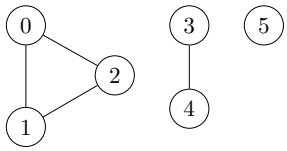
Définitions

- Un **graphe non orienté** est un couple $G = (V, E)$ où V est un ensemble fini de **sommets** et E est un ensemble d'**arêtes**, chaque arête étant un ensemble de deux sommets de V.
  Si $e = \{u, v\} \in E$, on dit que u et v sont **adjacents** ou **voisins** et que ce sont les extrémités de e.
  Le **degré** $\deg(v)$ d'un sommet v est son nombre de voisins.
  Une **feuille** est un sommet de degré 1.
- Un **graphe orienté** est la même chose qu'un graphe non orienté, sauf que chaque arête est un couple (u, v) (représenté par $u \longrightarrow v$) au lieu d'un ensemble.
- Si G est un graphe non orienté à n sommets et p arêtes alors
$$p \leq \binom{n}{2} = O(n^2)$$
En effet, il y a $\binom{n}{2}$ arêtes possibles (il faut choisir les deux extrémités pour avoir une arête). De plus, un graphe avec toutes les arêtes possibles ($p = \binom{n}{2}$) est dit **complet**.
- (HP) Si $G = (V, E)$ est un graphe non orienté alors :
$$\sum_{v \in V} \deg(v) = 2|E|$$
Preuve : Récurrence sur le nombre d'arêtes ou double comptage ($\sum_{v \in V} \deg(v)$ compte deux fois chaque arête, puisqu'une arête a deux extrémités).
- (HP) Un graphe possède un nombre pair de sommets de degrés impairs.
  Preuve :
$$\underbrace{\sum_{\deg(v) \text{ pair}} \deg(v)}_{\text{pair}} + \sum_{\deg(v) \text{ impair}} \deg(v) = \underbrace{2|E|}_{\text{pair}}$$
- Un **chemin** est une suite d'arêtes consécutives différentes.
  La **longueur** d'un chemin est son nombre d'arêtes.
  La **distance** $d(u, v)$ de u à v est la plus petite longueur d'un chemin de u à v (∞ si il n'y a pas de chemin) : c'est une distance au sens mathématique, qui vérifie en particulier l'inégalité triangulaire.
  Dans le cas d'un graphe pondéré (avec des poids sur les arêtes), on somme les poids des arêtes du chemin.

Représentations

- Exemple de graphe avec ses représentations en OCaml :



Matrice d'adjacence

```
[|[|0; 1; 1; 0; 0; 0|];
[|1; 0; 1; 0; 0; 0|];
[|1; 1; 0; 0; 0; 0|];
[|0; 0; 0; 0; 1; 0|];
[|0; 0; 0; 1; 0; 0|];
[|0; 0; 0; 0; 0; 0|]|]
```

Liste d'adjacence

```
[|[1; 2];
[0; 2];
[0; 1];
[4];
[3];
[]|]
```

Une matrice adjacence d'un graphe non orienté est toujours symétrique.
Dans le cas d'un graphe pondéré : on met le poids de l'arête au lieu de 1 pour une matrice d'adjacence et, en général, un couple (voisin, poids) pour une liste d'adjacence.

- Pour un graphe orienté à n sommets et p arêtes :

	Matrice d'adjacence	Liste d'adjacence
complexité mémoire	$O(n^2)$	$\boxed{O(n+p)}$
tester si $\{u, v\} \in E$	$\boxed{O(1)}$	$O(\deg(u))$
parcourir voisins de u	$O(n)$	$\boxed{O(\deg(u))}$

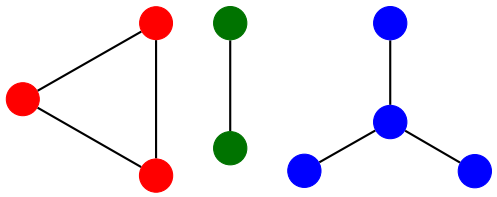
On utilise plutôt une liste d'adjacence pour un graphe avec peu d'arêtes et une matrice d'adjacence sinon.

Connexité, cycle

- Un graphe G non orienté est **connexe** si, pour tout sommets u et v, il existe un chemin entre u et v dans G.
- Un graphe G orienté est **fortement connexe** si, pour tout sommets u et v, il existe un chemin de u à v dans G.
- (HP) Un graphe connexe à n sommets a au moins $n - 1$ arêtes.
  Preuve : Soit $\mathcal{H}(n)$: « un graphe connexe à n sommets a au moins $n - 1$ arêtes ».
  $\mathcal{H}(1)$ est vraie car un graphe à 1 sommet possède 0 arête.
  Supposons $\mathcal{H}(n)$. Soit $G = (V, E)$ un graphe connexe à $n+1$ sommets.
  - Si G a un sommet v de degré 1 alors $G - v$ (G auquel on a enlevé le sommet v) est un graphe *connexe* à n sommets donc, par $\mathcal{H}(n)$, $G - v$ a au moins $n - 1$ arêtes. Donc G a au moins n arêtes.
  - Sinon, tous les sommets de G sont de degré ≥ 2.
    Alors $2|E| = \sum_{v \in V} \deg(v) \geq 2(n + 1) \geq 2n$.
    Donc $|E| \geq n$, ce qui montre $\mathcal{H}(n + 1)$.
- La relation suivante est une relation d'équivalence sur un graphe non orienté :
$$u \sim v \iff \text{il existe un chemin entre } u \text{ et } v$$

Les classes d'équivalences pour $\sim$ sont les sous-graphes connexes maximaux (au sens de $\subseteq$) de G, ils sont appelés **composantes connexes**.

Exemple : un graphe avec 3 composantes connexes.



Remarque : Un graphe est connexe ssi il contient une unique composante connexe.

- (HP) Un graphe acyclique contient un sommet de degré au plus 1.
  Preuve : Supposons que tous les sommets soient de degrés ≥ 2.
  Faisons partir un chemin depuis un sommet quelconque en visitant à chaque étape le sommet adjacent différent du prédécesseur (possible car les degrés sont ≥ 2). Comme le nombre de sommets est fini, ce chemin revient nécessairement sur un sommet déjà visité, ce qui donne un cycle.

- (HP) Un graphe acyclique à n sommets possède au plus $n-1$ arêtes.
  Preuve : Montrons par récurrence $\mathcal{H}(n)$: « un graphe acyclique à n sommets a au plus $n-1$ arêtes ».
  $\mathcal{H}(1)$ est vraie car un graphe à 1 sommet possède 0 arête.
  Supposons $\mathcal{H}(n)$. D'après le lemme, un graphe G acyclique à $n+1$ sommets possède un sommet v de degré ≤ 1.
  Comme G est acyclique, $G-v$ l'est aussi et a au plus $n-1$ arêtes, par $\mathcal{H}(n)$.
  Donc G a au plus $n-1+\deg(v) \leq n$ arêtes, ce qui montre $\mathcal{H}(n+1)$.

- Un graphe T à n sommets et p arêtes est un **arbre** s'il vérifie l'une des conditions équivalentes :
  1. T est connexe acyclique (définition).
  2. T est connexe et $p = n-1$.
  3. T est acyclique et $p = n-1$.

  Remarque : les arbres vus en 1ère année étaient enracinés. Ici il n'y a pas de racine.
  Preuve :
  $1 \implies 2$: Si T est connexe et acyclique alors $p \geq n-1$ (connexe) et $p \leq n-1$ (acyclique) donc $p = n-1$.
  $2 \implies 3$: Supposons T connexe et $p = n-1$.
  Par l'absurde, supposons que T contienne un cycle C. Soit e une arête de C.
  Montrons alors que $T-e$ est connexe.
  Soient donc u et v deux sommets de T. Comme T est connexe, il existe un chemin C' dans T entre u et v.
  Si C' ne contient pas e, C' est un chemin de u à v dans T.
  Si C' contient e, remplaçons e par le reste de C. C' est alors un chemin entre u et v.
  $T-e$ est donc connexe, ce qui contredit le fait que $T-e$ contienne n sommets et $n-2$ arêtes.
  $3 \implies 1$: Supposons T acyclique et $p = n-1$.
  Supposons par l'absurde que T ne soit pas connexe.
  Alors T a $k \geq 2$ composantes connexes T_1, ..., T_k. Notons n_i et p_i le nombre de sommets et d'arêtes de T_i.
  Comme T_i est connexe, $p_i \geq n_i - 1$. Donc $p = \sum p_i \geq \sum (n_i - 1) = n - k$, ce qui est absurde avec l'hypothèse $p = n-1$.

Parcours de graphe

- **Parcours en profondeur** : on visite les sommets le plus profondément possible avant de revenir en arrière.

```
let dfs (g : int list array) r =
    let n = Array.length g in
    let visited = Array.make n false in
    let rec aux v =
        if not visited.(v) then (
            visited.(v) <- true;
            (* traiter v *)
            List.iter aux g.(v)
        ) in
    aux r
```

Complexité avec représentation par liste d'adjacence :
Soit n le nombre de sommets et p le nombre d'arêtes.
`Array.make n false` est en $O(n)$.
Chaque arête est parcourue au plus une fois, d'où $O(p)$ appels récursifs de `aux`.
Au total : $\boxed{O(n+p)}$.
Remarque : La complexité est $O(n^2)$ avec une matrice d'adjacence.

- Application : recherche d'un cycle dans un graphe non orienté, en faisant attention à ne pas considérer une arête comme cycle (pas d'appel récursif sur le père).

```
let has_cycle (g : int list array) =
  let n = Array.length g in
  let visited = Array.make n false in
  let ans = ref false in
  let rec aux p u = (* p a permis de découvrir u *)
    if not visited.(u) then (
      visited.(u) <- true;
      List.filter ((<>) p) g.(u)
      |> List.iter (aux u)
    )
    else ans := true in
  for i = 0 to n - 1 do
    if not visited.(i) then aux i i
  done;
  !ans
```

- **Parcours en largeur** : on visite les sommets par distance croissante depuis le sommet de départ. Pour cela, on stocke les prochains sommets à visiter dans une file `q` (en utilisant ici le module `Queue` d'OCaml) :

```
let bfs (g : int list array) r =
    let seen = Array.make (Array.length g) false in
    let q = Queue.create () in
    let add v =
        if not seen.(v) then (
            seen.(v) <- true; Queue.add v q
        ) in
    add r;
    while not (Queue.is_empty q) do
        let v = Queue.pop q in
        (* traiter v *)
        List.iter add g.(v)
    done
```

- Application : calcul de distance (en nombre d'arêtes), en stockant des couples (sommet, distance) dans `q`.

```
let bfs g r =
    let dist = Array.make (Array.length g) (-1) in
    let q = Queue.create () in
    let add d v =
        if dist.(v) = -1 then (
            dist.(v) <- d;
            Queue.add (v, d) q
        ) in
    add 0 r;
    while not (Queue.is_empty q) do
        let u, d = Queue.pop q in
        List.iter (add (d + 1)) g.(u)
    done;
    dist (* dist.(u) est la distance de r à u *)
```

- Pour obtenir les plus courts chemins, on peut stocker le sommet `pred.(v)` qui a permis de découvrir `v` :

```
let bfs g r =
    let pred = Array.make (Array.length g) (-1) in
    let q = Queue.create () in
    let add p v = (* p est le père de v *)
        if pred.(v) = -1 then (pred.(v) <- p; Queue.add v q) i
    add r r;
    while not (Queue.is_empty q) do
        let v = Queue.pop q in
        List.iter (add v) g.(v)
    done;
    pred (* pred.(v) = père de v dans le parcours *)
```

On peut alors obtenir un plus court chemin de `r` à `v` en remontant les pères de `v` jusqu'à obtenir `r` :

```
let rec path pred v =
    if pred.(v) = v then [v]
    else v::path pred pred.(v)
```