

Τεχνολογίες Υλοποίησης Αλγορίθμων

Ενότητα 6

Χρήστος Δ. Ζαρολιάγκης

Καθηγητής

Τμήμα Μηχ/κων Υπολογιστών & Πληροφορικής

Πανεπιστήμιο Πατρών

email: `zaro@ceid.upatras.gr`

Ενότητα 6

- Μέγιστη Ροή
 - Γενικά
 - Ελεγκτής Ορθότητας
 - Αλγόριθμος Προροής-Ώθησης και Υλοποίηση
- Αριθμητικοί Τύποι και Ορθότητα Αλγορίθμων Γραφημάτων & Δικτύων

Μέγιστη Ροή - Γενικά

- Διγράφημα $G = (V, E)$ με συνάρτηση χωρητικότητας $cap : E \rightarrow \mathbb{R}_{\geq 0}$ και δύο διακεκριμένες κορυφές s και t .
- $In(v) = \{e : target(e) = v\}$, $Out(v) = \{e : source(e) = v\}$, $\forall v \in V$.
- (s, t) -ροή : συνάρτηση $f : E \rightarrow \mathbb{R}_{\geq 0}$ για την οποία:
 - (1) $0 \leq f(e) \leq cap(e) \quad \forall e \in E$
 - (2) $excess(v) = \sum_{e \in Out(v)} f(e) - \sum_{e \in In(v)} f(e) = 0 \quad \forall v \in V - \{s, t\}$
- Τιμή ροής $|f| = excess(t)$.
- Μέγιστη Ροή : ροή που μεγιστοποιεί την $|f|$.

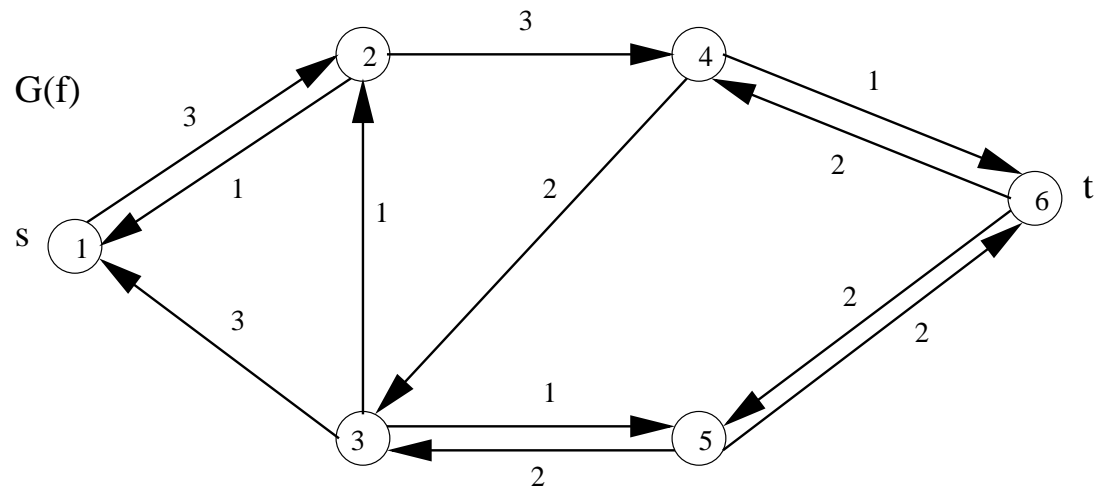
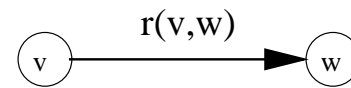
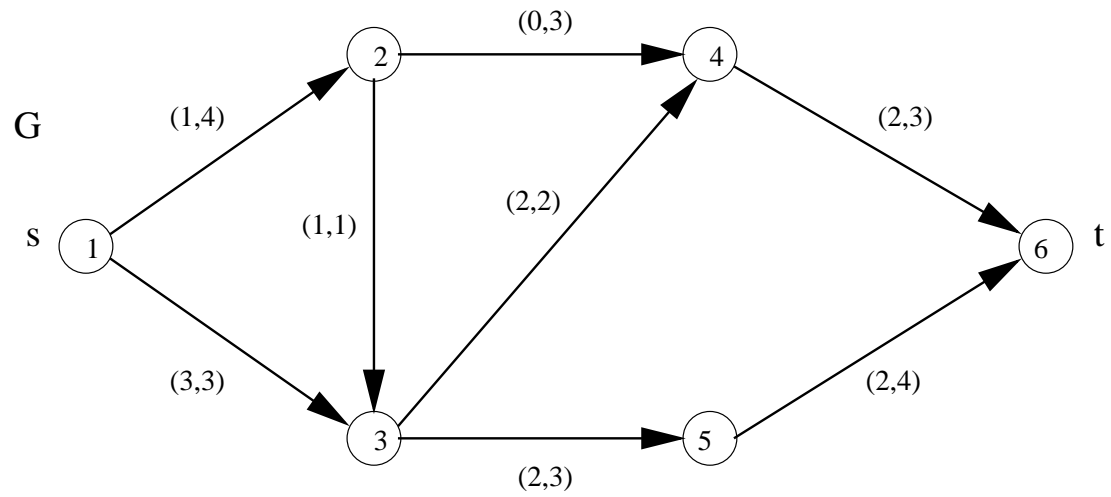
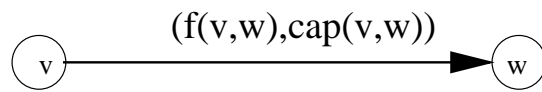
- (s, t) -τομή : σύνολο S κορυφών με $s \in S$ και $t \in T = V - S$.
Έστω $(S, T) = \{e : source(e) \in S, target(e) \in T\}$.

- Χωρητικότητα τομής :

$$cap(S) = \sum_{e \in (S, T)} cap(e)$$

- Κορεσμένη τομή : τομή S για την οποία $f(e) = cap(e), \forall e \in (S, T)$ και $f(e) = 0, \forall e \in (T, S)$.
- Έστω f οποιαδήποτε ροή και έστω S οποιαδήποτε τομή. Τότε, $|f| \leq cap(S)$.

- Υπολειπόμενη χωρητικότητα πλευράς $e = (u, v)$:
 $r(e) = \text{cap}(e) - f(e) + f(e')$, όπου $e' = (v, u)$.
- Υπολειπόμενο δίκτυο $G_f = (V, E_f)$, όπου
 $E_f = \{e \in E : r(e) > 0\}$.
- Έστω f μια ροή και έστω S το σύνολο των κορυφών που είναι προσπελάσιμες από την s .
 - Αν $t \in S$, τότε η f δεν είναι μέγιστη.
 - Αν $t \notin S$, τότε η f είναι μέγιστη και το S αποτελεί μια κορεσμένη τομή.



Ελεγκτής Ορθότητας

- `bool CHECK_MAX_FLOW_T(const graph& G, node s, node t, const edge_array<NT>& cap, const edge_array<NT>& f)`

ελέγχει αν η f είναι μια (s, t) -ροή και επιστρέφει `true` αν όντως είναι, αλλιώς `false`.

- Ιδέα ελεγκτή:
 - ▷ Έλεγχος της συνθήκης χωρητικότητας κάθε πλευράς.
 - ▷ Υπολογισμός πλεονάσματος (*excess*) κάθε κορυφής: όλες οι κορυφές, εκτός από τις s και t , πρέπει να έχουν πλεόνασμα μηδέν.
 - ▷ Υπολογισμός των προσπελάσιμων από την s κορυφών με χρήση ΑΠΠ. Η t δεν πρέπει να είναι προσπελάσιμη.

Ελεγκτής ορθότητας

```
template <class NT>
bool CHECK_MAX_FLOW_T(const graph& G, node s, node t,
                      const edge_array<NT>& cap,
                      const edge_array<NT>& f)
{
    node v; edge e;
    string loc = "CHECK_MAX_FLOW_T: ";
    bool res = true;

    forall_edges(e, G)
        res = res && leda_assert(f[e] >= 0 && f[e] <= cap[e],
                                loc + "illegal flow value");

    node_array<NT> excess(G, 0);
    forall_edges(e, G)
    { node v = G.source(e); node w = G.target(e);
      excess[v] -= f[e]; excess[w] += f[e];
    }
    forall_nodes(v, G)
        res = res && leda_assert(v == s || v == t ||
                                excess[v] == 0, loc + "node with non-zero excess");

    // Compute nodes reachable from s using BFS
}
```

Ελεγκτής ορθότητας - συνέχεια

```
// Compute nodes reachable from s using BFS

node_array<bool> reached(G, false);
queue<node> Q;

Q.append(s); reached[s] = true;
while ( !Q.empty() )
{ node v = Q.pop();
  forall_out_edges(e, v)
  { node w = G.target(e);
    if ( f[e] < cap[e] && !reached[w] )
      { reached[w] = true; Q.append(w); }
  }
  forall_in_edges(e, v)
  { node w = G.source(e);
    if ( f[e] > 0 && !reached[w] )
      { reached[w] = true; Q.append(w); }
  }
}
res = res && leda_assert(!reached[t],
                        "t is reachable in G_f");
return res;
```

Αλγόριθμος Προροής-Ώθησης και Υλοποίηση

- Μέθοδος προροής-ώθησης (*preflow-push*).

- *Προροή*: συνάρτηση $f : E \rightarrow \mathbb{R}_{\geq 0}$ για την οποία:

$$(1) \quad 0 \leq f(e) \leq \text{cap}(e) \quad \forall e \in E$$

$$(2) \quad \text{excess}(v) \geq 0 \quad \forall v \in V^+ = V - \{s, t\}$$

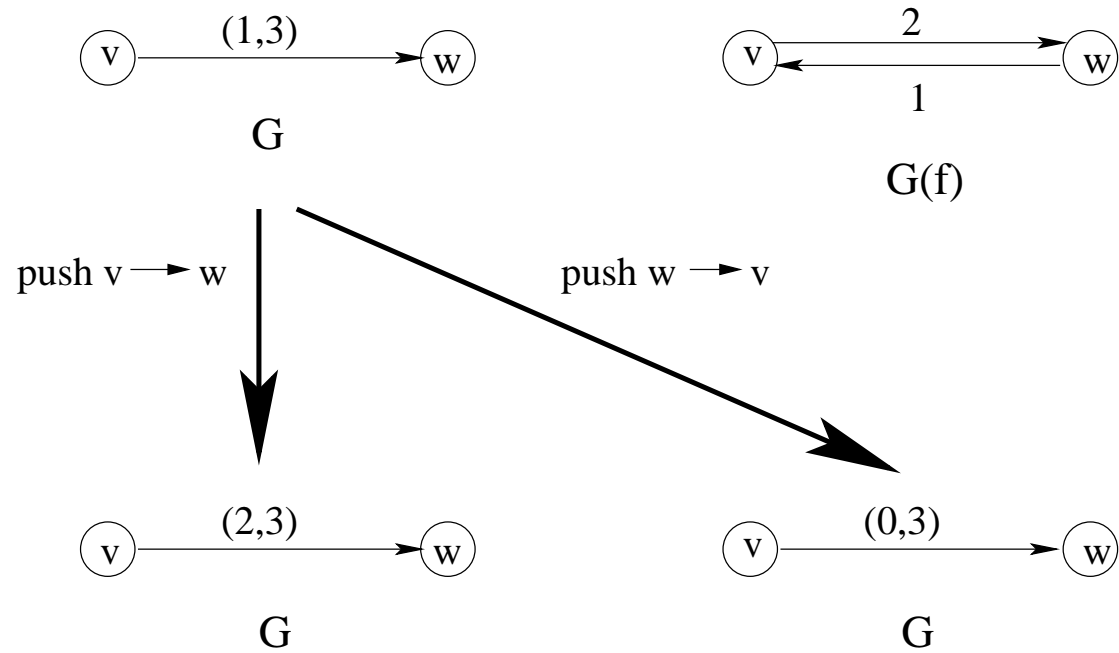
- *Ενεργή κορυφή*: $v \in V^+$ και $\text{excess}(v) > 0$.

- Λειτουργία ώθησης: έστω v ενεργή κορυφή, $e = (v, w)$, και $\delta \leq \min\{\text{excess}(v), r(e)\}$.

Ώθηση δ μονάδων ροής κατά μήκος της $e \implies$

$$\begin{cases} \text{αυξάνει την } f(e) \text{ κατά } \delta, & \text{αν } e \in E \\ \text{μειώνει την } f(e') \text{ κατά } \delta, & \text{αν } e' \in E \end{cases}$$

- Αν $\delta = r(e)$, τότε η ώθηση καλείται *κορεσμένη*.



Ώθηση 1 μονάδας ροής κατά μήκος της (v, w) (ή της (w, v))

- *Ποιες ωθήσεις πρέπει να εκτελεστούν;*

- Ποιες ωθήσεις πρέπει να εκτελεστούν;

Τοποθέτηση των κορυφών σε επίπεδα αρχίζοντας από την t και εκτέλεση ωθήσεων που μεταφέρουν πλεόνασμα από μεγαλύτερο σε μικρότερο επίπεδο.

- Έστω $d(v)$ το επίπεδο της κορυφής v . Μια πλευρά $e = (v, w) \in G_f$ καλείται *κατάλληλη* αν $d(w) < d(v)$.
- Ώθηση κατά μήκος της e εκτελείται αν η v είναι ενεργή και η e κατάλληλη.

- *Τι γίνεται όμως αν κάποια κορυφή v είναι ενεργή και δεν υπάρχει κατάλληλη πλευρά προσκείμενη στην v ;*

- Τι γίνεται όμως αν κάποια κορυφή v είναι ενεργή και δεν υπάρχει κατάλληλη πλευρά προσκείμενη στην v ;

Η v ανεβαίνει επίπεδο, δηλ. το $d(v)$ αυξάνεται κατά 1.

Γενικός Αλγόριθμος Προροής-Ώθησης

/* αρχικοποίηση */

Θέσε $f(e) = cap(e)$ για όλες τις πλευρές με $source(e) = s$;

Θέσε $f(e) = 0$ για όλες τις υπόλοιπες πλευρές ;

Θέσε $d(s) = n$ και $d(v) = 0$ για όλες τις κορυφές ;

/* κύριος βρόχος */

while \exists ενεργή κορυφή

{ έστω v μια ενεργή κορυφή;

if \exists κατάλληλη πλευρά $e = (v, w)$ in G_f **then**

{ ώθησε δ μονάδες ροής στην e ,

όπου $\delta \leq \min\{excess(v), r(e)\};$ }

else

{ αύξησε το $d(v)$; }

}

- Μη κορεσμένες ωθήσεις μπορεί να δημιουργήσουν πρόβλημα.



Υπόθεση 1: Για κάθε ώθηση, $\delta = \min\{excess(v), r(e)\}$.

Αυτό εξασφαλίζει ότι κάθε μη-κορεσμένη ώθηση μιας πλευράς (v, w) αφήνει την v ανενεργή.

Υπόθεση 2: Όταν επιλέγεται μια ενεργή κορυφή v , εκτελούνται ωθήσεις από την v είτε μέχρις ότου η v γίνει ανενεργή, είτε μέχρις ότου δεν υπάρχουν προσκείμενες κατάλληλες πλευρές. Στην τελευταία περίπτωση αυξάνεται το $d(v)$.

- Πως επιλέγονται οι ενεργές κορυφές;

Επιλογή Ενεργών Κορυφών

- *Αυθαίρετη* \Rightarrow αριθμός μη-κορεσμένων ροών $O(n^2m)$.
- *FIFO* \Rightarrow αριθμός μη-κορεσμένων ροών $O(n^3)$.
- *Υψηλότερου Επιπέδου* \Rightarrow αριθμός μη-κορεσμένων ροών $O(n^2\sqrt{m})$.

Υλοποίηση

- Υπάρχουν δύο παράμετροι αρχετύπων: ο αριθμητικός τύπος NT και η υλοποίηση του συνόλου των ενεργών κορυφών U.

- `// max_flow_basic`

```
template<class NT, class SET>
NT MAX_FLOW_BASIC_T(const graph& G, node s, node t,
                    const edge_array<NT>& cap,
                    edge_array<NT>& flow,
                    SET& U, int& num_pushes,
                    int& num_edge_inspection,
                    int& num_relabels)
{
    if (s == t) error_handler(1, "MAXFLOW: source == sink");

    // MF_BASIC: initialization

    // MF_BASIC: main loop

    #ifndef LEDA_CHECKING_OFF
        assert(CHECK_MAX_FLOW_T(G, s, t, cap, flow));
    #endif

    return excess[t];
}
```

Αρχικοποίηση και Δομές Δεδομένων

```
// MF_BASIC: initialization

// initialize flow and excess, and saturate edges out of s

flow.init(G, 0);
if (G.outdeg(s) == 0) return 0;

int n = G.number_of_nodes();
int max_level = 2*n - 1;
int m = G.number_of_edges();

node_array<NT> excess(G, 0);

// saturate all edges leaving s
edge e;
forall_out_edges(e, s)
{ NT c = cap[e];
  if (c == 0) continue;
  node v = target(e);
  flow[e] = c;
  excess[s] -= c;
  excess[v] += c;
}
```

Αρχικοποίηση και Δομές Δεδομένων - συνέχεια

```
// MF_BASIC: initialize dist and U

node_array<int> dist(G,0); dist[s] = n;
node v;
forall_nodes(v,G)
  if ( excess[v] > 0 ) U.insert(v,dist[v]);

// MF_BASIC: initialize counters

num_relabels = num_pushes = num_edge_inspectoins = 0;
```

Υλοποίηση του Συνόλου U των Ενεργών Κορυφών

Λειτουργίες:

Συναρτήσεις κατασκευής και κατάργησης.

`node U.del()` : διαγραφή κορυφής από το U και επιστροφή της (επιστρέφει `nil` αν το U είναι κενό).

`U.insert(node v, int d)` : εισάγει την v με επίπεδο d . Αυτή η εκδοχή χρησιμοποιείται στην φάση της αρχικοποίησης και όταν μια κορυφή επανεισάγεται στο U μετά από αύξηση του επιπέδου της.

`U.insert0(node v, int d)` : εισάγει την v με επίπεδο d . Αυτή η εκδοχή χρησιμοποιείται όταν μια κορυφή γίνεται ενεργή από μια ώθηση προς αυτή την κορυφή.

`bool U.empty()` : επιστρέφει `true` όταν το U είναι κενό.

`U.clear()` : διαγράφει όλα τα στοιχεία του U .

Η υλοποίηση FIFO: αποθηκεύει το U σαν μια ουρά.

```
// FIFO implementation of SET

#include <LEDA/list.h>

class fifo_set{

    list<node> L;

public:

    fifo_set(){}
    node del() { if (!L.empty()) return L.pop();
                else return nil; }

    void insert(node v, int d) { L.append(v); }
    void insert0(node v, int d) { L.append(v); }

    bool empty() { return L.empty(); }
    void clear() { L.clear(); }
    ~fifo_set(){}
};
```

Η υλοποίηση MFIFO (τροποποιημένη FIFO): αποθηκεύει τις κορυφές του U σε μια λίστα. Κορυφές που επανεισάγονται μετά από αύξηση επιπέδου τοποθετούνται στην αρχή της ουράς, ενώ οι κορυφές που γίνονται ενεργές λόγω μιας ώθησης τοποθετούνται στο τέλος της ουράς.

```
// MFIFO implementation of SET

#include <LEDA/list.h>

class mfifo_set{

    list<node> L;

public:

    mfifo_set(){}
    node del() { if ( !L.empty() ) return L.pop();
                else return nil; }

    void insert(node v, int d) { L.push(v); }
    void insert0(node v, int d){ L.append(v); }

    bool empty() { return L.empty(); }
    void clear() { L.clear(); }
    ~mfifo_set(){}
};
```

Η Υλοποίηση Υψηλότερου Επιπέδου

- Διατηρεί ένα διάνυσμα A γραμμικών λιστών με αριθμοδείκτες στο διάστημα $[0..max_level]$, όπου max_level είναι μια παράμετρος της συνάρτησης κατασκευής.
- Η λίστα $A[d]$ περιέχει όλες τις κορυφές v που εισήχθησαν από $insert(v, d)$ ή από $insert0(v, d)$.
- Μια μεταβλητή max διατηρείται έτσι ώστε η $A[d]$ να είναι κενή αν $d > max$.
- Στην $insert0$ λαμβάνεται υπόψη το γεγονός ότι η εισαγωγή κορυφών αφορά επίπεδα μικρότερα του max_level .

```

// Highest level implementation of SET

#include <LEDA/list.h>
#include <LEDA/array.h>

class hl_set{

    int max, max_lev;
    array<list<node> > A;

public:

    hl_set(int max_level):A(max_level+1)
    { max = -1; max_lev = max_level;}

    node del()
    { while (max >= 0 && A[max].empty()) max--;
      if (max >= 0) return A[max].pop(); else return nil;
    }

    void insert(node v, int d)
    { A[d].push(v);
      if (d > max) max = d;
    }
}

```

```
void insert0(node v, int d) { A[d].append(v); }

bool empty()
{ while (max >= 0 && A[max].empty()) max--;
  return ( max < 0 );
}

~hl_set(){}

void clear()
{ for (int i = 0; i <= max_lev; i++) A[i].clear();
  max = -1;
}
};
```

Κύριος Βρόχος

- Επιλέγεται μια κορυφή v από το U . Αν δεν υπάρχει, η εκτέλεση του βρόχου σταματάει, ή αν η v είναι ταυτόσημη με την t , τότε προχωράμε στην επόμενη επανάληψη.
- Προσπαθούμε να ωθήσουμε το πλεόνασμα της v στις γειτονικές κορυφές της στο G_f . Αυτό γίνεται εξετάζοντας πρώτα τις εξερχόμενες και μετά τις εισερχόμενες πλευρές της v .
- Αν η v παραμένει ενεργή μετά τον κορεσμό όλων των γειτονικών πλευρών της, τότε της αυξάνουμε το επίπεδο και την εισάγουμε ξανά στο U .

```

// MF_BASIC: main loop

for(;;)
{
    node v = U.del();

    if (v == nil) break;
    if (v == t) continue;

    NT ev = excess[v]; // excess of v
    int dv = dist[v]; // level of v
    edge e;

    // MF_BASIC: push across edges out of v

    if ( ev > 0 )
    { // MF_BASIC: push across edges into v }

    excess[v] = ev;

    if (ev > 0)
    { dist[v]++;
      num_relabels++;
      U.insert(v, dist[v]);
    }
}

```

Ώθηση πλεονάσματος μέσω εξερχόμενων πλευρών

- Η ώθηση του πλεονάσματος γίνεται μέσω των κατάλληλων πλευρών. Μια πλευρά $e \in G_f$ είτε είναι και πλευρά του G (οπότε $flow[e] < cap[e]$), είτε είναι η αντίθετη μιας πλευράς του G (οπότε $flow[e'] > 0$).
- Για κάθε εξερχόμενη πλευρά e μιας κορυφής v , ωθούμε $\min\{excess[v], cap[e] - flow[e]\}$. Αν το πλεόνασμα της v γίνει μηδέν, τότε ο βρόχος `for` σταματάει.

```

// MF_BASIC: push across edges out of v

for (e = G.first_adj_edge(v); e; e = G.adj_succ(e))
{ num_edge_inspection++;
  NT& fe = flow[e];
  NT rc = cap[e] - fe;
  if (rc == 0) continue;
  node w = target(e);
  int dw = dist[w];
  if ( dw < dv ) // equivalent to ( dw == dv - 1 )
  { num_pushes++;
    NT& ew = excess[w];
    if (ew == 0) U.insert0(w, dw);
    if (ev <= rc)
    { ew += ev; fe += ev;
      ev = 0; // stop: excess[v] exhausted
      break;
    }
    else
    { ew += rc; fe += rc;
      ev -= rc;
    }
  }
}
}

```

```

// MF_BASIC: push across edges into v

for (e = G.first_in_edge(v); e; e = G.in_succ(e))
{ num_edge_inspection++;
  NT& fe = flow[e];
  if (fe == 0) continue;
  node w = source(e);
  int dw = dist[w];
  if ( dw < dv ) // equivalent to ( dw == dv - 1 )
  { num_pushes++;
    NT& ew = excess[w];
    if (ew == 0) U.insert0(w, dw);
    if (ev <= fe)
    { fe -= ev; ew += ev;
      ev = 0; // stop: excess[v] exhausted
      break;
    }
    else
    { ew += fe; ev -= fe;
      fe = 0;
    }
  }
}
}

```

Αριθμητικοί Τύποι και Ορθότητα Αλγορίθμων

Γραφημάτων & Δικτύων

- Οι υλοποιήσεις με αλγορίθμους γραφημάτων και δικτύων δουλεύουν σωστά με αριθμητικούς τύπους οι οποίοι δεν εισάγουν αριθμητικά σφάλματα.
- Π.χ. αριθμητική με τύπους `int` μπορεί να προκαλέσει υπερχείλιση, ή περίεργα αποτελέσματα λόγω αναδίπλωσης, ενώ αριθμητική με τύπους `double` μπορεί να προκαλέσει σφάλματα στρογγυλοποίησης.

Πώς εξασφαλίζουμε ορθότητα ;

1. Αναλύουμε τις αριθμητικές απαιτήσεις του αλγορίθμου.

Στόχος: εύρεση μιας τιμής f , για την οποία ισχύει το εξής.

Αν η μέγιστη απόλυτη τιμή εισόδου φράσσεται από την C , τότε όλοι οι αριθμοί που χρησιμοποιούνται από τον αλγόριθμο φράσσονται από την ποσότητα $f \cdot C$. Στην περίπτωση αυτή λέμε ότι ο αλγόριθμος φράσσεται από την f .

Π.χ. για αλγόριθμους συντομότερων διαδρομών $f = n$, ενώ για αλγόριθμους ροών $f = \text{out-degree}(s)$.

Πώς εξασφαλίζουμε ορθότητα ; (συνέχεια)

2. (τύπος `int`) Ελέγχουμε αν όλες οι τιμές w της εισόδου ικανοποιούν $f \cdot w \leq \text{MAXINT}$.

2. (τύπος `double`) Μετασχηματίζουμε τις αριθμητικές τιμές εισόδου

$$w \longrightarrow \text{sign}(w) \cdot \lfloor |w| \cdot S \rfloor / S$$

όπου $S = 2^s$ είναι η παράμετρος κλιμάκωσης. Επομένως δεν θα υπάρχουν σφάλματα στρογγυλοποίησης όταν τα ενδιάμεσα αποτελέσματα είναι της μορφής $z \cdot 2^{-s}$, όπου z είναι ακέραιος.

Πως επιλέγεται η τιμή του s ;

- Έστω C η μεγαλύτερη απόλυτη τιμή εισόδου.

Το βήμα 1 πρέπει να ισχύει και για τις μετασχηματισμένες ακέραιες τιμές $sign(w) \cdot \lfloor |w| \cdot S \rfloor$. Επομένως για αλγόριθμο που φράσσεται από το f πρέπει να έχουμε

$$f \cdot \lfloor C \cdot S \rfloor < 2^{53}$$

αφού η αριθμητική κινητής υποδιαστολής για τύπο `double` μπορεί να αναπαραστήσει όλους τους ακραίους στο διάστημα $[-(2^{53} - 1)..2^{53} - 1]$.

- Η παραπάνω ανισότητα ικανοποιείται αν

$$f \cdot C \cdot S < 2^{53} \iff s < 53 - \log(f \cdot C)$$

Ποια είναι η σχέση μεταξύ του αποτελέσματος με μετασχηματισμένες τιμές και εκείνου με τις αυθεντικές τιμές ;

- Δεν υπάρχει γενικός κανόνας. Συνήθως το αποτέλεσμα με μετασχηματισμένες τιμές είναι μια καλή προσέγγιση του πραγματικού αποτελέσματος.
- Έστω ότι το αποτέλεσμα είναι το άθροισμα L τιμών. Η πραγματική τιμή w με την μετασχηματισμένη w' διαφέρουν κατά

$$w - w' = w - \lfloor w \cdot S \rfloor / S = (w \cdot S - \lfloor w \cdot S \rfloor) / S \leq 1/S$$

Επομένως, το άθροισμα L μετασχηματισμένων τιμών διαφέρει από το άθροισμα L πραγματικών τιμών το πολύ κατά L/S .

Αν επιλέξουμε το S να είναι η μέγιστη δύναμη του 2 για την οποία $S < 2^{53}/(f \cdot C)$, τότε $S \geq 2^{52}/(f \cdot C)$ και κατά συνέπεια

$$\frac{L}{S} \leq \frac{L \cdot f \cdot C}{2^{52}}$$

που είναι το μέγιστο απόλυτο λάθος.