2.2 Parcours en largeur

Définition 15.4 — Distance dans un graphe non pondéré

Soit G un graphe, éventuellement orienté mais non pondéré. La distance $d(x, y)$ d'un sommet x à un sommet y est la longueur minimale, en nombre d'arêtes, d'un chemin reliant x à y.
Si un tel chemin n'existe pas, $d(x, y) = \infty$.

Remarques

- Comme vu au chapitre précédent, s'il existe un chemin de x à y, alors il existe un chemin élémentaire de x à y, et un plus court chemin est nécessairement élémentaire. Par conséquent, la définition précise choisie pour la notion de chemin n'influe pas sur la définition de la distance.
- Si G est connexe et non orienté, alors d est bien une distance au sens mathématique usuel.
- Si G est non orienté et non connexe, d est essentiellement une distance, mais prend ses valeurs dans $\mathbb{R}_+ \cup \{\infty\}$.
- Si G est orienté, d n'est plus symétrique : ce n'est donc pas une distance. En revanche, l'inégalité triangulaire est toujours vérifiée (et on a bien sûr $d(x, y) \geqslant 0$ avec égalité ssi $x = y$).

Propriété 15.5

S'il existe un arc yy', alors pour tout sommet x on a $d(x, y') \leqslant d(x, y) + 1$.

Le *parcours en largeur* d'un graphe à partir d'un sommet v permet de visiter les sommets par distance croissante à v :

Algorithme 7 Parcours en largeur (*Breadth-First Search*) à l'aide d'une file.

```
fonction BFS(G, x0)
    ouverts ← file_vide()
    PUSH(x0, ouverts)
    vus ← {x0}
    tant que ouverts n'est pas vide faire
        x ← POP(ouverts)                          ▷ On extrait l'élément de tête
        TRAITEMENT(x)
        pour y ∈ G.successeurs(x) faire
            si y ∉ vus alors
                PUSH(y, ouverts)                  ▷ On ajoute y en queue.
                vus ← vus ∪{y}
```

Remarques

- Le fait que *ouverts* soit une *file* (structure FIFO) est crucial !
- On pourrait bien sûr définir une fonction BFS-COMPLET comme plus haut, mais elle serait assez peu utile : le parcours en largeur n'est en règle général intéressant qu'à partir d'un certain nœud distingué.
- L'appel à TRAITEMENT pourrait être fait au moment où l'on pousse le nœud sur la file sans impacter la propriété fondamentale (les nœuds sont traités par distance croissante à x_0).

Théorème 15.6 — Propriété fondamentale du parcours en largeur

Un appel à BFS(G, x_0), où x_0 est un sommet du graphe fini G, termine après avoir visité tous les sommets accessibles depuis x_0 une et une seule fois. Les visites de ces sommets se font par distance croissante à x_0.
Ainsi, si $d(x_0, x) < d(x_0, y) < \infty$, alors TRAITEMENT$(x)$ sera exécuté avant TRAITEMENT(y).

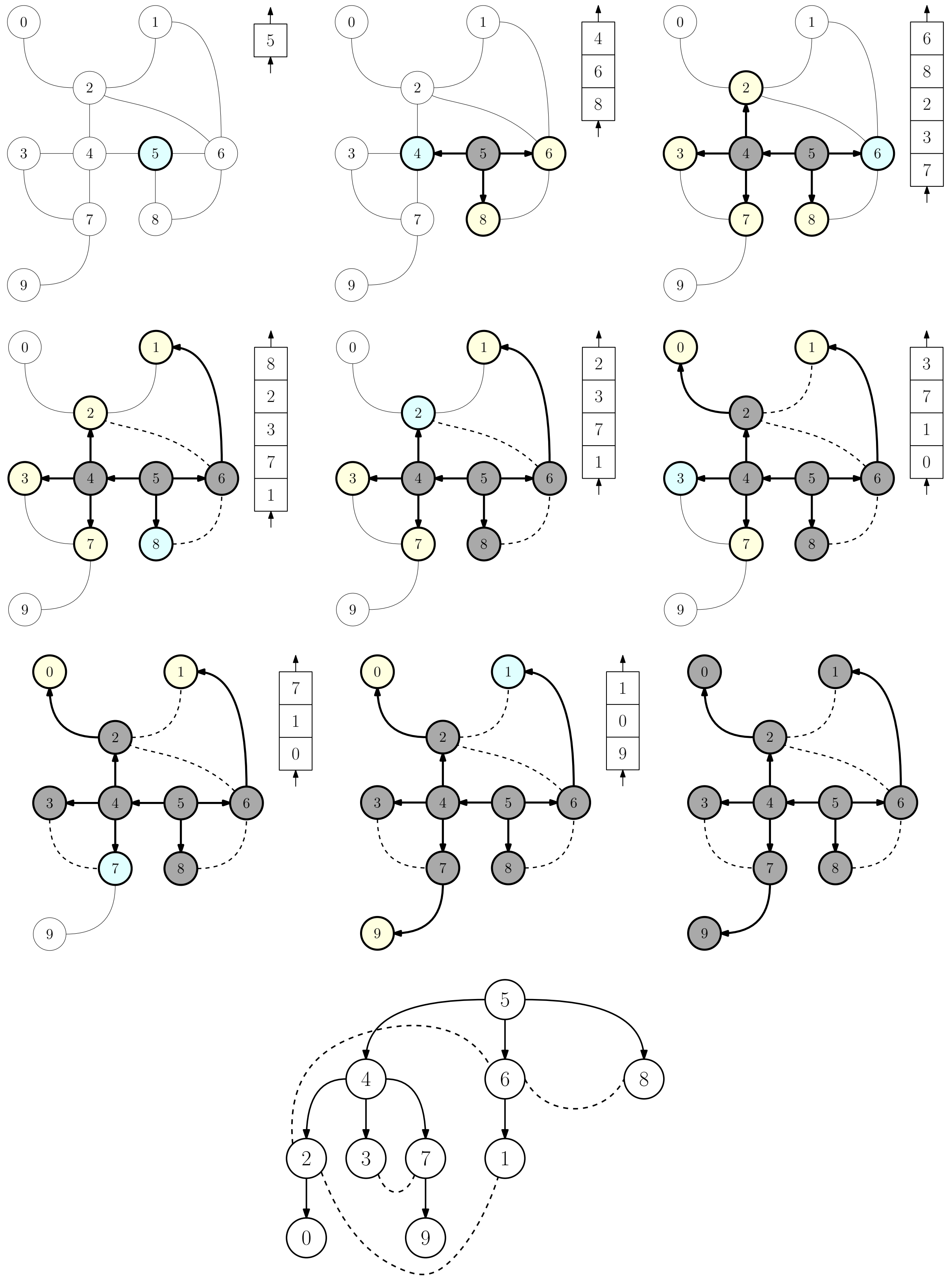

FIGURE 15.4 – Parcours en largeur d'un graphe non orienté.

Démonstration
Quelques observations pour commencer :

- un sommet est ajouté au plus une fois à la file (G étant fini, cela garantit la terminaison), et tout sommet ajouté finit par être traité ;
- on adapte facilement la preuve faite pour le parcours en profondeur pour montrer que les sommets visités sont exactement ceux accessibles depuis x_0 ;
- l'ordre de traitement des sommets est le même que l'ordre dans lequel ils sont ajoutés à la file.

À un instant donné, un sommet sera dit :

- *ouvert* s'il appartient à *ouverts*[3] ;
- *fermé* s'il appartient à `vus` mais pas à *ouverts* ;
- *vierge* sinon (il n'y a que trois cas car *ouverts* $\subset$ `vus`).

On numérote les sommets dans l'ordre de leur ouverture, $x_0, \dots, x_n$, et l'on note $d_k := d(x_0, x_k)$. L'invariant, valable à la fermeture de x_k, est le suivant :

(1) la file a la forme $x_{k+1}, \dots, x_{k+p}$ avec $d_k \leqslant d_{k+1} \leqslant \dots \leqslant d_{k+p} \leqslant d_k + 1$ (un nombre éventuellement nul de sommets à distance d_k suivis d'un nombre éventuellement nul de sommets à distance $d_k + 1$) ;

(2) si $d_i < d_k$, alors x_i est fermé (*i.e* $i < k$) ;

(3) si $d_i = d_k$, alors x_i n'est pas vierge.

On passe à la preuve :

Initialisation Quand on ferme x_0, la file est vide ; de plus, aucun i ne vérifie $d_i < d_0$ et seul $i = 0$ vérifie $d_i = d_0$, donc l'invariant est vérifié.

Invariance On suppose l'invariant vérifié à l'étape $k < n$, on ferme x_k et l'on ajoute ses successeurs vierges $y_1, \dots, y_l$ sur la file. Comme les y_i sont vierges, on a d'après l'invariant $d(x_0, y_i) > d_k$. Comme de plus $d(x_0, y_i) \leqslant d_k + 1$ d'après la proposition 15.5, on a en fait $d(x_0, y_i) = d_k + 1$: la forme de la file est préservée.
On distingue maintenant deux cas :

- si $d_{k+1} = d_k$, il n'y a rien de plus à prouver ;
- si $d_{k+1} = d_k + 1$, alors tous les sommets à distance d_k sont fermés, puisqu'il ne peut en rester sur la file et que l'invariant garantit qu'aucun n'est vierge. Cela montre le point (2) de l'invariant. De plus, comme tous ces sommets sont fermés, aucun de leurs successeurs ne peut être vierge ; comme tout sommet à distance $d_k + 1$ est successeur d'un sommet à distance d_k, cela montre le point (3) de l'invariant.

Conclusion L'invariant est donc vérifié pour tout k, et son point (2) garantit donc que si $d_i < d_k$, alors $i < k$: c'est la conclusion cherchée.

■

Exercice 15.8 p. 322

1. Écrire une fonction `bfs` qui effectue un parcours en largeur d'un graphe à partir d'un sommet x_0 fourni en argument. Cette fonction affichera les sommets du graphe par distance croissante à x_0. On supposera que le graphe est donné sous forme d'un tableau de listes d'adjacence, et l'on pourra utiliser le module `Queue` pour réaliser la file.
   - `Queue.create : unit -> 'a Queue.t` crée une file vide.
   - `Queue.is_empty : 'a Queue.t` teste si une file est vide.
   - `Queue.push : 'a -> 'a Queue.t -> unit` ajoute un élément à la file.
   - `Queue.pop : 'a Queue.t -> 'a` extrait un élément de la file (qui doit être non vide).

   ```
   bfs : int list array -> int -> unit
   ```

2. Simuler l'exécution de cet algorithme sur le graphe de la figure 15.2 à partir du sommet 0, et dessiner l'arbre de parcours correspondant.

3. Wow !

3. Si l'on ne souhaite pas utiliser le module **Queue**, comment peut-on réaliser de manière efficace une file impérative ? fonctionnelle ? On ne demande pas d'implémenter ces structures mais simplement de se remémorer les techniques que nous avons vues.

Théorème 15.7 — Complexité du parcours en largeur

Le parcours en larguer a une complexité spatiale en $\Theta(|V|)$. En supposant que les opérations élémentaires sur les files et les ensembles se font en temps constant, sa complexité temporelle est en $\mathcal{O}(|E| + |V|)$.

Démonstration
Si l'on réalise `vus` par un tableau de $|V|$ booléens, l'initialisation se fait en temps $\Theta(|V|)$. Ensuite, le traitement de chaque nœud prend un temps proportionnel à son nombre de successeurs (plus une constante). Ainsi, le temps total est proportionnel à la taille (nombre d'arêtes plus nombre de sommets) du sous-graphe accessible depuis le sommet initial. Cette taille est bien en $\mathcal{O}(|E|+|V|)$, donc la complexité temporelle totale est en $\mathcal{O}(|E| + |V|)$.
Pour l'espace, on consomme $\Theta(|V|)$ pour `vus` et $\mathcal{O}(|V|)$ sur la file (puisque tous les sommets présents sur la file sont distincts). Au total, la complexité spatiale est donc en $\mathcal{O}(|V|)$. ■

Exercice 15.9 — Parcours en largeurs avec générations explicites p. 323

On considère l'algorithme suivant :

Algorithme 8 Parcours en largeur avec générations explicites.

fonction BFS(G, x_0)
  $\mathrm{vus} \leftarrow \{x_0\}$ ▷ Ensemble
  $\mathrm{actuels} \leftarrow [x_0]$ ▷ Liste
  tant que $\mathrm{actuels} \neq \emptyset$ **faire**
    $\mathrm{nouveaux} \leftarrow []$
    pour $x \in \mathrm{actuel}$ **faire**
      Traiter x.
      pour $y \in G.\mathrm{successeurs}(x)$ **faire**
        si $y \notin \mathrm{vus}$ **alors**
          Ajouter y à $\mathrm{nouveaux}$.
          Ajouter y à vus.
    $\mathrm{actuels} \leftarrow \mathrm{nouveaux}$

1. Justifier que cet algorithme termine.
2. Énoncer un invariant permettant de montrer que cet algorithme effectue bien un parcours en largeur (qu'il traite exactement les sommets accessibles depuis x_0, par distance croissante à x_0).
3. Rédiger la preuve de correction de l'algorithme.
4. Écrire en OCaml une fonction `tableau_distances` prenant en entrée un graphe sous forme d'un tableau de listes d'adjacences et un sommet x_0, et utilisant l'algorithme ci-dessus pour calculer un tableau `dist` tel que `dist.(i)` soit la distance entre x_0 et le sommet i. On mettra une valeur de -1 dans le tableau pour les sommets qui ne sont pas accessibles depuis x_0.

```
tableau_distances : int list array -> int -> int array
```