Matthieu Boyer
Planche A

Graphe pondéré et algorithme de Dijkstra en SQL

On souhaite représenter un **graphe orienté pondéré** dans une base de données relationnelle, puis implémenter l'algorithme de Dijkstra à l'aide de requêtes SQL. On dispose du schéma relationnel suivant :

```
Sommet(id INTEGER, nom TEXT)
Arc(source INTEGER, destination INTEGER, poids INTEGER)
```

avec les contraintes :

- `Arc.source` et `Arc.destination` sont des clés étrangères référençant `Sommet.id` ;
- `poids` est strictement positif.

1. Écrire la requête SQL permettant de créer les deux tables avec leurs contraintes d'intégrité (clé primaire, clé étrangère, vérification du poids).
2. Écrire une requête retournant, pour un sommet d'identifiant `:s` donné, la liste de ses **voisins directs** (nom et poids de l'arc).
3. Écrire une requête retournant les sommets n'ayant **aucun arc entrant** (racines potentielles du graphe).
4. Écrire une requête retournant, pour chaque sommet, le **degré sortant** (nombre d'arcs issus de ce sommet), en incluant les sommets de degré 0.

On ajoute deux tables de travail (tables temporaires ou tables classiques vidées en début d'exécution) :

```
Distance(sommet INTEGER, dist INTEGER, predecesseur INTEGER)
Visite(sommet INTEGER)
```

`dist` contient la distance provisoire depuis la source `:src` (`NULL` signifie $+\infty$), et `predecesseur` permet de reconstituer le chemin.

5. Écrire les requêtes d'initialisation : toutes les distances à `NULL`, sauf celle de la source fixée à 0, et la table `Visite` vide.
6. À chaque itération de Dijkstra, on extrait le sommet non visité de distance minimale. Écrire la requête SELECT permettant de l'identifier. *(On suppose que les distances nulles/infinies sont gérées via `IS NULL` et `COALESCE`.)*
7. Soit `:u` le sommet extrait à l'étape précédente. Écrire la requête de **relaxation** des arcs : mettre à jour `Distance` pour tous les voisins `v` de `:u` tels que $d[\texttt{u}] + \texttt{poids(u,v)} < d[\texttt{v}]$ (ou $d[\texttt{v}]$ infini).
8. Écrire la requête marquant `:u` comme visité.
9. Après convergence, écrire la requête permettant de retrouver le chemin le plus court de la source `:src` à une cible `:dst` en remontant les prédécesseurs. *(On utilisera une CTE récursive.)*
10. Pourquoi l'algorithme de Dijkstra *tel qu'implémenté ici* ne fonctionne-t-il pas si certains poids sont négatifs ? Quelle modification de schéma ou de requête permettrait de détecter ce cas et de lever une erreur ?

Matthieu Boyer
Planche B

Implémentation d'un mini-langage de requêtes en OCaml

On souhaite implémenter en OCaml un petit moteur de requêtes relationnel, inspiré de SQL, opérant sur des tables représentées en mémoire.

On fixe les types de base suivants, déjà donnés :

```
type valeur = Int of int | Str of string
type schema = string list (* liste ordonnée des noms de colo
type ligne = valeur list (* une ligne : autant de valeurs que de colonnes *)
type table = { schema : sch
```

On dispose aussi de la fonction auxiliaire :

```
(* index_of col sch renvoie l'indice de col dans sch, lève Not_found
let index_of (col : string) (sch : schema) : int =
let rec aux i = function
| [] -> raise Not_found
| c :: _ when c = col -> i
| _ :: rest -> aux (i + 1) rest
in aux 0 sch
```

1. Écrire la fonction `val get : string -> schema -> ligne -> valeur` qui retourne la valeur de la colonne `col` dans une ligne donnée, en utilisant `index_of`.

2. On représente les conditions de filtrage par le type :

```
type cond =
| Egal of string * valeur (* col = val *)
| Inf of string * valeur (* col < val *)
| Et of cond * cond
| Ou of cond * cond
| Non of cond
```

   Écrire la fonction `val eval_cond : schema -> ligne -> cond -> bool` qui évalue une condition sur une ligne. On supposera que les comparaisons `Inf` sont uniquement entre entiers ; lever `Invalid_argument` sinon.

3. Écrire la fonction `val where : cond -> table -> table` qui filtre les lignes d'une table selon une condition.

4. Écrire la fonction `val select_cols : string list -> table -> table` qui projette une table sur une liste de colonnes (dans l'ordre donné), en éliminant les autres.

5. Écrire la fonction `val product : table -> table -> table` qui calcule le produit cartésien de deux tables. Le schéma résultant est la *concaténation* des deux schémas. *On ne traitera pas ici les conflits de noms de colonnes.*

6. En déduire la fonction `val join : cond -> table -> table -> table` qui réalise la jointure interne (*inner join*) des deux tables selon une condition, par la méthode des boucles imbriquées (*nested loop join*).

On représente les fonctions d'agrégation disponibles par : `type agregat = Count | Sum of string | Max of string`

7. Écrire la fonction `val eval_agregat : agregat -> schema -> ligne list -> valeur` qui calcule un agrégat sur une liste de lignes.
   - `Count` retourne `Int (List.length lignes)` ;
   - `Sum col` retourne la somme des entiers de la colonne `col` (lever `Invalid_argument` si une valeur n'est pas un `Int`) ;
   - `Max col` retourne le maximum des valeurs entières de la colonne `col` (lever `Not_found` si la liste est vide).

8. Écrire la fonction `val group_by : string -> agregat -> string -> table -> table` d'appel `group_by col_groupe agr col_res t` qui :
   (a) regroupe les lignes de `t` par valeur distincte de `col_groupe` ;
   (b) calcule `agr` sur chaque groupe ;
   (c) retourne une table à deux colonnes `[col_groupe ; col_res]`.

   On pourra utiliser une table de hachage ou trier les lignes au préalable.

9. On souhaite ajouter un filtrage *après* agrégation (l'équivalent du `HAVING` SQL). Proposer une signature OCaml pour cette fonctionnalité et expliquer comment l'implémenter en réutilisant les fonctions précédentes.