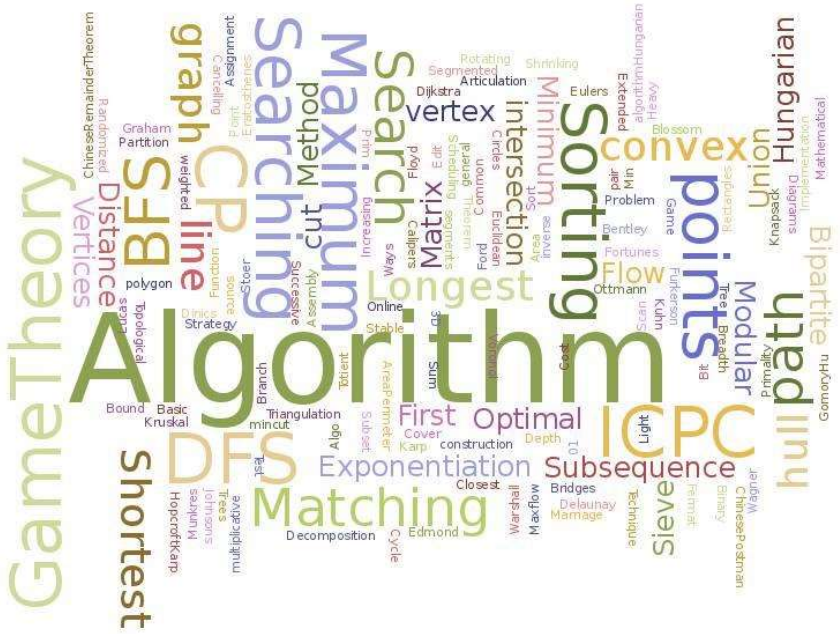
ΤΕΧΝΟΛΟΓΙΕΣ ΥΛΟΠΟΙΗΣΗΣ ΑΛΓΟΡΙΘΜΩΝ



Τελική εργασία

Επιβλέπων καθηγητής: Χρήστος Ζαρολιάγκης

Παναγιώτης Καββαδίας

ΑΜ:1054350

Περιεχόμενα

Περίληψη

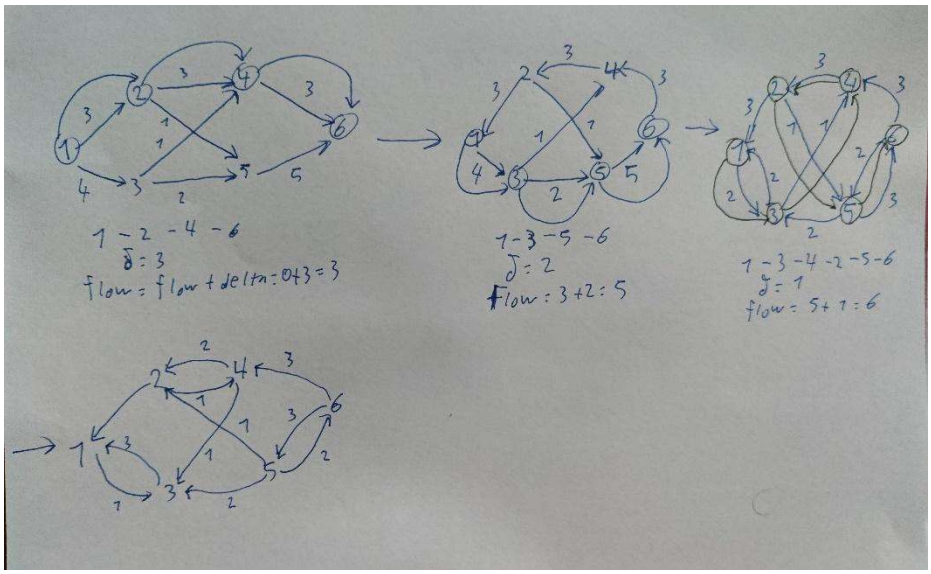
Για την εργασία κλήθηκα να υλοποιήσω τον αλγόριθμο μέγιστης ροής shortest augmenting path όπως περιγράφεται στο βιβλίο “Network Flows” των R. Ahuja, T. Magnanti, J. Orlin. Η υλοποίηση έγινε με την χρήση της βιβλιοθήκης Boost και η πειραματική αξιολόγηση έγινε με σύγκριση των αποτελεσμάτων με τα αποτελέσματα του αλγορίθμου μέγιστης ροής της βιβλιοθήκης LEDA. Οι κώδικες εκτελέστηκαν στο σύστημα Διογένης του ΤΜΗΥΠ.

Ανάλυση αλγορίθμου

Ο αλγόριθμος μέγιστης ροής αυξάνει τη ροή κατά μήκος του μικρότερου μονοπατιού από έναν κόμβο s προς έναν κόμβο προορισμό t στο δίκτυο. Η εύρεση των μικρότερων μονοπατιών γίνεται μέσω αναζήτησης BFS. Ο αλγόριθμος προχωράει αυξάνοντας την ροή κατά μήκος αποδεκτών μονοπατιών. Ένα αποδεκτό μονοπάτι κατασκευάζεται προσθέτοντας ένα τόξο κάθε φορά.

Ο αλγόριθμος διατηρεί ένα μερικώς αποδεκτό μονοπάτι, δηλαδή ένα μονοπάτι από τον αρχικό κόμβο s σε κάποιον ενδιάμεσο κόμβο i που αποτελείται αποκλειστικά από αποδεκτά τόξα. Ο αλγόριθμος εκτελεί επαναληπτικά βήματα προώθησης (advance) ή οπισθοχώρησης (retreat). Αν ο κόμβος i έχει αποδεκτό τόξο (i,j) ο αλγόριθμος προωθείτε και προσθέτει το (i,j) στο αποδεκτό μονοπάτι, διαφορετικά οπισθοχωρεί κατά ένα τόξο. Επαναλαμβάνουμε τα βήματα έως ότου βρεθεί ένα μονοπάτι προς τον κόμβο προορισμού t (sink node) και τότε πραγματοποιούμε αύξηση. Αυτή η διαδικασία επαναλαμβάνεται έως ότου η ροή είναι η μέγιστη.

Ένα παράδειγμα εκτέλεσης βήμα-βήμα του αλγορίθμου φαίνεται παρακάτω:



Τα μικρότερα μονοπάτια βρίσκονται μέσω της αναζήτησης BFS σε χρόνο O(m) και μπορούν να πραγματοποιηθούν το πολύ O(mn) αυξήσεις. Έτσι ο αλγόριθμος έχει χρόνο εκτέλεσης χειρότερης περίπτωσης $O(nm^2)$.

Λεπτομέρειες υλοποίησης

Η υλοποίηση έγινε με βάση τον ψευδοκώδικα που παρατίθεται παρακάτω:

```
algorithm shortest augmenting path;
begin
    x := 0;
    obtain the exact distance labels d(i);
    i := s;
    while d(s) < n do
    begin
        if i has an admissible arc then
            begin
                advance(i);
                if i = t then augment and set i = s
            end
        else retreat(i)
    end;
end;
```

Figure 7.5 Shortest augmenting path algorithm.

```
procedure advance(i);
begin
    let (i, j) be an admissible arc in A(i);
    pred(j) := i and i := j;
end;
```

(a)

```
procedure retreat(i);
begin
    d(i) := min{d(j) + 1 : (i, j) ∈ A(i) and r_ij > 0};
    if i ≠ s then i := pred(i);
end;
```

(b)

```
procedure augment;
begin
    using the predecessor indices identify an augmenting
    path P from the source to the sink;
    δ := min{r_ij : (i, j) ∈ P};
    augment δ units of flow along path P;
end;
```

R. Ahuja, T. Magnanti, J. Orlin, "Network Flows", Prentice-Hall, 1993 σελ.215

Στην αρχή γίνονται οι απαραίτητοι ορισμοί (typedefinition) των δομών που θα χρησιμοποιηθούν στην συνέχεια.

```
struct EdgeProperties {
    int capacity;
};

typedef adjacency_list<vecS, vecS, directedS, no_property, EdgeProperties> Graph;
typedef graph_traits<Graph>::vertex_descriptor Vertex;
typedef graph_traits<Graph>::edge_descriptor Edge;
typedef graph_traits<Graph>::vertex_iterator Vertex_iter;
typedef graph_traits<Graph>::edge_iterator Edge_iter;
typedef property_map<Graph, vertex_index_t>::type vertexIndexMap;
typedef vector<int> distVector;
typedef iterator_property_map<distVector::iterator, vertexIndexMap> distMap;
typedef vector<Vertex> predVector;
typedef iterator_property_map<predVector::iterator, vertexIndexMap> predMap;
typedef list<Edge> list_of_edges;
typedef vector<list_of_edges> aVector;
typedef iterator_property_map<aVector::iterator, vertexIndexMap> Map_t;

/*
* BFS visitor class
*/
class bfs_discovery_visitor : public bfs_visitor <> {
};
```

Οι distVector, predVector και οι αντίστοιχοι maps χρησιμοποιούνται για την διατήρηση των αποστάσεων μεταξύ κόμβων και της γνώσης σχετικά με τον προηγούμενο κόμβο, που απαιτείται όταν υπάρχει οπισθοχώρηση(retreat operation). Ο aVector με το αντίστοιχο map χρησιμοποιείται για την διατήρηση των ακμών που ανήκουν στο αποδεκτό μονοπάτι. Η κλάση bfs_discovery_visitor κληρονομεί την bfs_visitor της Boost και μέσω αυτής αργότερα δημιουργείται ένα αντικείμενο το οποίο θα χρησιμοποιηθεί ως παράμετρος στην κλήση της συνάρτησης breadth_first_search της Boost.

Ακολουθεί forward declaration των συναρτήσεων advance, augment και retreat καθώς καλούνται πριν οριστούν αφού ο κώδικας ακολουθεί την δομή του δοθέντος ψευδοκώδικα.

Για την κατασκευή του γραφήματος έχει επιλεγεί ένας «χειροκίνητος» τρόπος. Το πρόγραμμα διαβάζει ένα .txt αρχείο όπου κάθε γραμμή αποτελείται από δυο στήλες. Το πρόγραμμα κατασκευάζει ακμή μεταξύ των κόμβων που υποδεικνύουν οι στήλες. Για παράδειγμα αν στην πρώτη γραμμή η πρώτη στήλη έχει τον αριθμό 2 και η δεύτερη τον αριθμό 5 θα κατασκευαστεί μια ακμή μεταξύ των κόμβων 1 και 5. Αυτή η επιλογή κάνει την κατασκευή ενός συνόλου δεδομένων για έλεγχο πιο δύσκολη και χρονοβόρα σε σχέση με το να δημιουργούνταν ένα τυχαίο γράφημα μέσω βιβλιοθηκών της Boost. Όμως δίνει μεγαλύτερο έλεγχο καθώς γνωρίζω ακριβώς ποιο γράφημα διατρέχει ο αλγόριθμος και επιπλέον διευκολύνει την σύγκριση με τα αποτελέσματα της LEDA καθώς και η LEDA θα εξετάσει ακριβώς το ίδιο γράφημα. Στον κώδικα έχει προβλεφθεί χειρισμός και για την περίπτωση που δεν δοθεί κάποιο .txt αρχείο σαν όρισμα ή το δοθέν αρχείο δεν υπάρχει.

```
/*
* Enter max capacity
*/
cout << "Enter max capacity ";
cin >> max_capacity;
cout << endl;

/*
* Randomly assign capacity on each edge between 1 and max_capacity
*/
srand ( (unsigned)time(NULL));

for(tie(ei, ei_end) = edges(g); ei != ei_end; ++ei) {
    g[*ei].capacity = rand() % ((max_capacity - 1) + 1) + 1;
}
```

Έπειτα ζητείται από τον χρήστη να εισάγει την μέγιστη χωρητικότητα που θέλει να υπάρχει και αναθέτει σε κάθε ακμή χωρητικότητα από 1 έως τη μέγιστη.

Έπειτα με τη χρήση της συνάρτησης make_iterator_property_map της Boost δημιουργούνται maps για απόσταση, προηγούμενο κόμβο και A(i). Έπειτα θεωρώ ότι ο αρχικός κόμβος θα είναι ο s και ο κόμβος προορισμός(sink) θα είναι ο t. Το πρόγραμμα εντοπίζει τους συγκεκριμένους κόμβους. Ο κόμβος που δεν έχει εισερχόμενες ακμές είναι ο αρχικός ενώ ο κόμβος που δεν έχει εξερχόμενες ακμές είναι ο τελικός.

```
for(pair<Vertex_iter, Vertex_iter> vIter = vertices(g); vIter.first != vIter.second; ++vIter.first) {
    inEdges = 0;
    outEdges = out_degree(*vIter.first, g);

    for(tie(ei, ei_end) = edges(g); ei != ei_end; ++ei) {
        if(*vIter.first == target(*ei, g)) {
            ++inEdges;
        }
    }

    if(inEdges == 0) {
        s = *vIter.first;
    }
    else if(outEdges == 0) {
        t = *vIter.first;
    }
}
```

Ύστερα εκτελείται αναζήτηση BFS και υπολογίζονται οι αποστάσεις μεταξύ των κόμβων. Επίσης η λίστα του A(i) αρχικοποιείται με όλες τις ακμές που εξέρχονται από το i. Όσο d(s)<n υπολογίζονται οι εξερχόμενες ακμές. Αν υπάρχει αποδεκτό μονοπάτι ο αλγόριθμος προωθείται (advance) και αν μετά την προώθηση ο i κόμβος έχει γίνει ίσως με τον t, δηλαδή ο αλγόριθμος καταλήξει στον τελικό κόμβο, γίνεται αύξηση (augment). Εάν πάλι δεν υπάρχει αποδεκτό μονοπάτι ο αλγόριθμος εκτελεί οπισθοχώρηση (retreat).

```
/*
* If i has admissible arc
*/
if((outEdges > 0) && (dist[j] < dist[i])) {
    i = advance(i, j, *ei, pred);

    if(i == t) {
        augment(g, s, t, pred, max_capacity, A_Map);
        i = s;
    }
}
else {
    i = retreat(i, s, A_Map, pred, dist, g, n);
}
```

Η συνάρτηση advance είναι αρκετά απλή. Το i ισούται πλέον με j και σαν πρόγονος του j θεωρείται το i.

```
Vertex advance(Vertex i, Vertex j, Edge e, predMap& pred) {
    pred[j] = i;
    i = j;

    return i;
}
```

Στην συνάρτηση retreat η απόσταση d(i) γίνεται το ελάχιστο της απόσταση d(j) + 1 όπως στον ψευδοκώδικα που δίνεται. Αν το i δεν είναι ο αρχικός κόμβος οπισθοχωρούμε και πλέον το i ισούται με τον πρόγονο του.

```
Vertex retreat(Vertex i, Vertex s, Map_t& A_Map, predMap& pred, distMap& dist, Graph& g, int n) {

    int min;
    Vertex j;

    min = n;

    for(list<Edge>::iterator itr_a = A_Map[i].begin() ; itr_a != A_Map[i].end(); ++itr_a) {
       if((dist[target(*itr_a, g)] < min) && (g[*itr_a].capacity > 0)) {
           min = dist[target(*itr_a, g)];
        }
    }

    dist[i] = min + 1;

    if(i != s) {
        i = pred[i];
    }

    return i;
}
```

Στην συνάρτηση augment ο αλγόριθμος βρίσκει το augmenting path από τον αρχικό κόμβο ως τον τελικό (sink) χρησιμοποιώντας τους προγόνους.

Γνωρίζοντας ότι δ = min{r:(i, j)∈ P αυξάνουμε δ μονάδες ροής κατά μήκος του μονοπατιού P.

Πειραματική αξιολόγηση

Για την εκτέλεση του κώδικα υπάρχει Makefile. Αρκεί η εκτέλεση της εντολής make. Η μεταγλώττιση (compile) γίνεται με το μέγιστο επίπεδο βελτιστοποίησης(-O3). Μετά, με την εντολή ./run filename.txt όπου filename.txt το αρχείο που περιγράφει την δομή του γραφήματος εκτελείται ο κώδικας. Με την εντολή ./ledabenchmark filename.txt εκτελείται η συνάρτηση MAX_FLOW() της LEDA.

Για αυτό υπάρχει ο κώδικας στο αρχείο ledabenchmark.cpp. Σε αυτό το αρχείο διαβάζεται το .txt αρχείο και δημιουργείται γράφημα με συναρτήσεις της LEDA. Ύστερα εκτελείται η συνάρτηση MAX_FLOW της LEDA και μετράμε τον χρόνο. Ο χρόνος μετράται με την χρήση της δομής timespec που βρίσκεται στην βιβλιοθήκη time.h της C++.

Για την πειραματική αξιολόγηση δημιουργήθηκαν δυο .txt αρχεία, τα small.txt και big.txt. Το πρώτο περιλαμβάνει ένα πολύ μικρό γράφημα, το δεύτερο ένα μεγαλύτερο. Παρακάτω φαίνονται αποτελέσματα από την εκτέλεση τους. Για την πειραματική αξιολόγηση θα δοκιμάσουμε με μέγιστη χωρητικότητα 5 και 10.

```
[pkavvadias@diogenis alg_project]$ ./run small.txt

Enter max capacity 5

Time spent: 0.00610676 seconds
The maximum flow is: 5
[pkavvadias@diogenis alg_project]$ ./ledabenchmark small.txt

Enter max capacity 5

Time spent: 0.00945736 seconds
The maximum flow is: 5
[pkavvadias@diogenis alg_project]$ ./run small.txt

Enter max capacity 10

Time spent: 0.00659915 seconds
The maximum flow is: 14
[pkavvadias@diogenis alg_project]$ ./ledabenchmark small.txt

Enter max capacity 10

Time spent: 0.00779336 seconds
The maximum flow is: 8
```

Όπως βλέπουμε ο αλγόριθμος που υλοποίησα στη Boost είναι πιο γρήγορος από την συνάρτηση MAX_FLOW της LEDA τόσο για max capacity 5 όσο και για max capacity 10.

```
[pkavvadias@diogenis alg_project]$ ./run big.txt

Enter max capacity 5

Time spent: 0.0262062 seconds
The maximum flow is: 6
[pkavvadias@diogenis alg_project]$ ./ledabenchmark big.txt

Enter max capacity 5

Time spent: 0.00786794 seconds
The maximum flow is: 5
[pkavvadias@diogenis alg_project]$ ./run big.txt

Enter max capacity 10

Time spent: 0.00697797 seconds
The maximum flow is: 7
[pkavvadias@diogenis alg_project]$ ./ledabenchmark big.txt

Enter max capacity 10

Time spent: 0.00773204 seconds
The maximum flow is: 10
```

Για το μεγάλο dataset διαπιστώνω ότι για max capacity 5 ο αλγόριθμος υλοποιημένος στην Boost είναι σημαντικά πιο γρήγορος από την MAX_FLOW της LEDA. Όμως για max capacity 10 ο αλγόριθμος σε Boost είναι πιο γρήγορος από τον MAX_FLOW της LEDA. Αυτό λογικά οφείλεται στην υλοποίηση της LEDA. Όπως μπορούμε να δούμε από το documentation της LEDA, ο αλγόριθμος μέγιστης ροής που υλοποιεί έχει χρόνο εκτέλεσης $O(n^3)$ ενώ ο αλγόριθμος που υλοποίησα χρόνο $O(nm^2)$.