ΕΘΝΙΚΟ ΚΑΙ ΚΑΠΟΔΙΣΤΡΙΑΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

Εξεταστική Περίοδος Χειμερινού Εξαμήνου
2014 - 2015

Ανάπτυξη Λογισμικού για Πληροφοριακά Συστήματα

Υλοποίηση - Επίπεδο 1

ΟΝΟΜΑ: Γαλούνη Κωνσταντίνα **Α.Μ:** 1115201000034
ΟΝΟΜΑ: Γιαννακέλος Κωνσταντίνος **Α.Μ:** 1115201000029

ΔΙΔΑΣΚΩΝ: Ιωάννης Ιωαννίδης

ΑΘΗΝΑ – 2014

Περιγραφή Υλοποίησης

Η υλοποίηση αποτελείται από τα εξής αρχεία :

Φάκελος graph: **graph.c** **graph.h** **makefile**

Φάκελος queue: **queue.c** **queue.h**

makefile

Εχούν υλοποιηθεί τα αρχεία για τον έλεγχο της υλοποίησης :

Φάκελος main: **main1.c** **main_functions.c**

main2.c **main_functions.h**

main3.c

Παρατηρήσεις:

- Η μεταγλώττιση μέσω make θα μεταγλωττίσει όλα τα αρχεία οπότε η χρήση της κάθε main γίνεται άμεσα (ανάλογα με τη main της επιλογής μας, εκτελούμε ./main1 , ./main2 ή ./main3).

- Τα αρχεία στο φάκελο main έχουν βασιστεί και προσαρμοστεί στις main που δόθηκαν (ίδιες συναρτήσεις, ίδια παραδείγματα κλπ : main1.c), και σε δικά μας παραδείγματα για την επιβεβαίωση ορθότητας της άσκησης (main2.c : σύνθετος γράφος 15 κόμβων, main3.c : γράφος σειριακά συνδεδεμένων 1000000 κόμβων).

- Η υλοποίηση αναπτύχθηκε και δοκιμάστηκε σε Ubuntu 12.04 και στα Linux της σχολής.

Παρακάτω θα γίνει μία περιγραφή των δομών δεδομένων που χρησιμοποιήθηκαν καθώς και των συναρτήσεων που υλοποιήθηκαν.

Δομές Δεδομένων

Graph : Η δομή αυτή αναπαριστά τον γράφο του κοινωνικού δικτύου. Περιέχει τον πίνακα κατακερματισμού ο οποίος αποθηκεύει τις εγγραφές, καθώς και όλα τα απαραίτητα δεδομένα για τη χρήση του γραμμικού κατακερματισμού. Ο πίνακας κατακερματισμού αποτελεί έναν πίνακα με δείκτες προς **Bucket.** Η δομή είναι η παρακάτω:

```
typedef struct Graph{
    void **ht;                  // hash table
    int m;                      // initial hash table size
    int c;                      // bucket size
    int p;                      // pointer to split bucket
    int i;                      // splitting round
    int size;                   // size of hash table
}Graph;
```

Bucket : Η δομή αυτή αναπαριστά τους κάδους κατακερματισμού. Αποτελείται από έναν πίνακα με void* , δηλαδή έναν πίνακα με δείκτες προς τις εγγραφές – κόμβους, το μέγεθος του και τον αριθμό των εγγραφών τον οποίο περιέχει. Η δομή είναι η παρακάτω:

```
typedef struct Bucket{
    void **cells;
    int size;
    int nrecs;
}Bucket;
```

Properties : Η δομή αυτή αναπαριστά τις ιδιότητες είτε για μία ακμή είτε για μία εγγραφή. Αποτελεί έναν πίνακα από void* ώστε να είναι δυνατή η αποθήκευση δεδομένων κάθε τύπου. Η δομή είναι η παρακάτω:

```
typedef struct Properties{
```

```
    void **properties;
}Properties;
```

Edge : Η δομή αυτή αναπαριστά τις ακμές ανάμεσα στους κόμβους. Αποτελεί μία λίστα η οποία κρατάει το id του κόμβου στο άλλο άκρο, καθώς και μία δομή **Properties** η οποία αναπαριστά τη σχέση της ακμής. Η δομή είναι η παρακάτω:

```
typedef struct Edge{
        int id;
        Properties      *props;
        struct Edge     *next;
}Edge;
```

Record : Η δομή αυτή αναπαριστά τον κόμβο – εγγραφή του γράφου. Αποτελείται από το id της εγγραφής, μία λίστα ακμών **Edge** και μία δομή ιδιοτήτων **Properties.** Η δομή είναι η παρακάτω:

```
typedef struct Record{
        int id;
        Edge   *edges;
        Properties *props;
}Record;
```

Pair : Αποτελεί τη δομή που επιστρέφει η συνάρτηση **next.** Περιέχει τον **κόμβο – target** και την απόσταση μεταξύ του **source – target** .

```
struct pair {
    int ID;
    int distance;
};
typedef struct pair Pair;
```

ResultSet : Αποτελεί τη δομή που επιστρέφει η συνάρτηση **reachNodeN**, η οποία κρατάει τα κατάλληλα δεδομένα για τον υπολογισμό των αποστάσεων μέσω της συνάρτησης next. Συγκεκριμένα κρατάει το id του **κόμβου – source**, τον αριθμό του κάδου και τον αριθμό της εγγραφής (για το συγκεκριμένο κάδο) η οποία είναι η επόμενη προς υπολογισμό.

```
typedef struct ResultSet{
    Graph *graph;
    int from;
    int hash;
    int record;
}ResultSet;
```

Queue_Node : Αποτελεί τη δομή που αναπαριστά τους κόμβους για τις ουρές του Bidirectional Breadth First Search, η οποία περιέχει ένα δείκτη σε Record (στα properties φυλάσσεται η απόσταση του κόμβου από τον **κόμβο - source** ή **target** αντίστοιχα) και ένα δείκτη next για τη σύνδεση του ενός κόμβου της ουράς με τον επόμενό του.

```
typedef struct Queue_Node{
    Record *node;
    struct Queue_Node *next;
}Queue_Node;
```

Queue : Αποτελεί τη δομή που αναπαριστά μία ουρά, με δείκτες στον πρώτο και τον τελευταίο κόμβο, ώστε να γίνεται αποδοτικά η εισαγωγή (enqueue) και η εξαγωγή (dequeue) ενός κόμβου. Οι κόμβοι είναι τύπου **Queue_Node** όπως αναφέρθηκε παραπάνω.

```
typedef struct Queue{
    Queue_Node *first;
    Queue_Node *last;
}Queue;
```

Συναρτήσεις Γράφου

```
Bucket* create_Bucket(int c)
```

Η συνάρτηση αυτή δημιουργεί έναν κάδο μεγέθους c, και επιστρέφει έναν δείκτη σε αυτόν.

```
Graph* create_Graph(int m, int c)
```

Η συνάρτηση αυτή δημιουργεί τη δομή που αναπαριστά το γράφο, κάνοντας χρήση της συνάρτησης **create_Bucket** για τη δημιουργία των αρχικών m buckets μεγέθους c. Επιστρέφει έναν δείκτη στο γράφο.

```
int* destroyGraph(Graph **g)
```

Η συνάρτηση αυτή αποδεσμεύει τη δομή που αναπαριστά το γράφο, κάνοντας free όλα τα buckets, τα records, τα properties και τα edges. Επιστρέφει 1 σε περίπτωση επιτυχίας.

```
int insertNode(Record *r , Graph *g)
```

Η συνάρτηση εισάγει έναν κόμβο – εγγραφή στο γράφο. Αρχικά υπολογίζεται ο αριθμός του κάδου στον οποίο αντιστοιχεί μέσω της hash_function και του search scheme. Αν η εγγραφή δε χωράει στον κάδο, τότε ο πίνακας κατακερματισμού περνάει από τη φάση του split. Πιο συγκεκριμένα δημιουργείται νέος κάδος, αυξάνεται το μέγεθος του hash table με τη χρήση της realloc και έπειτα γίνεται ανακατανομή των εγγραφών του κάδου που δείχνει ο δείκτης p. Σε περίπτωση που η εγγραφή προορίζεται στον κάδο που γίνεται split, πρώτα θα γίνει το split και μετά η εισαγωγή της στον κάδο (αφού περάσει από τον έλεγχο του search scheme ξανά). Αν κατά την ανακατανομή, γεμίσει κάποιος από τους δύο κάδους, τότε αυξάνεται το μέγεθος του με τη χρήση της realloc κατά c κελιά.

Στην περίπτωση όπου η εγγραφή χωράει κανονικά στον κάδο, η εγγραφή περνά πρώτα από μία συνάρτηση ταξινόμησης και αποθηκεύεται στην κατάλληλη θέση ταξινομημένη. Συγκεκριμένα υπάρχει η εξής συνάρτηση :

```
int sort(Bucket *b , int newid)
```

Η συνάρτηση αυτή υλοποιεί μία **Insertion Sort** με την έννοια ότι δέχεται ένα **Bucket** και το id της εγγραφής προς εισαγωγή και επιστρέφει τη θέση – index στον πίνακα του **Bucket**. Συγκεκριμένα αφού υπολογίσει το index της θέσης για την καινούρια εγγραφή, μετακινεί τις θέσεις του **Bucket** μέσω της memmove ώστε να αδειάσει η θέση για την εισαγωγή της εγγραφής.

```
Record* lookupNode(int id, Graph *g)
```

Η συνάρτηση αυτή δέχεται ένα id και επιστρέφει έναν δείκτη προς την εγγραφή του γράφου με αυτό το id. Χρησιμοποιεί **Binary Search** αφού οι εγγραφές στα **Bucket** είναι ήδη ταξινομημένες από την εισαγωγή.

```
int insertEdge(int id, Edge *e, Graph *g)
```

Η συνάρτηση αυτή με τη χρήση της lookupNode, βρίσκει τον κόμβο με το συγκεκριμένο id και τοποθετεί την ακμή στην αρχή της λίστας **Edge**.

```
int reachNode1(int start, int end, Graph *g)
```

Η συνάρτηση αυτή βρίσκει την ελάχιστη απόσταση μεταξύ δύο κόμβων start και end. Σε περίπτωση που υπάρχει μονοπάτι σύνδεσης επιστρέφει την απόσταση του συντομότερου μονοπατιού, ενώ αν οι κόμβοι δε συνδέονται επιστρέφει INFINITY_REACH_NODE το οποίο έχει οριστεί ως ο μεγαλύτερος ακέραιος στο αρχείο επικεφαλίδας. Για την υλοποίηση δημιουργεί δύο δομές **Graph**, για την αποθήκευση των κόμβων που έχει συναντήσει κάθε μονοπάτι (εκμεταλλεύεται τον ήδη υλοποιημένο πίνακα κατακερματισμού για βέλτιστη αναζήτηση) και δύο δομές **Queue**, για την αποθήκευση των κόμβων που πρόκειται να εξερευνηθούν. Συγκεκριμένα, υλοποιείται Bidirectional search , με εκτέλεση Breadth First Search από τον **κόμβο – source** στον **κόμβο – target** και αντίστροφα και γι ‘αυτό καλείται η συνάρτηση bfs (αρχείο queue.c).

```
void bfs(Queue **queue, Graph *visited1, Graph *visited2,
Record **node1, Record **node2, Graph *g, int *min)
```

Η συνάρτηση αυτή υλοποιεί τον BFS αλγόριθμο, λαμβάνοντας υπόψη και τις συνθήκες τερματισμού, δεδομένου ότι χρησιμοποιείται στα πλαίσια bidirectional bfs. Τα ορίσματα queue, visited1 και node1 αναφέρονται στο μονοπάτι που εξετάζεται κάθε φορά και είναι η ουρά των υπό εξέταση κόμβων, ο πίνακας κατακερματισμού στον οποίο βρίσκονται οι κόμβοι που έχει συναντήσει το μονοπάτι και ο κόμβος που θα «επιστρέψει» η συνάρτηση ως ο τελευταίος που εξερευνήθηκε. Τα ορίσματα visited2 και node2 αφορούν στο άλλο μονοπάτι. Το g είναι ο αρχικός μας γράφος με όλους τους κόμβους και τα χαρακτηριστικά τους και τέλος,

το min είναι ένας ακέραιος ώστε να καθίσταται δυνατή η σύγκριση της απόστασης πιθανών λύσεων. Εσωτερικά, εξάγει τον πρώτο κόμβο από την ουρά, αν αυτός υπάρχει, και για κάθε ακμή του, αν δεν υπάρχει στη δομή visited1 ο κόμβος που καταλήγει η ακμή, τον δημιουργεί και τον εισάγει τόσο στη visited1, όσο και στην ουρά. Παράλληλα, αν ο κόμβος αυτός υπάρχει στη δομή visited2 (το άλλο μονοπάτι τον έχει συναντήσει), τότε έχει βρεθεί μία πιθανή λύση. Για να εξασφαλιστεί η ελάχιστη απόσταση, εξετάζουμε όλους τους κόμβους της ουράς με ίδια απόσταση με τον τρέχοντα κόμβο (current), με αναδρομική κλήση της συνάρτησης bfs.

```
ResultSet* reachNodeN(int id, Graph *g)
```

Η συνάρτηση δέχεται ένα id και επιστρέφει μία δομή **ResultSet,** την οποία θα χρησιμοποιήσει η συνάρτηση next.

```
int next(ResultSet *r, Pair *p)
```

Η συνάρτηση δέχεται μία δομή **ResultSet** και κάθε κλήση της επιστρέφει μία απόσταση από έναν άλλο διαφορετικό κόμβο στο γράφο. Συγκεκριμένα χρησιμοποιεί την reachNode1 για τον υπολογισμό της απόστασης, όπου τα ορίσματά της είναι εύκολα προσπελάσιμα μέσω της δομής **ResultSet** η οποία κρατάει τη θέση του επόμενου κόμβου στο hash_table.

Βοηθητικές Συναρτήσεις & Συναρτήσεις για τη main

Φάκελος graph

```
int power(int i)
```

Η συνάρτηση υλοποιεί την ύψωση του αριθμού 2 στη δύναμη του ακέραιου i που δέχεται ως όρισμα. Έχει δημιουργηθεί για την αποφυγή της συνάρτησης pow και τη χρήση της βιβλιοθήκης math.h.

```
int hash_function(int k, int m, int i)
```

Η συνάρτηση υλοποιεί τη συνάρτηση κατακερματισμού που δόθηκε για το γραμμικό κατακερματισμό h(k) = k % (2^i)*m.

```
void overflow(Bucket *b, int c)
```

Η συνάρτηση δέχεται ως όρισμα ένα **Bucket** και αυξάνει το μέγεθός του με τη χρήση της realloc κατά c κελιά. Ονομάζεται overflow, αφού καλείται κάθε φορά που ένα bucket υπερχειλίζει και απαιτείται overflow page.

```
void print_ht(Graph *g)
```

Η συνάρτηση δέχεται ως όρισμα ένα γράφο και εκτυπώνει όλες τις εγγραφές του, αναπαριστώντας το bucket που βρίσκεται κάθε μία, καθώς και πληροφορίες σχετικά με το μέγεθος και τον αριθμό των εγγραφών κάθε bucket.

Φάκελος queue

```
void create_queue(Queue **q)
```

Η συνάρτηση δεσμεύει χώρο για μία ουρά, και θέτει τους δείκτες στον πρώτο και τον τελευταίο κόμβο σε NULL, εφόσον η ουρά μόλις δημιουργήθηκε και είναι κενή.

```
void destroy_queue(Queue **q)
```

Η συνάρτηση εξάγει όλους τους κόμβους από την ουρά και έπειτα αποδεσμεύει το χώρο που δεσμεύθηκε αρχικά από την create_queue.

```
void enqueue(Queue **q, Record *node)
```

Η συνάρτηση εισάγει έναν κόμβο node στο τέλος της ουράς q.

```
Record* dequeue(Queue **q)
```

Η συνάρτηση εξάγει έναν κόμβο από την αρχή της ουράς q και επιστρέφει ένα δείκτη σε αυτόν.

```
int empty(Queue *q)
```

Η συνάρτηση επιστρέφει 1 εάν η ουρά είναι κενή, και 0 διαφορετικά.

Φάκελος main

```
Record* create_Node(int id, Properties *p)
```

Η συνάρτηση δεσμεύει χώρο για μία εγγραφή (**Record**), και αρχικοποιεί τα πεδία id και props με βάση τα ορίσματα που δέχεται. Επιστρέφει την εγγραφή που δημιούργησε.

```
Record* setPersonProperties(int id, char *name, char *surname,
int age)
```

Η συνάρτηση δημιουργεί μία δομή **Properties**, καλώντας τη συνάρτηση create_Properties με όρισμα PERSON_PROPERTIES_NUM, το οποίο έχει οριστεί στο αντίστοιχο αρχείο επικεφαλίδας. Έπειτα, αρχικοποιεί τα πεδία της δομής αυτής με τα ορίσματα που δέχεται (name, surname, age) και τέλος, δημιουργεί μία εγγραφή μέσω της συνάρτησης create_Node, δίνοντας ως ορίσματα το id που δέχτηκε και τη δομή που δημιούργησε. Επιστρέφει την εγγραφή αυτή.

```
void printPersonProperties(Record *r)
```

Η συνάρτηση εκτυπώνει όλα τα χαρακτηριστικά της εγγραφής r, συμπεριλαμβανομένου και του id της.

```
Edge* setEdgeProperties(int start, int end, char *property, int
weight)
```

Λειτουργεί όπως η συνάρτηση setPersonProperties, μόνο που δημιουργεί μία δομή **Properties**, καλώντας τη συνάρτηση create_Properties με όρισμα PERSON_REL_PROPERTIES_NUM, και δημιουργεί μία ακμή Edge καλώντας τη συνάρτηση create_Edge, την οποία και επιστρέφει.

```
Properties* createProperties(int size)
```

Δημιουργεί μία δομή **Properties** και δεσμεύει χώρο για size διαφορετικές ιδιότητες. Επιστρέφει τη δομή που δημιούργησε.

```
void setStringProperty(char *property, int index, Properties*
p)
```

Θέτει στη θέση index του πίνακα properties της δομής p, το όρισμα property, αφού πρώτα γίνει η κατάλληλη δέσμευση χώρου.

```
void setIntegerProperty(int property, int index, Properties* p)
```

Θέτει στη θέση index του πίνακα properties της δομής p, το όρισμα property, αφού πρώτα γίνει η κατάλληλη δέσμευση χώρου.

```
char * getStringProperty(int index, Properties* p)
```

Επιστρέφει τα περιεχόμενα της θέσης index του πίνακα properties της δομής p, όταν γνωρίζουμε ότι είναι συμβολοσειρά.

```
int getIntegerProperty(int index, Properties* p)
```

Επιστρέφει τα περιεχόμενα της θέσης index του πίνακα properties της δομής p, όταν γνωρίζουμε ότι είναι ακέραιος.