Formulaire BAPC 2013

Team UCooL

Auteurs : François Aubry, Guillaume Derval, Benoît Legat, Anthony Gégo.

Table des matières

1 Remarques

1.1 Attention !

1. Lire **TOUS** les énoncés avant de commencer la moindre implémentation
2. Faire attention au copier-coller bête et méchant.
3. Surveiller les overflow. Parfois, un long peux régler pas mal de problèmes

1.2 Opérations sur les bits

1. Vérification parité de n : `(n & 1) == 0`
2. 2^n : `1 << n.`
3. Tester si le $i$ème bit de n est 0 : `(n & 1 << i) != 0`
4. Mettre le $i$ème bit de n à 0 : `n &= ~(1 << i)`
5. Mettre le $i$ème bit de n à 1 : `n |= (1 << i)`
6. Union : `a | b`
7. Intersection : `a & b`
8. Soustraction bits : `a & ~b`
9. Vérifier si n est une puissance de 2 : `(x & (x−1) == 0)`
10. Passage au négatif : `0 x7fffffff ^n`

2 Graphes

2.1 Bases

- Adjacency matrix : $A[i][j] = 1$ if i is connected to j and 0 otherwise
- Undirected graph : $A[i][j] = A[j][i]$ for all i, j (i.e. $A = A^T$)
- Adjacency list : `LinkedList<Integer>[] g;` $g[i]$ stores all neightboors of i
- Useful alternatives :

```
HashSet<Integer>[] g; // for edge deletion
HashMap<Integer, Integer>[] g; // for weighted
    graphs
```

- Classes de base (à adapter, les notations changent)

```
class Vertex implements Comparable<Vertex>
{
  int i; long d;
  public Vertex(int i, long d)
  {
    this.i = i; this.d = d;
  }
  public int compareTo(Vertex o)
  {
    return d < o.d ? -1 : d > o.d ? 1 : 0;
  }
}

class Edge implements Comparable<Edge>
{
  int o, d, w;
  public Edge(int o, int d, int w)
  {
    this.o = o; this.d = d; this.w = w;
  }
  public int compareTo(Edge o)
  {
    return w - o.w;
  }
}
```

2.2 BFS (Parcours en largeur)

Calcule à partir d'un graphe g et d'un noeud v un vecteur d t.q. $d[u]$ réprésente le nombre d'arète min. à parcourir pour arrive au noeud u.

$d[v] = 0$, $d[u] = \infty$ si u injoignable. Si $(u, w) \in E$ et $d[u]$ connu et $d[w]$ inconnu, alors $d[w] = d[u] + 1$.

```
int[] bfsVisit(LinkedList<Integer>[] g, int v, int c
    []) //c is for connected components only
{
  Queue<Integer> Q = new LinkedList<Integer>();
  Q.add(v);
  int[] d = new int[g.length];
  c[v]=v; //for connected components
  Arrays.fill(d, Integer.MAX_VALUE);
  // set distance to origin to 0
  d[v] = 0;
  while(!Q.isEmpty())
  {
    int cur = Q.poll();
    // go over all neighboors of cur
    for(int u : g[cur])
    {
      // if u is unvisited
      if(d[u] == Integer.MAX_VALUE) //or c[u] == -1 if
    we calculate connected components
      {
        c[u] = v; //for connected components
        Q.add(u);
        // set the distance from v to u
        d[u] = d[cur] + 1;
      }
    }
  }
  return d;
}
```

2.2.1 Composantes connexes

```
int[] bfs(LinkedList<Integer>[] g)
{
  int[] c = new int[g.length];
  Arrays.fill(c, -1);
  for(int v = 0; v < g.length; v++)
    if(c[v] == -1)
      bfsVisit(g, v, c);
  return c;
}
```

2.2.2 Vérifier Biparticité (Bicolorabilité)

```
boolean isBipartite(LinkedList<Integer>[] g)
{
  int[] d = bfs(g);
  for(int u = 0; u < g.length; u++)
    for(Integer v: g[u])
      if((d[u]%2)!=(d[v]%2)) return false;
  return true;
}
```

2.3 DFS (Parcours en profondeur)

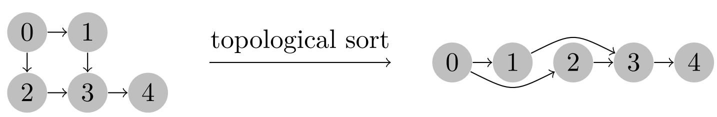
Soit = BFS avec *Stack* à la place de *Queue* ou implémentation récursive hyper-simple. Complexité $O(|V| + |E|)$

```
int UNVISITED = 0, OPEN = 1, CLOSED = 2;
boolean cycle; // true iff there is a cycle

void dfsVisit(LinkedList<Integer>[] g, int v,int[]
    label)
{
  label[v] = OPEN;
  for(int u : g[v])
  {
    if(label[u] == UNVISITED)
      dfsVisit(g, u, label);
    if(label[u] == OPEN)
      cycle = true;
  }
  label[v] = CLOSED;
}

void dfs(LinkedList<Integer>[] g)
{
  int[] label = new int[g.length];
  Arrays.fill(label, UNVISITED);
  cycle = false;
  for(int v = 0; v < g.length; v++)
    if(label[v] == UNVISITED)
      dfsVisit(g, v, label);
}
```

2.3.1 Ordre topologique



Le graphe doit être acyclique. On modifie légèrement DFS :

```
Stack<Integer> toposort; // add stack to global
    variables
/* ... */
void dfs(LinkedList<Integer>[] g)
{
  /* ... */
  toposort = new Stack<Integer>();
  for(int v = 0; v < g.length; v++) { /* ... */ }
}

void dfsVisit(LinkedList<Integer>[] g, int v,int[]
    label)
{
  /* ... */
  toposort.push(v); // push vertex when closing it
  label[v] = CLOSED;
}
```

2.3.2 Composantes fortement connectées

Calculer l'ordre topologique du graphe avec les arêtes inversées, puis exécuter un BFS dans l'ordre topologique (et sans repasser par un nœud déjà fait). Les nœuds parcourus à chaque execution du BFS sont fortement connectés.

```
int[] scc(LinkedList<Integer>[] g)
{
  // compute the reverse graph
  LinkedList<Integer>[] gt = transpose(g);
  // compute ordering
  dfs(gt);
  // !! last position will contain the number of scc's
  int[] scc = new int[g.length + 1];
  Arrays.fill(scc, -1);
  int nbComponents = 0;
  // simulate bfs loop but in toposort ordering
  while(!toposort.isEmpty())
  {
    int v = toposort.pop();
    if(scc[v] == -1)
    {
      nbComponents++;
      bfsVisit(g, v, scc);
    }
  }
  scc[g.length] = nbComponents;
  return scc;
}
```

2.4 Arbre de poids minimum (Prim)

On ajoute toujours l'arète de poids minimal parmit les noeuds déja visités.

```
double mst(LinkedList<Edge>[] g)
{
  boolean[] inTree = new boolean[g.length];
  PriorityQueue<Edge> PQ = new PriorityQueue<Edge>();
  // add 0 to the tree and initialize the priority
    queue
  inTree[0] = true;
  for(Edge e : g[0]) PQ.add(e);
  double weight = 0;
  int size = 1;
  while(size != g.length)
  {
    // poll the minimum weight edge in PQ
    Edge minE = PQ.poll();
    // if its endpoint in not in the tree, add it
    if(!inTree[minE.dest])
    {
      // add edge minE to the MST
      inTree[minE.dest] = true;
      weight += minE.w;
      size++;
      // add edge leading to new endpoints to the PQ
      for(Edge e : g[minE.dest])
        if(!inTree[e.dest]) PQ.add(e);
    }
  }
  return weight;
}
```

2.5 Dijksta

Plus court chemin d'un noeud v à tout les autres. Le graphe doit être sans cycles de poids négatif.

```
double[] dijkstra(LinkedList<Edge>[] g, int v)
{
  double[] d = new double[g.length];
  Arrays.fill(d, Double.POSITIVE_INFINITY);
  // initialize distance to v and the priority queue
  d[v] = 0;
  PriorityQueue<Edge> PQ = new PriorityQueue<Edge>();
  for(Edge e : g[v])
    PQ.add(e);
```

```
  while(!PQ.isEmpty())
  {
    // poll minimum edge from PQ
    Edge minE = PQ.poll();
    if(d[minE.dest] == Double.POSITIVE_INFINITY)
    {
      // set the distance to the new found endpoint
      d[minE.dest] = minE.w;
      for(Edge e : g[minE.dest])
      {
        // add to the queue all edges leaving the new
        // endpoint with the increased weight
        if(d[e.dest] == Double.POSITIVE_INFINITY)
          PQ.add(new Edge(e.orig, e.dest, e.w + d[e.
    orig]));
      }
    }
  }
  return d;
}
```

2.6 Bellman-Ford

Plus court chemin d'un noeud v à tout les autres. Le graphe peut avoir des cycles de poids négatif, mais alors l'algorithme ne retourne pas les chemins les plus courts, mais retourne l'existence de tels cycles.
$d[i][u]$ = shortest path from v to u with $\leq i$ edge
$d[0][v] = 0$
$d[0][u] = \infty \quad$ for $u \neq v$
$d[i][u] = \min\{d[i-1][u], \quad \min_{(s,u)\in E} d[i-1][s] + w(s,u)\}$
Si pas de cycle, la solution est dans $d[|V|-1]$. Si cycle il y a, $d[|V|-1] = d[V]$.
$O(|V||E|)$.

```
double[] bellmanFord(LinkedList<Edge>[] gt, int v)
{
  int n = gt.length;
  double[][] d = new double[n][n];
  for(int u = 0; u < n; u++)
    d[0][u] = u == v ? 0 : Double.POSITIVE_INFINITY;
  for(int i = 1; i < n; i++)
  {
    for(int u = 0; u < n; u++)
    {
      double min = d[i - 1][u];
      for(Edge e : gt[u])
        min = Math.min(min, d[i - 1][e.dest] + e.w);
      d[i][u] = min;
    }
  }
  return d[n - 1];
}
```

2.7 Floyd-Warshall

Plus court chemin de tout les noeuds à tout les autres. Prend en argument la matrice d'adjacence. $O(|V|^3)$ en temps et $O(|V|^2)$ en mémoire.
Le graphe contient des cycles de poids négatif ssi $result[v][v] < 0$.

```
double[][] floydWarshall(double[][] A)
{
  int n = A.length;
  // initialization: base case
  double[][] d = new double[n][n];
  for(int v = 0; v < n; v++)
    for(int u = 0; u < n; u++)
      d[v][u] = A[v][u];

  for(int k = 0; k < n; k++)
    for(int v = 0; v < n; v++)
      for(int u = 0; u < n; u++)
        d[v][u] = Math.min(d[v][u], d[v][k] + d[k][u])
  ;
```

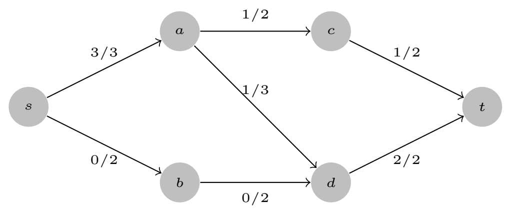
```
  return d;
}
```

2.8 Flux maximum

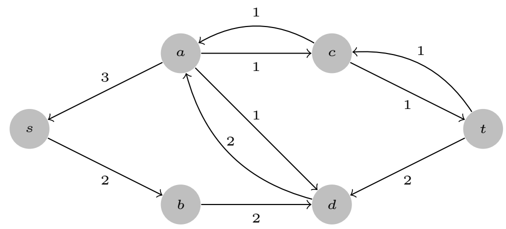
2.8.1 Bases

On cherche à calculer le flux maximum d'une source S à un puits T. Chaque arête à un débit maximum et un débit actuel (uniquement pendant la résolution). On construit le graphe résiduel comme sur les exemples.

flow graph



residual graph



L'algorithme de base fonctionne en cherchant un chemin de S à T dans le graphe résiduel.

2.8.2 Ford-Fulkerson

Si le chemin est cherché avec un DFS, la complexité est $O(|E|f^*)$ où f^* est le flux maximum. On préferera pour les problèmes l'algorithme avec un BFS (Edmonds-Karps).

2.8.3 Edmonds-Karps (BFS)

Chemin cherché avec un BFS. On a $O(|V||E|^2)$.

```
int maxFlow(HashMap<Integer, Integer>[] g, int s, int
    t)
{
  // output 0 for s = t (convention)
  if(s == t) return 0;
  // initialize maxflow
  int maxFlow = 0;
  // compute an augmenting path
  LinkedList<Edge> path = findAugmentingPath(g, s, t);
  // loop while augmenting paths exists and update g
  while(path != null)
  {
    int pathCapacity = applyPath(g, path);
    maxFlow += pathCapacity;
    path = findAugmentingPath(g, s, t);
  }
  return maxFlow;
}

LinkedList<Edge> findAugmentingPath(HashMap<Integer,
    Integer>[] g, int s, int t)
{
  // initialize the queue for BFS
  Queue<Integer> Q = new LinkedList<Integer>();
  Q.add(s);
  // initialize the parent array for path
    reconstruction
  Edge[] parent = new Edge[g.length];
  Arrays.fill(parent, null);
  // perform a BFS
  while(!Q.isEmpty())
  {
```

```
    int cur = Q.poll();
    for(Entry<Integer, Integer> e : g[cur].entrySet())
    {
      int next = e.getKey();
      int w = e.getValue();
      if(parent[next] == null)
      {
        Q.add(next);
        parent[next] = new Edge(cur, next, w);
      }
    }
  }
  // reconstruct the path
  if(parent[t] == null) return null;
  LinkedList<Edge> path = new LinkedList<Edge>();
  int cur = t;
  while(cur != s)
  {
    path.add(parent[cur]);
    cur = parent[cur].orig;
  }
  return path;
}

int applyPath(HashMap<Integer, Integer>[] g,
    LinkedList<Edge> path)
{
  int minCapacity = Integer.MAX_VALUE;
  for(Edge e : path)
    minCapacity = Math.min(minCapacity, e.w);
  for(Edge e : path)
  {
    // treat path edge
    if(minCapacity == e.w)
    {
      // the capacity became 0, remove edge
      g[e.orig].remove(e.dest);
    }
    else
    {
      // there remains capacity, update capacity
      g[e.orig].put(e.dest, e.w - minCapacity);
    }
    // treat back edge
    Integer backCapacity = g[e.dest].get(e.orig);
    if(backCapacity == null)
    {
      // the back edge does not exist yet
      g[e.dest].put(e.orig, minCapacity);
    }
    else
    {
      // the back edge already exists, update capacity
      g[e.dest].put(e.orig, backCapacity+minCapacity);
    }
  }
  return minCapacity;
}
```

2.8.4 Coupe minimale

On cherche, avec deux noeuds s et t, V_1 et V_2 tel que $s \in V_1$, $t \in V_2$ et $\sum_{e \in E(V_1,V_2)} w(e)$ minimum.
Il suffit de calculer le flot maximum entre s et t et d'appliquer un parcours du graphe résiduel depuis s(BFS par exemple). Tout les noeuds ainsi parcourus sont dans V_1, les autres dans V_2. Le poids de la coupe est le flot maximum.

3 Autres

3.1 Décomposition en fractions unitaires

Ecrire $0 < \frac{p}{q} < 1$ sous forme de sommes de $\frac{1}{k}$

```
void expandUnitFrac(long p, long q)
{
  if(p != 0)
  {
    long i = q % p == 0 ? q/p : q/p + 1;
    System.out.println("1/" + i);
    expandUnitFrac(p*i-q, q*i);
  }
}
```

3.2 Combinaison

Nombre de combinaison de taille k parmi n (C_n^k)
Cas spécial : $C_n^k \ mod\ 2 = n \oplus m$

```
long C(int n, int k)
{
  double r = 1;
  k = Math.min(k, n - k);
  for(int i = 1; i <= k; i++)
    r /= i;
  for(int i = n; i >= n - k + 1; i--)
    r *= i;
  return Math.round(r);
}
```

3.3 Suite de fibonacci

$f(0) = 0$, $f(1) = 1$ et $f(n) = f(n\breve{}1) + f(n-2)$
Valeur réelle mais avec des flottant : $f(n) = \frac{1}{\sqrt{5}}((\frac{1+\sqrt{5}}{2})^n - (-\frac{2}{1+\sqrt{5}})^n)$
En fait, $f(n)$ est toujours l'entier le plus proche de $f_{approx}(n) = \frac{1}{\sqrt{5}}(\frac{1+\sqrt{5}}{2})^n$

```
long fib(n)
{
  int i=1; int h=1; int j=0; int k=0; int t;
  while(n > 0)
  {
    if(n % 2 == 1)
    {
      t = j * h;
      j=i * h + j * k + t;
      i=i * k + t;
      }
      t = h * h;
      h = 2 * k * h + t;
      k = k * k + t;
  }
  n = (int)n / 2;
  return j;
}
```