Crino' G., 794281, "Facebook", 21.01.2016

Cosa richiede il progetto

Il progetto richiede di implementare una rete di persone che instaurano fra di loro una relazione di amicizia in un anno specifico.

Bisogna quindi scrivere un programma che accetti da `stdin` comandi per inserire una nuova persona al *database*, cercarla, aggiungere una relazione di amicizia con un'altra persona, estrarre le persone che direttamente o *indirettamente* la conoscono.

Il programma, implementato in `C`, accetta i comandi seguenti:

- `add <first_name> <last_name> <id>`: aggiunge `last_name, first_name` al *database* assegnandogli l'`id` passato da linea di comando.
- `find <id>`: cerca nel database l'utente con id `id`.
- `mfriends <id0> <id1>`: stringe un'amicizia fra `id0` e `id1`
- `extract_groups [since <year_of_start>]`, per estrarre i gruppi di amici, con la possibilita' di selezionare solo le relazioni nate *dopo* un anno specificato da `year_of_start`.

Per esempio, un input valido e'

```
# Add some users
add Linus Torvalds 2
add David Bowie 0
add Kevin Spacey 4
add Steve Jobs 1
add Vincent Cassel 6
add David Gandy 10
add Sergio Rubini 11
add Al Pacino 3
add Forrest Gump 5
add Laura Morante 12

# Make some friendships!
mfriends 0 2 1988
mfriends 0 5 2015
mfriends 2 1 2024
mfriends 2 5 1987
mfriends 2 3 2027
mfriends 3 6 2005
mfriends 5 4 2008
mfriends 10 12 1999
mfriends 12 11 1998
```

```
# Print out the groups
extract_groups
extract_groups since 1988
extract_groups since 2060
```

Strutture dati scelte per l'implementazione

La struttura dati scelta per implementare la rete di persone e' stata quella del grafo, con un nodo per ogni persona all'interno della rete, e un lato per rappresentare una relazione d'amicizia (etichettato con l'anno in cui e' stata stretta).

In questo modo e' possibile raccogliere le informazioni richieste sulla rete di amicizie in un tempo dell'ordine di `O(|V| * |E|)` – con `E` numero di lati/amicizie nel grafo, `V` numero di nodi/utenti.

Il grafo e' stato implementato in `C`, attraverso liste di adiacenza: un array di *puntatori* a `struct user`, di dimensione fissata `|V|` (il numero massimo di persone che e' possibile inserire nella rete),

```
struct graph {
    struct user ** users;
    size_t nusers;
}

struct user {
    int32_t id;
    char * first_name;
    char * last_name;
    struct linkedlist_friendship * friendships;
    bool _flag; // Useful for later computations
}
```

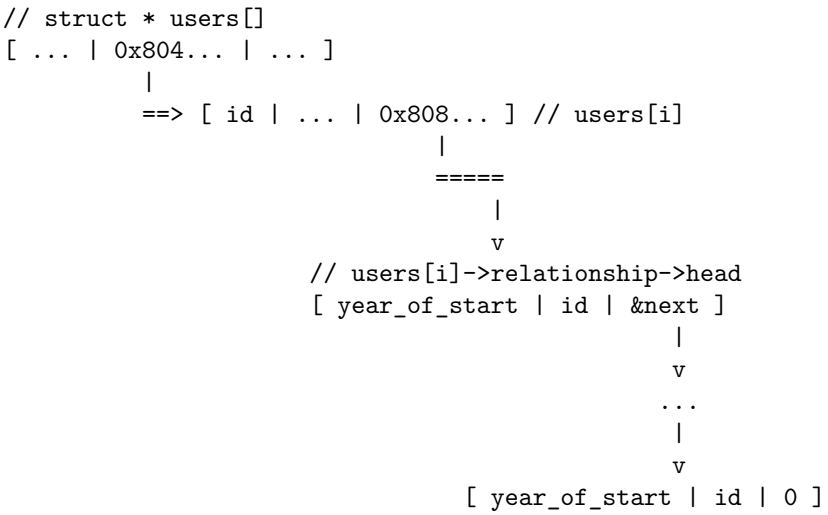
Ogni `struct user` contiene una reference ad una lista di amicizie, cioe' ad una lista di

```
struct node_friendship {
    int32_t year_of_start;
    int32_t id;
    struct node_friendship * next;
}
```

con informazioni sull'anno in cui e' stata instaurata l'amicizia e l'`id` della persona con cui e' stata instaurata.

L'inserimento all'interno della lista avviene in testa[0], quindi in tempo costante, `O(1)`.

Ricapitolando quindi



Per ragioni di raccolta di informazioni, una coda di interi `queue_int32`, e una coda di code `queue_queues_int32` sono risultate necessarie.

Modalita' di accesso e modifica alle strutture

Creazione della rete

Avviene via `new_graph(size_t nusers)`, che restituisce un grafo implementato con liste di adiacenza, cioe' una `struct graph *`.

Aggiunta di una persona alla rete

Passando `add` al prompt, viene aggiunta una nuova persona. Vista l'implementazione, viene aggiunta una reference ad una `struct user` relativa al nuovo utente nell'array che implementa i nodi del grafo, in posizione `id`. Semplicemente,

```
G->users[id] = new_user(id, first_name, last_name);
```

L'accesso a `G->users[id]` avviene in tempo costante, indipendente dal numero di vertici e archi, cosi' come la creazione dell'utente in `new_user()`.

Dunque l'aggiunta di un nuovo utente avviene in tempo `O(1)`.

Ricerca di una persone nella rete

Con `find <id>` e' possibile ricevere informazioni sull'utente con l'id `id`. Questo avviene in tempo costante, e in maniera semplice da scrivere data

l'implementazione:

```
print_user(G->users[id]);
```

Come sopra, dato che l'`id` ha una corrispondenza precisa con la posizione dell'utente nell'array `G->users`, l'accesso all'utente `id` avviene in tempo costante, `O(1)`.

Determinare gruppi di amici

Vista l'implementazione, la ricerca di un gruppo di amici coincide con la ricerca delle componenti connesse in un grafo.

Per l'estrazione delle componenti connesse, la funzione `bfs()`, che esplora il grafo in ampiezza, viene applicata su tutti i nodi del grafo.

Per tenere conto dei nodi gia' attraversati ho usato il campo `_flag` di `struct user` che ha valore `true` se il nodo e' gia' stato visitato.

Per la raccolta delle componenti connesse ho usato invece una coda di code, implementata in `struct queue_queues_int32`.

Quindi ad alto livello, la procedura implementata e'

```
def connected_components(G, cb):
    ret = []
    visited = []
    for vertex in G:
        if vertex.visited: continue
        component = bfs(G, v, cb)
        ret.append(component)
    return ret
# vim: set ft=python:
```

Visto che `connected_components()` accetta come parametro una funzione `cb(x)`, e' possibile vincolare l'esplorazione ai soli archi che superano una certa condizione.

E' immediato quindi implementare la funzionalita' richiesta dal punto `2.5`, cioe' di selezionare i gruppi di amici nati solo *dopo* un anno specificato.

```
int32_t year_of_start = 2013;
bool cb (int32_t x) {
    return x >= year_of_start;
}
qq = connected_components(G, cb);
```

E' possibile calcolare questo risultato passando al prompt `extract_groups since <year>`.

Per quanto riguarda il costo computazionale dell'operazione, qualche nota: l'implementazione dell'algoritmo `bfs()` non presenta differenze rispetto alla formulazione tradizionale, ha un costo del tipo `O(|E|)`.

La procedura `connected_components()` itera sui vertici in `G`, quindi `|V|` volte al piu', applicando `bfs()` su ciascuno.

Una stima del costo dell'intera procedura e' dunque `O(|V| * |E|)`.

Una sessione di esempio

Dato l'input

```
$ cat input2
add foo bar 0
add stack overflow 1
mfriends 0 1 2015

add cat meow 15
mfriends 15 1 1987

add what_the heck 8
add foobarism me 2
mfriends 8 2 2007

add hello world 12
mfriends 8 12 2013

add hexa decimal 11
mfriends 8 11 2008

add hacker who 9
mfriends 8 9 2016

add lonely boy 3

extract_groups
extract_groups since 2008
$
$ cc -m32 -g -O0 -pedantic -Wall main.c -std=c11 -o main
$ cat input2 |./main
Usage: main <max users allowed>
$ cat input2 |./main 20

Group #0:
    bar, foo (0)
```

```
    overflow, stack (1)
    meow, cat (15)

Group #1:
    me, foobarism (2)
    heck, what_the (8)
    who, hacker (9)
    decimal, hexa (11)
    world, hello (12)

Group #2:
    boy, lonely (3)

Group #0:
    bar, foo (0)
    overflow, stack (1)

Group #1:
    me, foobarism (2)

Group #2:
    boy, lonely (3)

Group #3:
    heck, what_the (8)
    who, hacker (9)
    decimal, hexa (11)
    world, hello (12)

Group #4:
    meow, cat (15)
```

---

[0] Il comando `mfriends <id0> <id1> <year_of_start>` potrebbe essere implementato in maniera piu' efficiente.

In questo momento, `mfriends` evita i doppioni andando a controllare se i due utenti sono gia' amici.

```
int32_t other_ix = 0;
struct linkedlist_friendship * shyest_friendships = 0;
if (G->users[i]->friendships->len < G->users[j]->friendships->len) {
    shyest_friendships = G->users[i]->friendships;
    other_ix = j;
}
else {
    shyest_friendships = G->users[j]->friendships;
```

```
    other_ix = i;
}
if (in_linkedlist_friendship(shyest_friendships, other_ix)) {
    printf("#%d and #%d are already friends.\n", i, j);
    continue; // Drop request
}
prepend_linkedlist_friendship(G->users[i]->friendships, \
                                year_of_start, j);
prepend_linkedlist_friendship(G->users[j]->friendships, \
                                year_of_start, i);
```

Il che risulta in un *full scan* di una lista di `struct friendship` – con l'accorgimento minimo che quella che viene controllata e' la piu' corta.

Sarebbe piu' efficiente mantenere la `linkedlist_friendship` ordinata, cosi' da evitare un *full scan* dell'intera lista.

Questo ha chiaramente lo svantaggio di perdere l'inserimento in tempo `O(1)`, ma e' certamente piu' adeguato nel caso si debba pensare ad un'applicazione che supporti un numero "*molto alto*" di utenti (o meglio, con molte *reti di amicizie*).