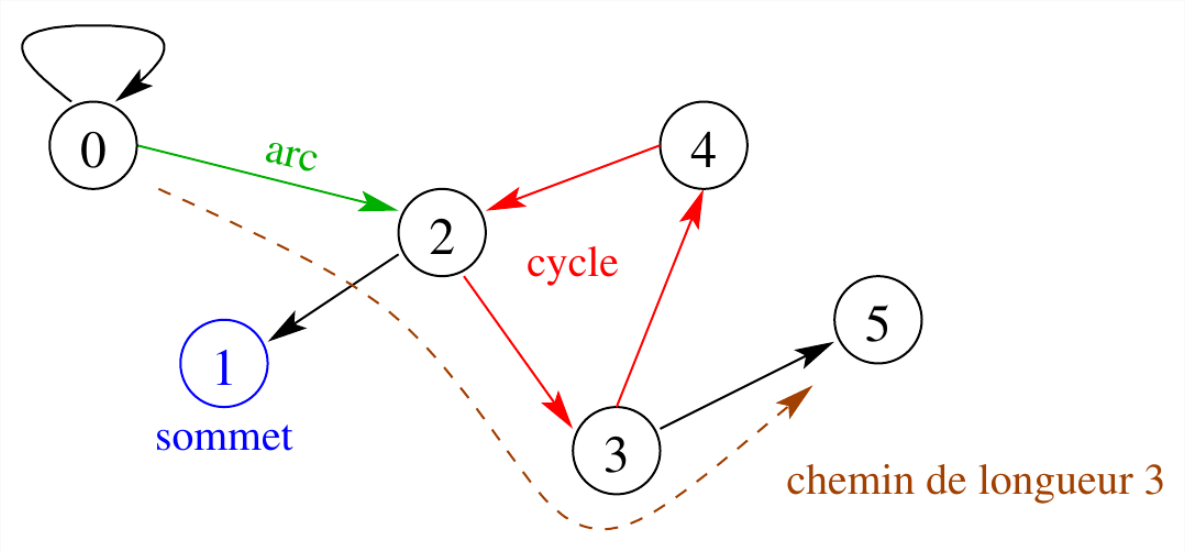
Graphes en C

MP2I - Informatique

Anthony Lick
Lycée Janson de Sailly

Rappels

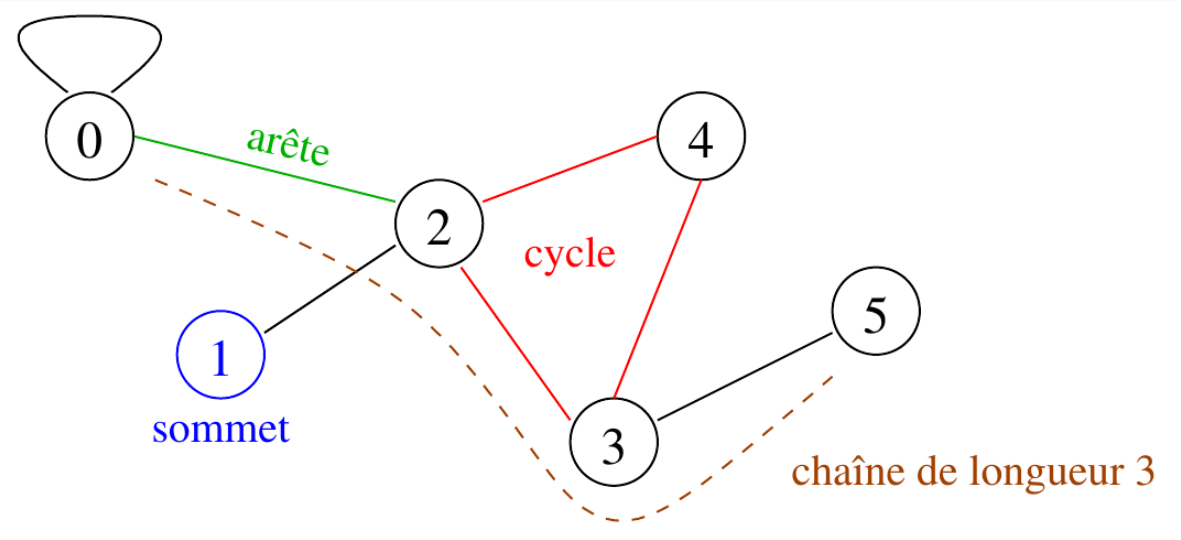
Rappels



Rappel : graphe orienté : $G = (S, A)$

- S : ensemble de **sommets**.
- A : ensemble d'**arcs** ($A \subset S \times S$).
- **Chemin** : suite d'arcs consécutifs.
- **Longueur** d'un chemin : nombre d'arcs sur le chemin.
- **Circuit** : chemin qui boucle (on parle parfois de **cycle**).

Rappels

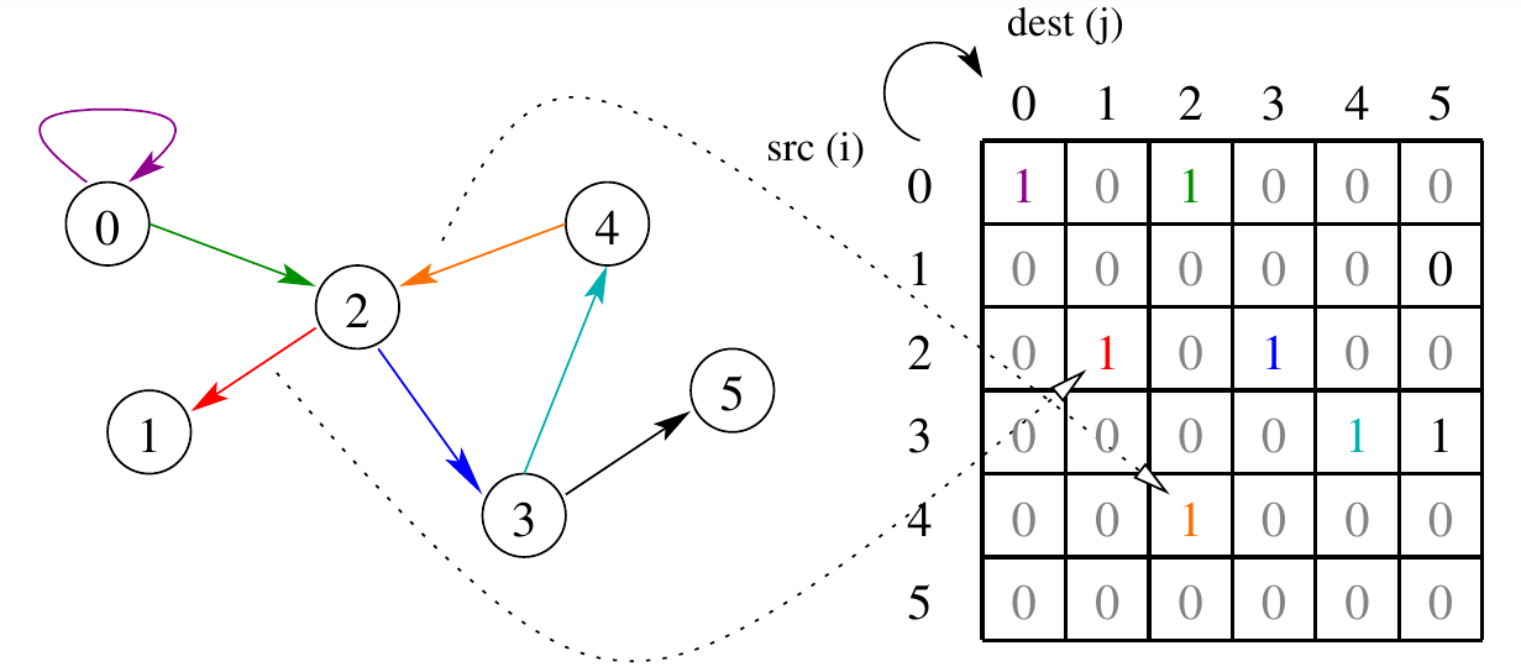


Rappel : graphe non orienté : $G = (S, A)$

- On parle d'**arêtes** et non d'**arcs**.
- On parle parfois de **chaîne** plutôt que de **chemin**.
- On parle de **cycle** et non de **circuit**.

Représentation par matrice d'adjacence

Matrices d'adjacence



Rappel :matrice d'adjacence

- $S = [\![0, n-1]\!]$;
- matrice M de taille $n \times n$ telle que :
  - $M_{i,j} = 1$ si $(i, j) \in A$;
  - $M_{i,j} = 0$ sinon.

Matrices d'adjacence en C

Tableaux statiques vs dynamiques

- Tableau **statique** à deux dimensions :

  ↪ `bool graph[NB_VERTICES][NB_VERTICES];`

- Tableau **dynamique** à deux dimensions :

  ↪ `bool **graph;`

```
1  bool **graph = (bool**)malloc(NB_VERTICES * sizeof(bool*));
2  for (int i=0; i < NB_VERTICES; i++) {
3    graph[i] = (bool*)malloc(NB_VERTICES * sizeof(bool)); // il faudra tous les libérer avec free
4  }
```

Matrices d'adjacence en C

Linéarisation des accès

Linéarisation des accès :

`graph[i][j]` $\rightsquigarrow$ `graph[i * NB_VERTICES + j]`.

- Tableau **statique** à une dimension :

  ↪ `bool graph[NB_VERTICES * NB_VERTICES];`

- Tableau **dynamique** à une dimension :

  ↪ `bool *graph;`

```
1  bool *graph = (bool*)malloc(NB_VERTICES * NB_VERTICES * sizeof(bool)); // un seul free nécessaire
```

Matrices d'adjacence en C

Graphes pondérés

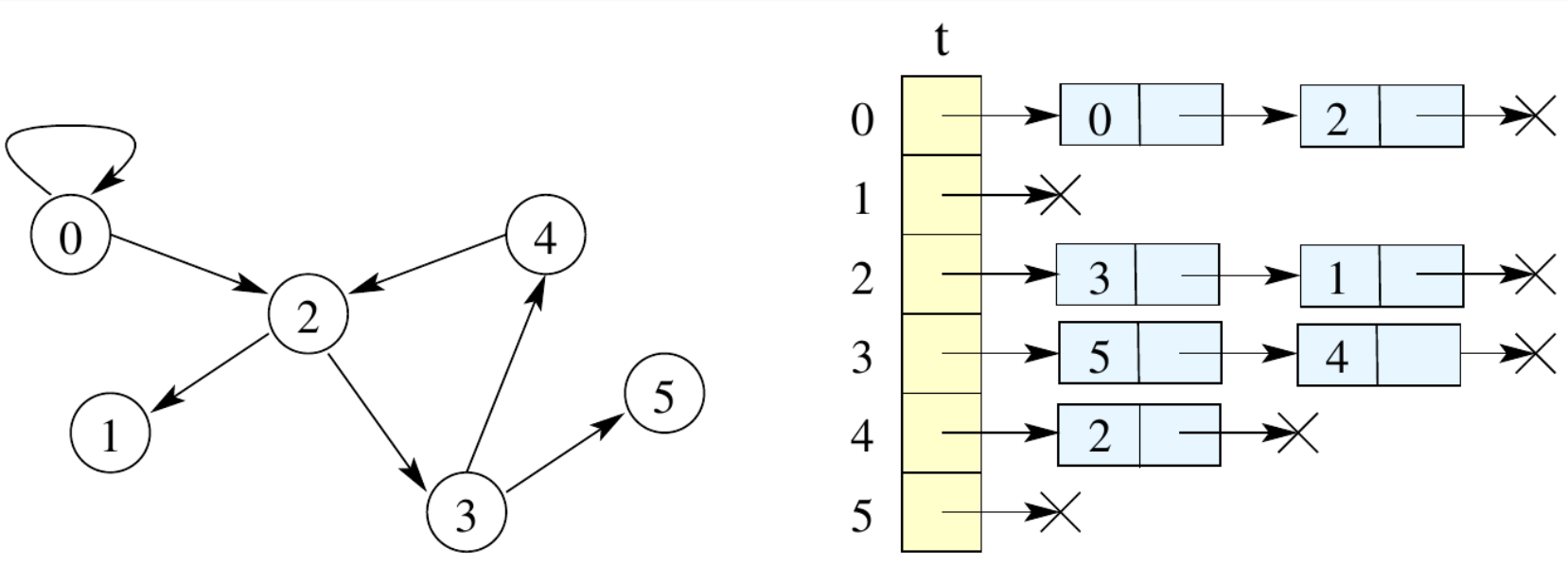
Pour les **graphes pondérés**, on utilisent des tableaux d'`int`, et on choisit une valeur spéciale pour représenter $+\infty$ dans la matrice.

Attention

Ne pas utiliser `NULL` qui vaut 0, mais plutôt une valeur dont on est sûr qu'elle ne sera pas utilisée en pratique.

Représentations par listes d'adjacence

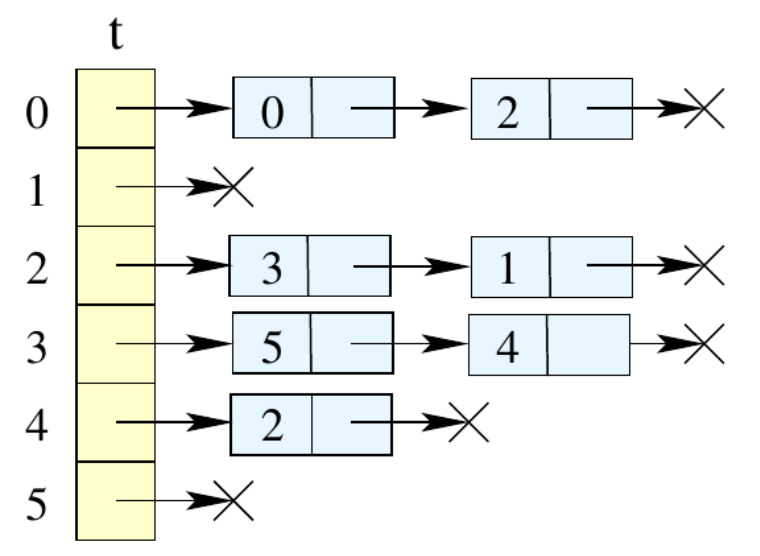
Listes d'adjacence



Listes d'adjacence

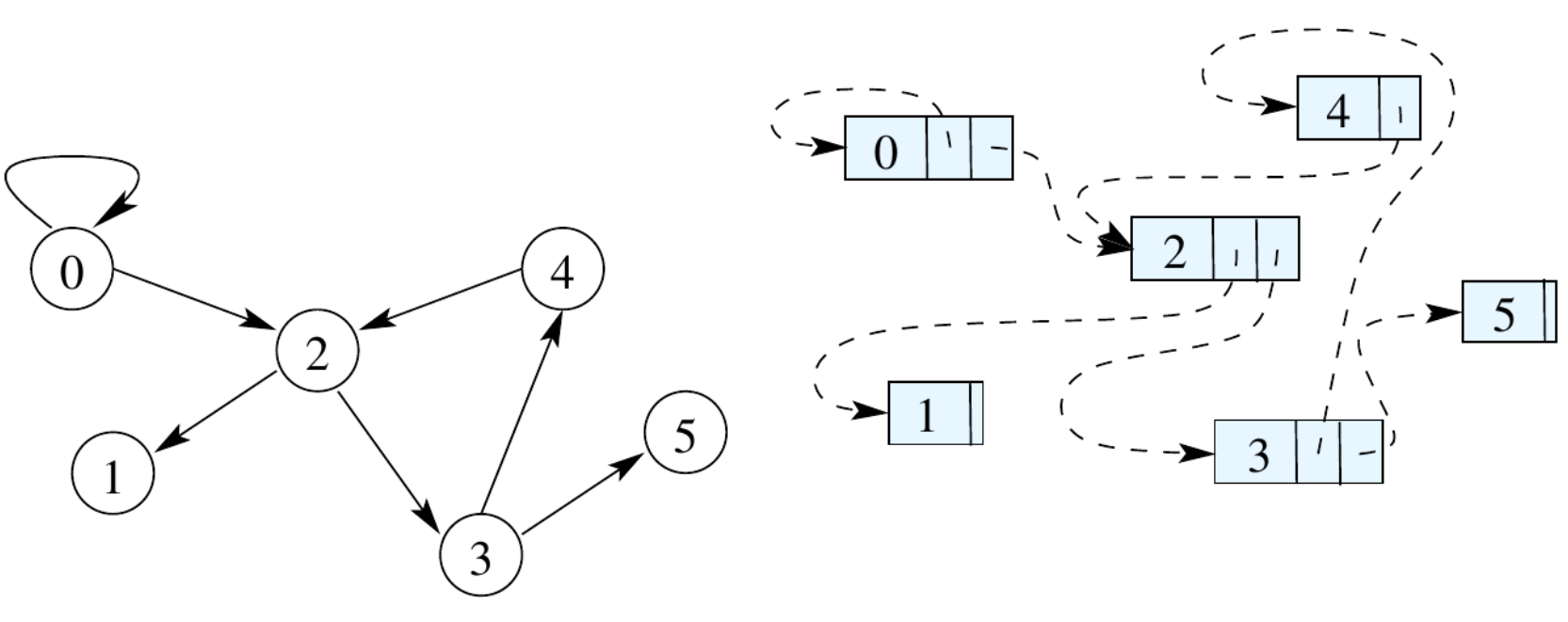
- Tableau `t` des **successeurs** pour chaque sommet ;
- `t[i]` pointe vers la **liste chaînée** des successeurs du sommet i ;
- accès direct à un sommet via `t` ;
- complexité spatiale optimale : $\mathcal{O}(|S| + |A|)$.

Listes d'adjacence




```
1   typedef int edge_val ;
2   typedef int vertex_name_t ;
3
4   struct edge_list_t {
5     int dest ;
6     edge_val data ; // graphes pondérés
7     struct edge_list_t *next ;
8   };
9   struct vertex_t {
10    vertex_name_t name ;
11    struct edge_list_t *edges ;
12  };
13  struct graph_t {
14    int nb_vertices ;
15    struct vertex_t *vertices [MAX_VERTICES] ;
16  };
```

Listes d'adjacence

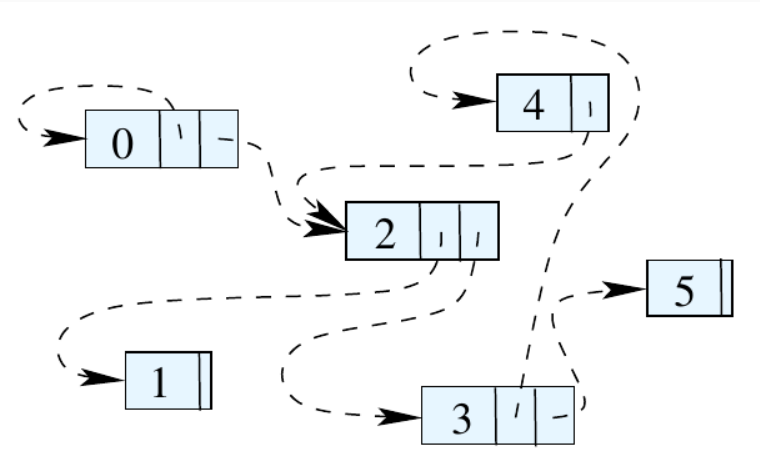


Variante

On peut aussi **partager physiquement** les sommets :

- chaque sommet est représenté **une et une seule fois** ;
- chaque sommet possède une **liste chaînée** dont les éléments pointent vers ses **successeurs**.

Listes d'adjacence




```
1   struct vertex_t {
2     vertex_name_t name ;
3     struct vertex_list_t *neighbours ;
4     bool seen ;
5   };
6   struct vertex_list_t {
7     struct vertex_t *vertex ;
8     struct vertex_list_t *next ;
9   };
10  struct graph_t {
11    int nb_vertices ;
12    struct vertex_list_t *vertices ;
13  };
```

Remarque

On peut aussi intégrer un **marqueur** de type **bool** à chaque sommet, pour les marquer comme **déjà visités**. Dans ce cas, il faut penser à les **réinitialiser** entre chaque parcours.

Listes d'adjacence

Tableaux statiques

Pour simplifier les implémentations en C, on peut utiliser des **tableaux statiques** :

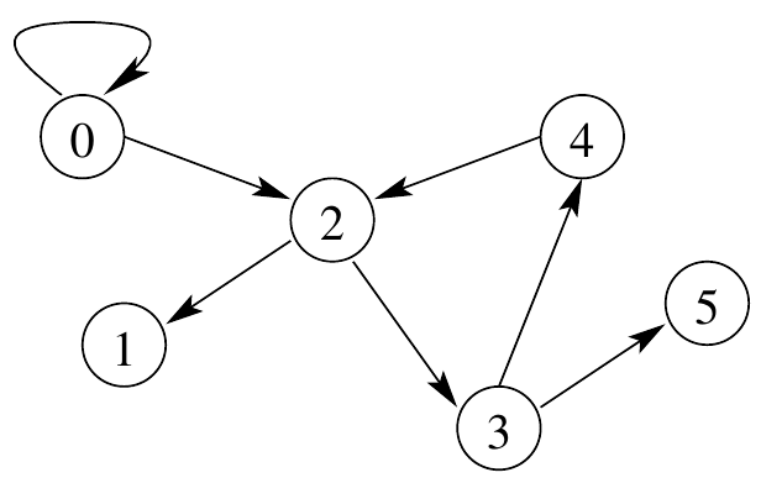
- chaque liste chaînée est remplacée par un tableau de taille $n+1$, dont les premières cases contiennent les successeurs du sommet ;
- pour savoir combien de cases de ce tableau "comptent vraiment", deux solutions :
  - utilisation d'une **sentinelle** ;
  - stocker le nombre de voisins dans la première case du tableau.

Listes d'adjacence

Tableaux statiques

Pour créer un graphe, on crée donc un tableau de taille $n \times (n+1)$, de la même manière que pour les matrices d'adjacence.

Listes d'adjacence



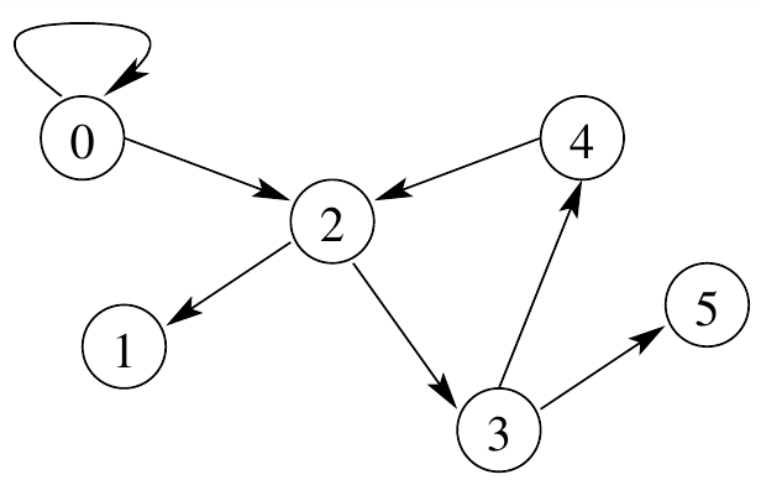
$$\begin{pmatrix} 0 & 2 & -1 & \cdot & \cdot & \cdot \\ -1 & \cdot & \cdot & \cdot & \cdot & \cdot \\ 3 & 1 & -1 & \cdot & \cdot & \cdot \\ 5 & 4 & -1 & \cdot & \cdot & \cdot \\ 2 & -1 & \cdot & \cdot & \cdot & \cdot \\ -1 & \cdot & \cdot & \cdot & \cdot & \cdot \end{pmatrix}$$

Sentinelle

Si on choisit l'implémentation avec **sentinelle**, on choisit une valeur spéciale (e.g. -1) qui marque la fin de la liste des sommets dans le tableau (les cases situées après ne comptent pas).

Il faut donc parcourir chaque liste **tant qu**'on ne tombe pas sur la **sentinelle**.

Listes d'adjacence



$$\begin{pmatrix} 2 & 0 & 2 & \cdot & \cdot & \cdot \\ 0 & \cdot & \cdot & \cdot & \cdot & \cdot \\ 2 & 3 & 1 & \cdot & \cdot & \cdot \\ 2 & 5 & 4 & \cdot & \cdot & \cdot \\ 1 & 2 & \cdot & \cdot & \cdot & \cdot \\ 0 & \cdot & \cdot & \cdot & \cdot & \cdot \end{pmatrix}$$

Indicateur de taille en premier indice

Si on choisit l'implémentation avec **indicateur de taille en premier indice**, chaque ligne `t[i]` contient une valeur $n_i =$ `t[i][0]` dans sa première case, et les successeurs de i sont stockés dans les cases `t[i][1]` à `t[i]`$[n_i]$.

Parcours en largeur

Parcours en largeur

Rappels

Parcours en **largeur** :

- utilisation d'une **file** ;
- traitement des sommets par **distance croissante** du sommet source.

Parcours en largeur

```c
struct vertex_t {
  vertex_name_t name ;
  struct vertex_list_t *neighbours ;
  bool seen ;
};
struct vertex_list_t {
  struct vertex_t *vertex ;
  struct vertex_list_t *next ;
};
struct graph_t {
  int nb_vertices ;
  struct vertex_list_t * vertices ;
};
typedef struct vertex_t *value_t ;
struct queue_t {
  unsigned int max_nb ;
  unsigned int cur_nb ;
  unsigned int first ;
  value_t *storage ;
};
struct queue_t *empty_queue () ;
bool pop(struct queue_t *q, value_t *res) ;
bool push(struct queue_t *q, value_t val) ;
bool is_empty(struct queue_t *q) ;
void free_queue(struct queue_t *q);
```

```c
void bfs(struct vertex_t *s){
  struct queue_t *q = empty_queue();
  s->seen = true;
  push(q,s);
  while (!is_empty(q)){
    struct vertex_t *n;
    pop(q, &n);
    printf("%d\n", n->name);
    struct vertex_list_t *neighb =
      n->neighbours;
    while (neighb != NULL){
      if (neighb->vertex != NULL
          && !neighb->vertex->seen){
        neighb->vertex->seen = true;
        push(q, neighb->vertex);
      }
      neighb = neighb->next;
    }
  }
  free_queue(q);
}
```

Parcours en profondeur

Parcours en profondeur

Rappels

Parcours en **profondeur** :

- on avance le plus **profondément** possible le long d'un chemin ;
- algorithme naturellement **récursif**.

Parcours en profondeur

```c
struct edge_list_t{
  int dest;
  edge_val data;
  struct edge_list_t *next;
};

struct vertex_t{
  vertex_name_t name;
  struct edge_list_t *edges;
};

struct graph_t{
  int nb_vertices;
  struct vertex_t *vertices[MAX_VERTICES];
};
```

```c
void pp(struct graph_t *g, int s, bool *seen){
  seen[s] = true;
  printf("%d",g->vertices[s]->name);
  struct edge_list_t *e =
    g->vertices[s]->edges;
  while (e != NULL) {
    if !seen[e->dest]{
      printf(" -%d->", e->data);
      pp(g, e->dest, seen);
    }
    e = e->next;
  }
}

void dfs(struct graph_t *g, int s){
  if (g->nb_vertices <= 0) return;
  bool *seen =
    malloc(sizeof(bool) * g->nb_vertices);
  if (seen == NULL) return;
  pp(g, s, seen);
  free(seen);
}
```

Parcours en profondeur

```c
struct vertex_t{
  vertex_name_t name;
  struct vertex_list_t *neighbours;
  bool seen;
};

struct vertex_list_t{
  struct vertex_t *vertex;
  struct vertex_list_t *next;
};

struct graph_t{
  int nb_vertices;
  struct vertex_list_t *vertices;
};
```

```c
void dfs(struct vertex_t *v){
  v->seen = true;
  printf(" %d", v->name);
  struct vertex_list_t *neigh = v->neighbours;
  while (neigh != NULL){
    if (!neigh->vertex->seen){
      printf(" ->");
      dfs(neigh->vertex);
    }
    neigh = neigh->next;
  }
}
```