

Informatique 09

Graphes

François Bourdoncle

`Francois.Bourdoncle@ensmp.fr`

`http://www.ensmp.fr/~bourdonc/`

Plan

- Composantes connexes d'un graphe non dirigé
- Tri topologique d'un DAG
- Existence d'un chemin dans un graphe dirigé
- Parcours en largeur d'abord
- Plus court chemin
- Test de cyclicité
- Composantes fortement connexes

Composantes connexes d'un graphe non dirigé

```
class Graph {
    List fils[];

    ....

    void composantes(boolean[] vu, int i) {
        if (!vu[i]) {
            vu[i] = true;
            writeln("-> " + i);
            List a = fils[i];
            while (!List.isEmpty(a)) {
                composantes(vu, a.head);
                a = a.tail;
            }
        }
    }

    void composantes() {
        boolean vu = new boolean[fils.length];
        int n = 0;
        for (int i = 0 ; i < fils.length ; ++i) {
            if (!vu[i]) {
                ++n;
                writeln("Composante numéro " + n);
                composantes(vu, i);
            }
        }
    }
}
```

Tri topologique d'un DAG

```
class Graph {
    List fils[];

    List triTopologique(boolean[] vu,
                        int i, List l) {
        if (vu[i]) {
            return l;
        } else {
            vu[i] = true;
            List a = fils[i];
            while (!List.isEmpty(a)) {
                l = triTopologique(vu, a.head, l);
                a = a.tail;
            }
            return List.cons(i, l);
        }
    }

    List triTopologique() {
        boolean vu = new boolean[fils.length];
        List l = List.empty;
        for (int i = 0 ; i < fils.length ; ++i) {
            if (!vu[i]) {
                l = triTopologique(vu, i, l);
            }
        }
        return l;
    }
}
```

Existence d'un chemin

```
class Graph {
    List fils[];
    ...
    boolean chemin(boolean[] vu,
                    int i, int j) {
        if (vu[i]) {
            return false;
        } else if (i == j) {
            writeln(i);
            return true;
        } else {
            vu[i] = true;
            List a = fils[i];
            while (!List.isEmpty(a)) {
                if (chemin(vu, a.head, j)) {
                    writeln(i);
                    return true;
                }
                a = a.tail;
            }
            return false;
        }
    }
}

boolean chemin(int i, int j) {
    boolean vu = new boolean[fils.length];
    return chemin(vu, i, j);
}
}
```

Parcours en largeur d'abord

- Utilisation d'une file FIFO (First In First Out)
- Algorithme itératif

```
class FIFO { ... }
class Graph {
    List[] fils;
    ...
    void parcoursLargeur() {
        boolean vu = new boolean[fils.length];
        FIFO fifo = new FIFO();

        for (int i = 0 ; i < fils.length ; ++i) {
            if (!vu[i]) {
                vu[i] = true;
                fifo.ajouterDernier(i);
                while (!fifo.estVide()) {
                    int j = fifo.enleverPremier();
                    writeln(j);
                    List a = fils[j];
                    while (!List.isEmpty(a)) {
                        if (!vu[a.head]) {
                            vu[a.head] = true;
                            fifo.ajouterDernier(a.head);
                        }
                        a = a.tail;
                    }
                }
            }
        }
    }
}
```

Plus court chemin

```
class Graph {
    List fils[];
    ...
    int distance(int i, int j) {
        // Tableau des distances à "i"
        int dist = new int[fils.length];
        for (int n = 0 ; n < fils.length ; ++n) {
            dist[n] = -1;
        }

        // "dist" donne la priorité des éléments
        FilePriorite file =
            new FilePriorite(fils.length, dist);

        dist[i] = 0; file.ajouterValeur(i));
        do {
            int k = file.enleverMinimum();
            if (k == j) {
                return dist[k];
            } else {
                List a = fils[k];
                while (!List.isEmpty(a)) {
                    if (dist[a.head] < 0) {
                        dist[a.head] = dist[k] + 1;
                        file.ajouterValeur(a.head);
                    }
                    a = a.tail;
                }
            }
        } while (!file.estVide());
        return -1;
    }
}
```

Test de cyclicité

```
class Graph {
    List fils[];
    ...
    boolean cyclique(boolean[] vu,
                    boolean[] actif, int i) {
        if (actif[i]) {
            return true;
        } else if (vu[i]) {
            return false;
        } else {
            List a = fils[i];
            vu[i] = true; actif[i] = true;
            while (!List.isEmpty(a)) {
                if (cyclique(vu, actif, a.head)) {
                    return true;
                }
                a = a.tail;
            }
            actif[i] = false;
            return false;
        }
    }
}

boolean cyclique() {
    boolean vu = new boolean[fils.length];
    boolean actif = new boolean[fils.length];
    for (int i = 0 ; i < fils.length ; ++i) {
        if (cyclique(vu, actif, i)) {
            return true;
        }
    }
    return false;
}
}
```

Composantes fortement connexes (1/2)

```
class Graph {
    List fils[];
    int numero;
    int num[];
    Pile pile;

    void composantes() {
        numero = 0;
        num = new int[fils.length];
        pile = new Pile();
        for (int i = 0 ; i < fils.length ; ++i) {
            num[i] = 0;
        }
        for (int i = 0 ; i < fils.length ; ++i) {
            composantes(i);
        }
    }
}

...
}
```

Composantes fortement connexes (2/2)

```
class Graph {
    List fils[];
    int numero;
    int num[];
    Pile pile;
    ...
    int composantes(int i) {
        if (num[i] != 0) {
            return num[i];
        } else {
            pile.empiler(i);
            num[i] = ++numero;
            int min = num[i];
            List a = fils[i];
            while (!List.isEmpty(a)) {
                min = Math.min(min, composantes(a.head));
                a = a.tail;
            }
            if (min == num[i]) {
                int j;
                write("(");
                do {
                    j = pile.depiler();
                    num[j] = Integer.MAX_VALUE;
                    write(" " + j);
                } while (j != i);
                write(")");
            }
            return min;
        }
    }
}
```