour] = dist[node] + 1
                prec[neighbour] = node
                to_visit.appendleft(neighbour)
    return dist, prec
```

4.3 Topological Sort

```
def topological_order(graph):
    V = range(len(graph))
    indeg = [0 for _ in V]
    for node in V:          # compute indegree
        for neighbor in graph[node]:
            indeg[neighbor] += 1
    Q = [node for node in V if indeg[node] == 0]
    order = []
    while Q:
        node = Q.pop()      # node without incoming arrows
        order.append(node)
        for neighbor in graph[node]:
            indeg[neighbor] -= 1
            if indeg[neighbor] == 0:
                Q.append(neighbor)
    return order
```

4.4 Composantes connexes

4.5 Composantes bi-connexe

```
def cut_nodes_edges(graph):
    n = len(graph)
    time = 0
    num = [None] * n
    low = [n] * n
    parent = [None] * n     # parent[v] = None if root else parent of v
    critical_children = [0] * n # cc[u] = #{children v | low[v] >= num[u]}
    times_seen = [-1] * n
    for start in range(n):
        if times_seen[start] == -1:           # init DFS path
            times_seen[start] = 0
            to_visit = [start]
            while to_visit:
                node = to_visit[-1]
                if times_seen[node] == 0:     # start processing
                    num[node] = time
                    time += 1
                    low[node] = float('inf')
                children = graph[node]
                if times_seen[node] == len(children): # end processing
                    to_visit.pop()
                    up = parent[node]         # propagate low to parent
                    if up is not None:
                        low[up] = min(low[up], low[node])
                        if low[node] >= num[up]:
                            critical_children[up] += 1
                else:
                    child = children[times_seen[node]] # next arrow
                    times_seen[node] += 1
                    if times_seen[child] == -1: # not visited yet
                        parent[child] = node   # link arrow
                        times_seen[child] = 0
                        to_visit.append(child) # (below) back arrow
                    elif num[child] < num[node] and parent[node] != child:
                        low[node] = min(low[node], num[child])
    cut_edges = []
    cut_nodes = []                            # extract solution
    for node in range(n):
        if parent[node] is None:              # characteristics
            if critical_children[node] >= 2:
                cut_nodes.append(node)
        else:                                 # internal nodes
            if critical_children[node] >= 1:
                cut_nodes.append(node)
            if low[node] >= num[node]:
                cut_edges.append((parent[node], node))
    return cut_nodes, cut_edges
```

4.6 Composantes fortement connexe

4.6.1 Kosaraju

```
def kosaraju_dfs(graph,nodes,order,sccp):
    times_seen = [-1] * len(graph)
    for start in nodes:
        if times_seen[start] == -1:
            to_visit = [start]
            times_seen[start] = 0
            sccp.append([start])
            while to_visit:
                node = to_visit[-1]
```

```
            children = graph[node]
            if times_seen[node] == len(children):
                to_visit.pop()
                order.append(node)
            else:
                child = children[times_seen[node]]
                times_seen[node] += 1
                if times_seen[child] == -1:
                    times_seen[child] = 0
                    to_visit.append(child)
                    sccp[-1].append(child)
def reverse(graph):
    rev_graph = [[] for node in graph]
    for node in range(len(graph)):
        for neighbour in graph[node]:
            rev_graph[neighbour].append(node)
    return rev_graph
def kosaraju(graph):
    n = len(graph)
    order = []
    sccp = []
    kosaraju_dfs(graph, range(n), order, [])
    kosaraju_dfs(reverse(graph),order[::-1], [], sccp)
    return sccp[::-1]
```

4.7 2-SAT

```
def vertex(lit):
    if lit > 0:
        return 2 * (lit - 1)
    else:
        return 2 * (-lit -1) +1
def two_sat(formula):
    n = max(abs(clause[p]) for p in (0,1) for clause in formula)
    graph = [[] for node in range(2*n)]
    for x,y in formula:
        graph[vertex(-x)].append(vertex(y))
        graph[vertex(-y)].append(vertex(x))
    sccp = kosaraju(graph)
    comp_id = [None] * (2*n)
    affectations = [None] * (2*n)
    for component in sccp:
        rep = min(component)
        for vtx in component:
            comp_id[vtx] = rep
            if affectations[vtx] == None:
                affectations[vtx] = True
                affectations[vtx ^ 1] = False
    for i in range(n):
        if comp_id[2*i] == comp_id[2*i+1]:
            return None
    return affectations[::2]
```

4.8 Postier Chinois

4.9 Chemin eulérien

4.9.1 Dirigé

```
def eulerian_tour_directed(graph):
    P = []
    Q = [0]
    R = []
```

```
    next = [0] * len(graph)
    while Q:
        node = Q.pop()
        P.append(node)
        while next[node] < len(graph[node]):
            neighbour = graph[node][next[node]]
            next[node] += 1
            R.append(neighbour)
            node = neighbour
        while R:
            Q.append(R.pop())
    return P
```

4.9.2 Non Dirigé

```
def eulerian_tour_undirected(graph):
    P = []
    Q = [0]
    R = []
    next = [0] * len(graph)
    seen = [set() for _ in graph]
    while Q:
        node = Q.pop()
        P.append(node)
        while next[node] < len(graph[node]):
            neighbour = graph[node][next[node]]
            next[node] += 1
            if neighbour not in seen[node]:
                seen[neighbour].add(node)
                R.append(neighbour)
                node = neighbour
        while R:
            Q.append(R.pop())
    return P
```

4.10 Chemin le plus court

4.10.1 Poids positif ou nul - Dijkstra

```
from heapq import heappop, heappush
def dijkstra(graph, weight, source=0, target=None):
    """single source shortest paths by Dijkstra
       :complexity: O(|V| + |E|log|V|)"""
    n = len(graph)
    assert all(weight[u][v] >= 0 for u in range(n) for v in graph[u])
    prec = [None] * n
    black = [False] * n
    dist = [float('inf')] * n
    dist[source] = 0
    heap = [(0, source)]
    while heap:
        dist_node, node = heappop(heap)    # Closest node from source
        if not black[node]:
            black[node] = True
            if node == target:
                break
            for neighbor in graph[node]:
                dist_neighbor = dist_node + weight[node][neighbor]
                if dist_neighbor < dist[neighbor]:
                    dist[neighbor] = dist_neighbor
                    prec[neighbor] = node
                    heappush(heap, (dist_neighbor, neighbor))
    return dist, prec
```

```python
def dijkstra_update_heap(graph, weight, source=0, target=None):
    """single source shortest paths by Dijkstra
       :complexity: O(|V| + |E|log|V|)"""
    n = len(graph)
    assert all(weight[u][v] >= 0 for u in range(n) for v in graph[u])
    prec = [None] * n
    dist = [float('inf')] * n
    dist[source] = 0
    heap = OurHeap([(dist[node], node) for node in range(n)])
    while heap:
        dist_node, node = heap.pop()    # Closest node from source
        if node == target:
            break
        for neighbor in graph[node]:
            old = dist[neighbor]
            new = dist_node + weight[node][neighbor]
            if new < old:
                dist[neighbor] = new
                prec[neighbor] = node
                heap.update((old, neighbor), (new, neighbor))
    return dist, prec
```

4.10.2 Poids arbitraire - Bellman-Ford

```python

def bellman_ford2(graph, weight, source):
    """    :complexity: O(|V|*|E|)"""
    n = len(graph)
    dist = [float('inf')] * n
    prec = [None] * n
    dist[source] = 0

    def relax():
        for nb_iterations in range(n-1):
            for node in range(n):
                for neighbor in graph[node]:
                    alt = dist[node] + weight[node][neighbor]
                    if alt < dist[neighbor]:
                        dist[neighbor] = alt
                        prec[neighbor] = node
    relax()
    intermediate = dist[:] # is fixpoint in absence of neg cycles
    relax()
    for node in range(n):
        if dist[node] < intermediate[node]:
            dist[node] = float('-inf')
    return dist, prec, min(dist) == float('-inf')
```

4.10.3 Floyd-Warshall

```python
def floyd_warshall(weight):
    """"O(|V|^3)"""
    for k, Wk in enumerate(weight):
        for _, Wu in enumerate(weight):
            for v, Wuv in enumerate(Wu):
                alt = Wu[k] + Wk[v]
                if alt < Wuv:
                    Wu[v] = alt
    for v, Wv in enumerate(weight):
        if Wv[v] < 0:    # negative cycle found
            return True
    return False
```

5 Points et polygones

5.1 Points

5.1.1 Points

```
point = [x,y]
```

5.1.2 Cross-product

```
def cross_product(p1, p2):
  return p1[0] * p2[1] - p2[0] * p1[1]
```

5.1.3 Direction

```
def left_turn(a,b,c):
  return (a[0]-c[0]) * (b[1]-c[1]) - (a[1]-c[1]) * (b[0]-c[0]) > 0
  # If floats are used, instead of 0 test if in [0-10E-7,0+10E-7]
```

5.2 Enveloppe convexe

Complexité : $\mathcal{O}(n\log(n))$

```
def andrew(S):
    S.sort()
    top = []
    bot = []
    for p in S:
        while len(top) >= 2 and not left_turn(p,top[-1],top[-2]):
            top.pop()
        top.append(p)
        while len(bot) >= 2 and not left_turn(bot[-2],bot[-1],p):
            bot.pop()
        bot.append(p)
    return bot[:-1] + top[:0:-1]
```

5.3 Aire d'un polygone

Uniquement pour les polygones simples. Réduire à des composantes simples sinon. Voir partie Mathématiques.

```
def area(p):
    A = 0
    for i in range(len(p)):
        A += p[i-1][0] * p[i][1] - p[i][0] * p[i-1][1]
    return A/2
```

5.4 Polygone simple

```
def is_simple(polygon):
    """complexity: O(n log n) for n=len(polygon)"""
    n = len(polygon)
    order = list(range(n))
    order.sort(key=lambda i: polygon[i])     # lexicographic order
    rank_to_y = list(set(p[1] for p in polygon))
    rank_to_y.sort()
    y_to_rank = {y: rank for rank, y in enumerate(rank_to_y)}
    S = RangeMinQuery([0] * len(rank_to_y))  # sweep structure
    last_y = None
    for i in order:
        x, y = polygon[i]
        rank = y_to_rank[y]
        right_x = max(polygon[i - 1][0], polygon[(i + 1) % n][0])
```

```
        left = x < right_x
        below_y = min(polygon[i - 1][1], polygon[(i + 1) % n][1])
        high = y > below_y
        if left:                    # y does not need to be in S yet
            if S[rank]:
                return False        # two horizontal segments intersect
            S[rank] = -1            # add y to S
        else:
            S[rank] = 0             # remove y from S
        if high:
            lo = y_to_rank[below_y] # check S between [lo + 1, rank - 1]
            if (below_y != last_y or last_y == y or
                    rank - lo >= 2 and S.range_min(lo + 1, rank)):
                return False        # horiz. & vert. segments intersect
        last_y = y                  # remember for next iteration
    return True
```

5.5 Paire de points les plus proches

6 Ensembles

6.1 Rendu de monnaie

Problème NP-Complet.

```
def coin(x, R):
    b = [False] * (R+1)
    b[0] = True
    for xi in x:
        for s in range(xi, R+1):
            b[s] |= b[s - xi]
    return b[R]
```

6.2 Sac à dos

Problème NP-Complet.

```
def knapsack(p, v, cmax):
    n = len(p)
    Opt = [[0] * (cmax + 1) for _ in range(n+1)]
    Sel = [[False] * (cmax + 1) for _ in range(n+1)]
    #cas de base
    for cap in range(p[0], cmax +1):
        Opt[0][cap] = v[0]
        Sel[0][cap] = True
    # cas d'induction
    for i in range(1,n):
        for cap in range(cmax+1):
            if cap >= p[i] and Opt[i-1][cap - p[i]] + v[i] > Opt[i-1][cap]:
                Opt[i][cap] = Opt[i-1][cap-p[i]] + v[i]
                Sel[i][cap] = True
            else:
                Opt[i][cap] = Opt[i-1][cap]
                Sel[i][cap] = False
    cap = cmax
    sol = []
    for i in range(n-1, -1, -1):
        if Sel[i][cap]:
            sol.append(i)
            cap -= p[i]
    return (Opt[n-1][cmax], sol)
```

6.3 k-somme

7 Calculs

7.1 PGCD

```python
def pgcd(a,b):
    return a if b == 0 else pgcd(b,a%b)
```

7.2 Coefficients de Bézout

```python
def bezout(a,b):
    if b == 0:
        return (1,0)
    else:
        u,v = bezout(b,a%b)
        return (v, u - (a//b) *v)
def inv(a,p):
    return bezout(a,p)[0]%p
```

7.3 Coefficients binomiaux

```python
def binom(n,k):
    prod = 1
    for i in range(k):
        prod = (prod * (n-i)) // (i+1)
    return prod
def binom_modulo(n,k,p):
    prod = 1
    for i in range(k):
        prod = (prod * (n-1) * inv(i+1,p)) %p
    return prod
```

7.4 Inverse

```python
def inv(a,p):
    return bezout(a,p)[0] %p
```

8 Mathématiques

8.1 Géométrie

8.1.1 3D

- Sphère : Volume : $\frac{4}{3}\pi r^3$ — Surface : $4\pi r^2$
- Cylindre droit : Volume $\pi r^2 h$ — Surface : $2\pi r(r+h)$
- Cone circulaire droit : Volume $\frac{1}{3}\pi r^2 h$ — Surface : $\pi r(r+s)$
- Prisme triangulaire : Volume Al ou $\frac{1}{2}bhl$ — Surface : $bh + 2ls + lb$
- Prisme : Volume Ah — Surface : $2A + (h \times p)$
- Pyramide : Volume : $\frac{1}{3}Ah$
- Tétrahèdre : Volume : $\frac{b^3}{6\sqrt[2]{2}}$ — Surface : $\sqrt[2]{3}b^2$
- Pyramide carré : Volume : $\frac{1}{3}s^2 \times h$ — Surface : $s^2 + 2sh$
- Cuboide : Volume : $l \times w \times h$ — Surface : $2lh + 2lw + 2wh$ (Cube : $6s^2$)

8.1.2 2D

- Polygone simple : Aire : $A = \frac{1}{2}\sum_{i=0}^{n-1}(x_i x_{i+1} - x_{i+1} y_i)$
- Cercle : Aire : πr^2 — Périmètre : $2 \times \pi \times r$
- Losange : Aire : $\frac{D \times d}{2}$
- Trapèze : Aire : $\frac{(B+b) \times h}{2}$
- Parralélogramme : Aire : $B \times h$

8.1.3 Points entiers dans un polygone

Sur le contour :
Dans le polygone :
Théorème de Pick : $P = n_i + \frac{n_b}{2} - 1$

8.1.4 Théorème de la galerie d'art

Pour garder un polygone simple à n sommets, $\lfloor \frac{n}{3} \rfloor$ gardiens suffisent.

8.2 Approximations

8.2.1 Méthode de Newton

$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}$ — $f(x) \approx f(x_0) + f'(x_0)(x - x_0)$

8.2.2 Méthode de la sécante

Cette méthode est à appliquer quand le calcul de la dérivée est couteux $\frac{f(x_k) - f(x_{k-1})}{x_k - x_{k-1}}$

8.2.3 Plus forte pente — Méthode du gradient

Cette méthode peut être assez couteuse (ZigZag)
Algorithme du gradient — On se donne un point initial $x_0 \in \mathbb{E}$ et un seuil de tolérance $\epsilon \geq 0$. L'algorithme du gradient donne une suite d'itérés $x_1, x_2 ... \in \mathbb{E}$, jusqu'à ce qu'un test d'arrêt soit satisfait. Il passe de x_k à x_{k+1} par les étapes suivantes :

- 1. *Simulation* : Calcul de $\nabla f(x_k)$
- 2. *Test d'arrêt* : Si $\|\nabla f(x_k)\| \leq \epsilon$, arrêt
- 3. *Calcul du pas* : $\alpha_k > 0$ par un règle de recherche linéaire sur f en x_k le long de la direction $-\nabla f(x_k)$
- 4. *Nouvel itéré* : $x_{k+1} = x_k - \alpha_k \nabla f(x_k)$

8.3 Probabilités et Statistiques

8.3.1 Lois de probabilités

8.3.2 Techniques statistiques

9 Techniques de programmation

9.1 Programmation dynamique

Résoudre le problème en le divisant en sous-problèmes, résoudre les sous-problèmes, stocker les résultats intermédiaires ("mémoisation")

9.2 Diviser pour régner

Diviser un problème en sous-problèmes; Résoudre les sous-problèmes; Combiner : calculer la solution grâce aux solutions des sous-problèmes.

9.3 Floyd's Hare and Tortoise

L'objectif de cette méthode est de détecter des cycles. L'idée est de parcourir la liste chaînée avec deux pointeurs : un lent (tortoise) et un deux fois plus rapide (hare). Si les deux pointeurs s'intersectent, il y a un cycle.