

Τεχνολογίες Υλοποίησης Αλγορίθμων

Βιβλιοθήκες Λογισμικού: LEDA

Χρήστος Δ. Ζαρολιάγκης

Καθηγητής

email: zaro@ceid.upatras.gr

Τμήμα Μηχ/κών Υπολογιστών & Πληροφορικής

Πανεπιστήμιο Πατρών

- Βιβλιοθήκες Λογισμικού Αλγορίθμων και Δομών Δεδομένων: **Το παράδειγμα της LEDA**

- Η Έννοια της Αντιγραφής
- Εντολές Επανάληψης
- Παραμετρικοί Τύποι
- Άλλοι Σημαντικοί Τύποι
- Παράμετροι Υλοποίησης
- Έλεγχος Ορθότητας Προγράμματος
- Διαχείριση Μνήμης

Κεντρική έννοια στην C++, άρα και στη LEDA (καταχώρηση, δημιουργία με αρχικοποίηση, μεταβίβαση παραμέτρων κατ' αξία, επιστροφή τιμής συνάρτησης).

- **Πρωτογενείς τύποι (primitive types):** ενσωματωμένοι τύποι, τύποι δεικτών και τύποι στοιχείων.

- **Μη-πρωτογενείς τύποι (non-primitive types)**

LEDA: Η καταχώρηση ορίζεται σε σχέση με την αντιγραφή και έχει ενιαία σημασιολογία (semantics) για όλους τους τύπους της LEDA.

LEDA - Κανόνας 6: Η καταχώρηση $x = A$ καταχωρεί ένα αντίγραφο της τιμής της παράστασης A στην μεταβλητή x .

C++ - Αξίωμα 1: Ο ορισμός $T \ x = A$ δημιουργεί μια νέα μεταβλητή x τύπου T και την αρχικοποιεί με ένα αντίγραφο της τιμής της A . Μία εναλλακτική συντακτική μορφή είναι η $T \ x(A)$. Η εντολή `new T(A)` επιστρέφει έναν δείκτη σε ένα νέο ανώνυμο αντικείμενο τύπου T . Το αντικείμενο αρχικοποιείται με ένα αντίγραφο της τιμής της A .

C++ - Αξίωμα 2:

- (α) Μια τυπική παράμετρος τιμής τύπου T και ονόματος x ορίζεται ως $T \ x$. Έστω A μια πραγματική παράμετρος (παράσταση) τύπου T . Μεταβίβαση παραμέτρων κατ' αξία είναι ισοδύναμη με τον ορισμό $T \ x = A$.
- (β) Έστω f μια συνάρτηση με τύπο επιστροφής T και έστω `return A` μια εντολή επιστροφής στο σώμα της f (η A είναι τύπου T). Η επιστροφή τιμής συνάρτησης είναι ισοδύναμη με τον ορισμό $T \ x = A$, όπου x είναι μια προσωρινή μεταβλητή τύπου T που δημιουργείται από τον μεταγλωππιστή.

Πώς ακριβώς ορίζεται η αντιγραφή;

LEDA - Κανόνας 7:

- (α) Όλοι οι ενσωματωμένοι τύποι, τύποι δεικτών και τύποι στοιχείων είναι πρωτογενείς.
- (β) Ένα αντίγραφο της τιμής ενός πρωτογενούς τύπου είναι η ίδια η τιμή του.



- ▷ Δεν μπορούμε να ξεχωρίσουμε μεταξύ των δύο αντιγράφων.
- ▷ Ο έλεγχος ισότητας είναι πάντα αληθής.

Η έννοια της αντιγραφής μη πρωτογενών τύπων είναι πιο πολύπλοκη (ευτυχώς χρειάζεται σπάνια) ...

Συμβουλή: Καλό είναι να αποφεύγεται η καταχώρηση, αρχικοποίηση με αντιγραφή, μεταβίβαση παραμέτρων κατ' αξία και επιστροφή τιμής συνάρτησης για μη πρωτογενείς τύπους. Χρειάζεται προσοχή όταν χρησιμοποιείτε μη πρωτογενείς τύπους σαν πραγματικές παραμέτρους.

Αντιγραφή μη Πρωτογενών Τύπων

```
L1 = L2; // L1 and L2 are lists
int f(list<int> A); // non-primitive value parameter
list<int> f(); // non-primitive value return

dictionary<string, list<int> > D;
// non-primitive type parameter
```

- Η τιμή ενός μη πρωτογενούς τύπου έχει «δομή»
⇒ *Μη Πρωτογενείς Τύποι ≡ Δομημένοι Τύποι (ΔΤ)*
- Ένα αντίγραφο της τιμής ενός δομημένου τύπου είναι *παρόμοιο αλλά όχι ταυτόσημο* με την πρωτότυπη τιμή.
 - ▷ ΔΤ βασισμένος σε στοιχεία (item-based): οι τιμές του ορίζονται σαν συλλογή στοιχείων (π.χ. dictionary, list, graph).
 - ▷ Απλός ΔΤ (π.χ. array, stack, set).

LEDA - Κανόνας 8:

- (α) Μια τιμή x ενός απλού ΔΤ είναι ένα σύνολο ή ακολουθία στοιχείων ή μεταβλητών κάποιου τύπου E . Ένα αντίγραφο της x είναι ένα αντίγραφο κατά συνιστώσες (component-wise copy).
- (β) Ένα αντίγραφο μιας μεταβλητής είναι μια νέα μεταβλητή του ίδιου τύπου που αρχικοποιείται με ένα αντίγραφο της τιμής της πρωτότυπης μεταβλητής.

```
array<int> A(0,2);  
array<int> B = A;  
int * p = A;  
int * q = B;  
p == q;           // evaluates to false
```

LEDA - Κανόνας 9: Μια τιμή ενός ΔΤ βασισμένου σε στοιχεία είναι μια δομημένη συλλογή στοιχείων καθένα από τα οποία έχει ≥ 0 κατηγορήματα. Ένα αντίγραφο μιας τέτοιας τιμής είναι μια συλλογή από νέα στοιχεία, ένα για κάθε στοιχείο της πρωτότυπης τιμής. Η συνδυαστική δομή που τίθεται στα νέα στοιχεία είναι ισομορφική με τη δομή της πρωτότυπης τιμής. Κάθε κατηγορημα ενός νέου στοιχείου που δεν έχει συνδυαστική δομή λαμβάνει ως τιμή ένα αντίγραφο του αντίστοιχου κατηγορήματος του στοιχείου της πρωτότυπης τιμής.

- Αντιγραφή μιας λίστας τύπου `list<E>` με 5 στοιχεία σημαίνει τη δημιουργία 5 νέων στοιχείων τύπου `list_item`, τη διάταξή τους σε μορφή λίστας, και την ανάθεση του περιεχομένου του i -οστού στοιχείου, $1 \leq i \leq 5$, με ένα αντίγραφο του περιεχομένου του i -οστού στοιχείου στην πρωτότυπη λίστα.
- Αντιγραφή ενός γραφήματος τύπου `graph` με n κόμβους και m πλευρές σημαίνει τη δημιουργία n νέων κόμβων και m νέων πλευρών και τη δημιουργία μιας ισομορφικής με το πρωτότυπο γράφημα δομής σε αυτά.

Μεταβίβαση παραμέτρων σε ΔΤ - 1

```
dictionary<string, dictionary<int, int> > M;  
dictionary<int, int> D;  
dic_item it = D.insert(1,1);
```

```
M.insert("Leda", D);  
M.access("Leda").inf(it);    // illegal
```

```
D.change_inf(it, 2);  
M.inf("Leda").access(1);    // returns 1
```

```
D.insert(2, 2);  
M.access("Leda").lookup(2); // returns nil
```

Η κλήση `M.insert("Leda", D)` έχει ως αποτέλεσμα:

⇒ ένα αντίγραφο του `D` αποθηκεύεται στο `M`.

⇒ οτιδήποτε ενέργεια κάνουμε στο `D` δεν επηρεάζει καθόλου το αντίγραφο του στο `M`.

Μεταβίβαση παραμέτρων σε ΔΤ - 2

Συμβουλή: Χρησιμοποιείτε μεταβίβαση παραμέτρων κατ' αναφορά (const ή όχι) για μη πρωτογενείς τύπους και χρησιμοποιείτε μεταβίβαση κατ' αξία μόνο για πρωτογενείς τύπους.

```
void DIJKSTRA(const graph& G, node s,  
              const edge_array<int>& cost,  
              node_array<int>& dist,  
              node_array<edge>& pred)
```

Τι θα συνέβαινε αν η μεταβίβαση του G γινόταν κατ' αξία;

Κατάργηση Αντικειμένων στη LEDA

LEDA - Κανόνας 10: *Όταν ένα αντικείμενο της LEDA καταργείται ο χώρος που είχε κατανεμηθεί για το αντικείμενο απελευθερώνεται. Είναι ακριβώς ο χώρος που θα αντιγράφετο αν δημιουργούσαμε ένα αντίγραφο του αντικειμένου.*

Εντολές Επανάληψης

- Η LEDA προσφέρει εντολές επανάληψης για τους περισσότερους τύπους δεδομένων.
- Έστω D ένας οποιοσδήποτε ΔT βασισμένος σε στοιχεία:

```
forall_items(it, D)
{
    /* the items in D are
       successively assigned to it */
}
```

Η επανάληψη αναθέτει διαδοχικά τα στοιχεία του D στο it και εκτελεί το σώμα του βρόχου για κάθε ένα από αυτά.

- Για λίστες και σύνολα υπάρχουν εντολές επανάληψης και για τα συνολοστοιχεία. Π.χ.

```
list<point> L;
point p;
forall(p, L)
{
    /* the elements of L are
       successively assigned to p */
}
```

- Για γραφήματα υπάρχουν εντολές επανάληψης πάνω σε κόμβους, πλευρές, γειτονικούς κόμβους, κλπ. Π.χ.

```
forall_nodes(v, G)
{
    /* the nodes of G are successively assigned to v*/
}
```

```
forall_edges(e, G)
{
    /* the edges of G are
    successively assigned to e*/
}
```

```
forall_adj_edges(e, v)
{
    /* all edges adjacent to v are
    successively assigned to e */
}
```

- Είναι επικίνδυνο να τροποποιείται μια συλλογή κατά τη διάρκεια μιας εντολής επανάληψης.

LEDA - Κανόνας 11: Μια επανάληψη πάνω στα στοιχεία μιας συλλογής C δεν πρέπει να προσθέτει νέα στοιχεία στην C . Μπορεί να διαγράψει το τρέχον στοιχείο της επανάληψης, αλλά κανένα άλλο. Τα κατηγορήματα των στοιχείων της C μπορούν να αλλάζουν χωρίς περιορισμούς.

```

list<int> L;
list<int>::item it;
// ...

// delete all occurrences of 5
forall_items(it, L)
    if ( L[it] == 5 )
        L.del(it);

forall_items(it, L)
    if ( L[it] == 5 )
        L.del(L.succ(it)); // illegal ??

// add 1 to the elements following a 5
forall_items(it, L)
    if ( L[it] == 5 )
        L[L.succ(it)]++;

forall_items(it, L)
    L.append(1); // infinite loop

graph G;
// ...
// add a new node s and edges (s,v) for all nodes of G

node s = G.new_node();
node v;
forall_nodes(v, G)
    if (v != s)
        G.new_edge(s, v);

```

- Οι εντολές επανάληψης στη LEDA είναι στην πραγματικότητα macros που ανάγονται σε εντολές `for` της C++.

```
forall_items(it, L)
{ <<body>> }
```

⇓ 1η προσπάθεια

```
for (it = (L).first_item();
     it != nil;
     it = (L).next_item(it))
{ <<body>> }
```

- **Μειονέκτημα:** δεν μπορεί να διαγραφεί το τρέχον στοιχείο από την `L`

- 2η προσπάθεια

```
forall_items(it, L)  
{ <<body>> }
```



```
for (list_item loop_it = (L).first_item();  
     it = loop_it, loop_it = (L).next_item(loop_it),  
     it)  
{ <<body>> }
```



```
for (void * loop_it = (L).first_item();  
     it = (T)loop_it,  
     loop_it = (L).next_item(loop_it), it; )  
{ <<body>> }
```

- Αυτό έχει δύο συνέπειες . . .

LEDA - Κανόνας 12: Οι εντολές *break* και *continue* μπορούν να χρησιμοποιηθούν σε όλους τους βρόχους *forall*.

```
list_item it;  
forall_items(it, L)  
    if ( L[it] == 5 )  
        break;
```

```
if ( it ) // there is an occurrence of 5 in L  
    ...  
else     // there is no occurrence of 5 in L  
    ...
```

- Η δεύτερη συνέπεια μπορεί να δημιουργήσει πρόβλημα. Π.χ.:

```
edge e;  
forall(e, G.all_edges())  
{ <<body>> }
```



```
for (list_item loop_it = (G.all_edges()).first_item();  
     it=loop_it, e=(G.all_edges()).inf(it),  
     loop_it=(G.all_edges()).next_item(loop_it), it;)  
{ <<body>> }
```



Η συνάρτηση `G.all_edges()` καλείται σε κάθε επανάληψη του βρόχου.

LEDA - Κανόνας 13: Το όρισμα του τύπου δεδομένων σε μια εντολή επανάληψης δεν πρέπει να είναι μια κλήση συνάρτησης η οποία παράγει ένα αντικείμενο του τύπου, αλλά ένα καθαυτό αντικείμενο του τύπου.

Ο σωστός τρόπος να γράψουμε το προηγούμενο παράδειγμα είναι ο εξής.

```
list<edge> E = G.all_edges();  
edge e;  
forall(e, E)  
    { <<body>> }
```

ή απλούστερα

```
forall_edges(e, G)  
    { <<body>> }
```

Απαριθμητές τύπου STL

- Η έννοια του επαναλήπτη της STL είναι διαφορετική από αυτή της LEDA και ειδικότερα από την έννοια του στοιχείου της LEDA.

- Στην STL ένας βρόχος `forall_items` για μια λίστα `list<int>` έχει ως εξής.

```
for (list<int>::iterator it = L.begin();
     it != L.end();
     it++)
{ <<body>> }
```

- Το περιεχόμενο του επαναλήπτη μέσα στο σώμα του βρόχου προσπελάζεται με `*it`. Στη LEDA θα γράφαμε `L[it]`.

Παραμετρικοί Τύποι

- Οι περισσότεροι τύποι δεδομένων της LEDA είναι παραμετρικοί.
- Οποιαδήποτε κλάση που παρέχει ένα συγκεκριμένο αλλά μικρό σύνολο συναρτήσεων μπορεί να χρησιμοποιηθεί σαν ένα πραγματικό όρισμα τύπου.
- *Ποιές είναι αυτές οι συναρτήσεις;*

LEDA - Κανόνας 14: Κάθε πραγματικό όρισμα τύπου T πρέπει να παρέχει τις εξής συναρτήσεις:

προεπιλεγμένη συνάρτηση κατασκευής	<code>T::T()</code>
συνάρτηση κατασκευής αντιγράφου	<code>T::T(const T&)</code>
τελεστή καταχώρησης	<code>T& T::operator=(const T&)</code>
συνάρτηση κατάργησης	<code>T::~~T()</code>

Ένας γραμμικά διατεταγμένος τύπος πρέπει επιπλέον να παρέχει

συνάρτηση σύγκρισης	<code>int compare(const T&, const T&)</code>
---------------------	--

Ένας τύπος κατακερματισμού (*hashed type*) πρέπει επιπλέον να παρέχει

συνάρτηση κατακερματισμού	<code>int Hash(const T&)</code>
τελεστή ισότητας	<code>bool operator==(const T&, const T&)</code>

Ένας αριθμητικός τύπος πρέπει επιπλέον να παρέχει τις βασικές αριθμητικές συναρτήσεις πρόσθεσης, αφαίρεσης και πολλαπλασιασμού, καθώς και τους καθιερωμένους τελεστές σύγκρισης.

Παράδειγμα: η γραμμικά διατεταγμένη κλάση `pair`

```
class pair
{
    public:
        pair() { x = y = 0; }
        pair(const pair& p) { x = p.x; y = p.y; }
        friend void Read(pair& p, istream& is)
            { is >> p.x >> p.y; }
        friend void Print(const pair& p, ostream& os)
            { os << p.x << " " << p.y; }
        friend int compare(const pair&, const pair&);
    private:
        double x, y;
};

namespace leda {
int compare(const pair& p, const pair& q)
{
    if (p.x < q.x) return - 1;
    if (p.x > q.x) return  1;
    if (p.y < q.y) return - 1;
    if (p.y > q.y) return  1;
    return 0;
}
}
```

Παρατηρήσεις

- (1) Οι συναρτήσεις `Read()`, `Print()` (αν απαιτείται είσοδος/έξοδος για τον τύπο) και `compare()` είναι καθολικές συναρτήσεις και όχι συναρτήσεις μέλη. Γι αυτό ορίζονται ως φίλες.

 - (2) Οι συναρτήσεις `compare()` και `Hash()` πρέπει να οριστούν μέσα στο namespace `leda`.

 - (3) Ο τελεστής καταχώρησης και η συνάρτηση κατάργησης δεν ορίσθηκαν \Rightarrow οι προεπιλεγμένες μέθοδοι της C++ θα δημιουργηθούν.
- *Τι συμβαίνει αν δεν έχουν ορισθεί για ένα πραγματικό όρισμα τύπου T οι συναρτήσεις για τις οποίες δεν υπάρχουν προεπιλεγμένες μέθοδοι;*

Ο μεταγλωπστής της C++ θα παράγει ένα μήνυμα λάθους ότι δεν μπορεί να ταιριάξει κάποιες συναρτήσεις. Π.χ. για το πρόγραμμα

```
#include <LEDA/core/dictionary.h>
using namespace leda;

class pair
{
public:
    pair() { x = y = 0; }
    pair(const pair& p) { x = p.x; y = p.y; }
private:
    double x, y;
};

int main()
{
    dictionary<pair,int> D;
    return 0;
}
```

θα εμφανίσει μήνυμα ότι δεν μπορεί να βρεί ταίριασμα για τη συνάρτηση `compare()`.

Άλλοι σημαντικοί τύποι

- Γραμμικά Διατεταγμένοι (linearly ordered)
- Ισότητας (equality)
- Κατακερματισμού (hashed)

Γραμμικά Διατεταγμένοι Τύποι

- Πολύ συχνά χρειάζεται να συγκρίνουμε αντικείμενα. Π.χ. σε λεξικά, ουρές προτεραιότητας, αλγορίθμους ταξινόμησης, γεωμετρικούς αλγορίθμους (πότε μια γραμμή είναι πάνω από μια άλλη), κλπ.



- **Γραμμική Διάταξη:** μια δυαδική σχέση \leq σε ένα σύνολο S για την οποία οι εξής τρεις ιδιότητες ισχύουν, $\forall x, y, z \in S$.

(1) $x \leq x$ (ανακλαστική)

(2) $x \leq y$ ή $y \leq x$ (αντισυμμετρική)

(3) $x \leq y$ και $y \leq z \Rightarrow x \leq z$ (μεταβατική)

- Τα x και y λέγονται **ισοδύναμα** αν $x \leq y$ και $y \leq x$. Το x **είναι μικρότερο** από το y , $x < y$ ή $y > x$, αν $x \leq y$ και τα x και y δεν είναι ισοδύναμα.

- **LEDA:** μια συνάρτηση `int cmp(const T&, const T&)` υλοποιεί μια γραμμική διάταξη σε έναν τύπο `T`, αν υπάρχει μια γραμμική διάταξη \leq στον `T` έτσι ώστε $\forall x, y \in T$

$$\text{cmp}(x, y) \begin{cases} < 0, & \text{αν } x < y \\ = 0, & \text{αν } x \text{ είναι ισοδύναμο με } y \\ > 0, & \text{αν } x > y \end{cases}$$

LEDA - Κανόνας 15: Ένας τύπος `T` καλείται γραμμικά διατεταγμένος αν η συνάρτηση `int compare(const T&, const T&)` έχει ορισθεί για τον τύπο `T` και υλοποιεί μια γραμμική διάταξη σε αυτόν τον τύπο. Αν η `compare(x, y)` επιστρέφει μηδέν για δύο αντικείμενα `x` και `y`, τότε τα αντικείμενα αυτά καλούνται `compare`-ισοδύναμα ή απλά ισοδύναμα.

Τύποι Ισότητας

- *Σχέση Ισοδυναμίας*: μια δυαδική σχέση R σε ένα σύνολο S για την οποία οι εξής τρεις ιδιότητες ισχύουν, $\forall x, y, z \in S$.

(1) xRx (ανακλαστική)

(2) $xRy \Rightarrow yRx$ (συμμετρική)

(3) xRy και $yRz \Rightarrow xRz$ (μεταβατική)

- Π.χ. σχέσης ισοδυναμίας: xRy αν $\text{compare}(x, y) == 0$.

LEDA - Κανόνας 16: Αν ο τελεστής ισότητας

`bool operator==(const T&, const T&)`

έχει ορισθεί για τον τύπο T , τότε ορίζει μια σχέση ισοδυναμίας στον T . Τα x και y καλούνται ίσα αν η παράσταση $x==y$ είναι αληθής.

- Ισότητα \neq compare-ισοδυναμία.

Π.χ. ένας γεωμετρικός αλγόριθμος μπορεί να συγκρίνει, μέσω μιας κάθετης ευθείας, ευθύγραμμα τμήματα με βάση την y -συντεταγμένη της τομής τους \Rightarrow δύο ευθύγραμμα τμήματα μπορεί να είναι compare-ισοδύναμα χωρίς να είναι ίσα.

- Όμως ... σε όλους τους τύπους της LEDA στους οποίους η `compare()` και ο τελεστής `==` έχουν ορισθεί, οι έννοιες της ισότητας και compare-ισοδυναμίας συμπίπτουν.

Τύποι Κατακερματισμού

LEDA - Κανόνας 17: Για κάθε τύπο κατακερματισμού T και για οποιαδήποτε αντικείμενα x και y του T :
αν $x==y$, τότε $Hash(x) == Hash(y)$.

- Είναι δυνατόν να γραφούν συναρτήσεις `compare()` και τελεστές ισότητας που να διακρίνουν μεταξύ μιας τιμής και ενός αντιγράφου της τιμής \Rightarrow **πιθανά καταστροφικά αποτελέσματα (!)**, π.χ. μια διερεύνηση σε ένα λεξικό θα αποτύγχανε να βρει ένα αποθηκευμένο κλειδί.



LEDA - Κανόνας 18: *Μια τιμή και το αντίγραφό της πρέπει να είναι compare-ισοδύναμες και ίσες.*

- Σε μερικές περιπτώσεις είναι χρήσιμο να έχουμε περισσότερες από μία γραμμικές διατάξεις για έναν τύπο T.
- Π.χ. θέλουμε να έχουμε δύο λεξικά D1 και D2 με τύπο κλειδιού `pair`, όπου στο D1 τα ζεύγη είναι διατεταγμένα σύμφωνα με τις Καρτεσιανές τους συντεταγμένες, ενώ στο D2 σύμφωνα με τις πολικές τους συντεταγμένες.

- Για το D1 έχουμε:

```
dictionary<pair, int> D1;
```

- *Πως ορίζεται όμως το D2;*

Λύση 1η

- Ορισμός ενός ισοδύναμου τύπου με την εναλλακτική διάταξη.

```
int pol_cmp(const point& x, const point& y)
{
    /* compute lexicographic ordering
       by polar coordinates */
}
```

```
DEFINE_LINEAR_ORDER(point, pol_cmp, pol_point);
// defines a type pol_point by a call to the macro;
// point and pol_point are equivalent, but the
// ordering of pol_point is given by pol_cmp
```

```
dictionary<pol_point, int> D2;
```

Λύση 2η

- Η γραμμική διάταξη γίνεται ένα (επιπλέον) όρισμα οποιουδήποτε τύπου δεδομένων που απαιτεί έναν γραμμικά διατεταγμένο τύπο. Π.χ.

```
dictionary<point, int> D2 (pol_cmp) ;
```

- Επίσης, αντί για μεταβίβαση μιας συνάρτησης σε ένα λεξικό, μπορούμε να μεταβιβάσουμε μια κλάση η οποία έχει έναν τελεστή συνάρτησης και η οποία παράγεται από την κλάση `leda_cmp_base`. Αυτό είναι χρήσιμο όταν η συνάρτηση `compare()` εξαρτάται από μια καθολική παράμετρο.

Παράδειγμα

Έστω ότι θέλουμε να συγκρίνουμε πλευρές ενός γραφήματος τύπου `GRAPH<point, int>` σύμφωνα με την απόσταση των άκρων τους (κάθε κόμβος του γραφήματος συσχετίζεται με ένα σημείο του επιπέδου).

```
class cmp_edges_by_length: public leda_cmp_base<edge>
{
public:
    cmp_edges_by_length(const GRAPH<point,int>& g): G(g) {}

    int operator()(const edge& e, const edge& f) const
    {
        point pe = G[G.source(e)];
        point qe = G[G.target(e)];
        point pf = G[G.source(f)];
        point qf = G[G.target(f)];
        return compare(pe.sqr_dist(qe), pf.sqr_dist(qf));
    }
private:
    const GRAPH<point,int>& G;
};
```

- Η κλάση έχει έναν τελεστή συνάρτησης ο οποίος παίρνει δύο πλευρές ενός γραφήματος `G` και τις συγκρίνει σύμφωνα με το μήκος τους. Το γράφημα `G` είναι μια παράμετρος της συνάρτησης κατασκευής.

```

int main()
{
    GRAPH<point,int> G;
    cmp_edges_by_length cmp(G);

    list<edge> E = G.all_edges();
    E.sort(cmp);
    return 0;
}

```

- Στο κυρίως πρόγραμμα ορίζουμε την `cmp(G)` σαν ένα στιγμιότυπο της κλάσης `cmp_edges_by_length` και μετά μεταβιβάζουμε την `cmp` σαν το αντικείμενο σύγκρισης της συνάρτησης ταξινόμησης του τύπου `list<edge>`.
- Στην υλοποίηση της συνάρτησης ταξινόμησης η σύγκριση μεταξύ δύο πλευρών γίνεται με κλήση της `cmp(e, f)`.

Παράμετροι Υλοποίησης

- Μερικοί τύποι της LEDA (π.χ. λεξικά, ουρές προτεραιότητας, κλπ) παρέχονται με διάφορες υλοποιήσεις.
- Ο χρήστης μπορεί να επιλέξει μια συγκεκριμένη υλοποίηση δίνοντας το όνομά της σαν μια επιπλέον παράμετρο.

«Κανονικός» παραμετρικός τύπος :

$T\langle T_1, \dots, T_k \rangle$

Παραμετρικός τύπος με υλοποίηση `xyz_impl`:

$T\langle T_1, \dots, T_k, xyz_impl \rangle$

Παράδειγμα

```
#include <LEDA/core/d_array.h>
#include <LEDA/core/impl/skiplist.h>
using namespace leda;

int main()
{
    // skiplist implementation
    d_array<string, int, skiplist> N(0);
    // default implementation
    // d_array<string,int> N(0);

    string s;
    while (cin >> s)
        N[s]++;
    forall_defined(s, N)
        cout << s << " " << N[s] << endl;
    return 0;
}
```

- Κάθε τύπος `T<T1, ..., Tk, xyz_impl>` είναι μια παράγωγη κλάση του αντίστοιχου «κανονικού» τύπου `T<T1, ..., Tk>`.



- Μπορούμε να μεταβιβάσουμε ένα στιγμιότυπο του τύπου `T<T1, ..., Tk, xyz_impl>` σαν όρισμα σε μια συνάρτηση που έχει τυπική παράμετρο τύπου `T<T1, ..., Tk>&` (αν δεν υπήρχε τέτοια σχέση κληρονομικότητας η μεταβίβαση δεν θα ήταν δυνατή, αφού στην C++ template κλάσεις με διαφορετικά ορίσματα τύπων θεωρούνται τελείως ξεχωριστές κλάσεις).

```
void word_count(d_array<string, int>& N)
{
    string s;
    while (cin >> s)
        N[s]++;
    forall_defined(s, N)
        cout << s << " " << N[s] << endl;
}
```

Κάθε υλοποίηση ενός `d_array` μπορεί να μεταβιβαστεί στη συνάρτηση `word_count`.

```
d_array<string, int> N1(0);
word_count(N1);
d_array<string, int, skiplist> N2(0);
word_count(N2);
```

Έλεγχος Ορθότητας Προγράμματος

- Διαδικασία που μας βοηθάει να αυξήσουμε σημαντικά το ποσοστό βεβαιότητας για μια σωστή υλοποίηση.
- Έστω P ένα πρόγραμμα το οποίο υπολογίζει μια συνάρτηση f . Θα λέμε ότι το P *μπορεί να ελεγχθεί* αν για κάθε είσοδο x επιστρέφει μια τιμή y , την υποτιθέμενη τιμή της $f(x)$, πιθανόν μαζί με επιπλέον πληροφορία I η οποία μας επιτρέπει *εύκολα* να επαληθεύσουμε ότι πραγματικά $y = f(x)$.
- *Εύκολα επαληθεύσιμο:*
 - (i) Πρέπει να υπάρχει ένα απλό πρόγραμμα C (πρόγραμμα ελέγχου) το οποίο, δεδομένων των x, y και I , να ελέγχει ότι πραγματικά $y = f(x)$. Το C πρέπει να είναι τόσο απλό που η ορθότητά του να είναι προφανής.
 - (ii) Ο χρόνος εκτέλεσης (ασυμπτωτικός ή μη) του C με είσοδο x, y και I δεν πρέπει να είναι μεγαλύτερος από εκείνον του P με είσοδο x .

Παράδειγμα - 1

- Θεωρείστε ένα πρόγραμμα το οποίο παίρνει σαν είσοδο έναν $m \times n$ πίνακα A και ένα $m \times 1$ διάνυσμα b και εξετάζει αν το γραμμικό σύστημα $A \cdot x = b$ έχει λύση ή όχι (επιστροφή λογικής τιμής true ή false).
- Το παραπάνω πρόγραμμα δεν μπορεί να ελεγχθεί ως προς την ορθότητά του. Για να μπορεί να ελεγχθεί πρέπει να επεκταθεί η διασύνδεσή του ως εξής.

Με είσοδο A και b το πρόγραμμα επιστρέφει είτε:

- «το σύστημα έχει λύση» και ένα διάνυσμα x έτσι ώστε $A \cdot x = b$, ή
 - «το σύστημα δεν έχει λύση» και ένα διάνυσμα c έτσι ώστε $c^T \cdot A = 0$ και $c^T \cdot b \neq 0$.
- Τώρα το πρόγραμμα είναι εύκολο να ελεγχθεί: απαιτούνται το πολύ δύο (απλοί και εύκολα ελέγξιμοι) πολλαπλασιασμοί πίνακα με διάνυσμα και διάνυσμα με διάνυσμα.

Παράδειγμα - 2

- Θεωρείστε ένα πρόγραμμα το οποίο παίρνει σαν είσοδο ένα γράφημα G και εξετάζει αν το G είναι επίπεδο ή όχι (επιστροφή λογικής τιμής true ή false.)
- Για να μπορεί να ελεγχθεί το παραπάνω πρόγραμμα ως προς την ορθότητά του, πρέπει να επεκταθεί ως εξής.

Με είσοδο G το πρόγραμμα επιστρέφει είτε :

- «επίπεδο» μαζί με μια επίπεδη απεικόνιση του G , ή
- «μη-επίπεδο» μαζί με ένα υπογράφημα του G ομοιομορφικό με το K_5 ή το $K_{3,3}$ (υπογραφήματα Kuratowski).

Τι κερδίζουμε από τον έλεγχο ορθότητας;

1. Η απάντηση του προγράμματος μπορεί να επαληθευθεί για κάθε ξεχωριστό στιγμιότυπο εισόδου (\neq επαλήθευση προγράμματος (program verification) η οποία παρέχει εγγύηση για όλα τα στιγμιότυπα εισόδου).
2. Ο χρήστης μπορεί να αυξήσει την εμπιστοσύνη του στην ορθότητα του προγράμματος καταβάλοντας πολύ μικρή διανοητική προσπάθεια.
3. Ο υλοποιητής του προγράμματος μπορεί να δώσει μια πειστική ένδειξη της ορθότητας του χωρίς να αποκαλύψει τις λεπτομέρειες της υλοποίησης.
4. Ο έλεγχος ορθότητας επιτρέπει να χρησιμοποιούμε ένα πιθανώς μη-ορθό πρόγραμμα σαν να ήταν ορθό \Rightarrow πολύ χρήσιμο κατά τη φάση διόρθωσης σφαλμάτων (debugging).

5. Ο έλεγχος ορθότητας υποστηρίζει τη δοκιμή προγράμματος (testing).

```
for (int n = 0; n < 100; n++)
  for (int m = 0; m < 100; m++)
  {
    random_graph(G, n, m);
    // random graph with n nodes and m edges

    list<edge> M = MAX_CARD_MATCHING(G, OSC);
    CHECK_MAX_CARD_MATCHING(G, M, OSC);
  }
```

6. Ένας ελεγκτής (πρόγραμμα ελέγχου ορθότητας) μπορεί να γραφεί αν υπάρχει αυστηρός ορισμός του προβλήματος που επιλύει το πρόγραμμα που ελέγχει.

Π.χ. ένας αλγόριθμος επίλυσης ενός προβλήματος γραφημάτων μπορεί να υποθέτει ότι δεν υπάρχουν μεμονωμένοι κόμβοι στο γράφημα.

Διαχείριση Μνήμης

- Η LEDA παρέχει ένα αρκετά αποδοτικό σύστημα διαχείρισης μνήμης που χρησιμοποιείται για όλους τους τύπους στοιχείων και των τύπων `node`, `edge`.
- Μπορεί πολύ εύκολα να χρησιμοποιηθεί από οποιαδήποτε κλάση `T` που ορίζει ο χρήστης με την προσθήκη της κλήσης της μακρο-εντολής `LEDA_MEMORY (T)` στον ορισμό της κλάσης.

```

class pair
{
    public:
        pair() { x = y = 0; }
        /* pair uses the default versions of copy constructor,
           assignment operator, and destructor */
        friend void Read(pair& p, istream& is)
            { is >> p.x >> p.y; }
        friend void Print(const pair& p, ostream& os)
            { os << p.x << " " << p.y; }
        friend int compare(const pair&, const pair&);

        LEDA_MEMORY(pair);

    private:
        double x, y;
};

```

Η κλήση της `LEDA_MEMORY(T)` δημιουργεί τελεστές `new` και `delete` για την κλάση `T` που βασίζονται στον διαχειριστή μνήμης της LEDA. Τα πλεονεκτήματα είναι τα εξής:

- Η μνήμη κατανέμεται σε μεγάλα τμήματα (chunks) \Rightarrow αποφεύγονται συχνές και χρονοβόρες κλήσεις του κατανομέα μνήμης.
- Η μνήμη που απελευθερώνεται από τον τελεστή `delete` επαναχρησιμοποιείται από μελλονικές κλήσεις του τελεστή `new`, δηλ. ο διαχειριστής μνήμης προσφέρει «συλλογή απορριμάτων» (garbage collection).

Μια χρήσιμη συνάρτηση για χρήση μνήμης και χρόνου

Το πρόγραμμα

```
list<point> L;  
for (int i = 0; i < 100000; i++)  
    L.append(point());  
list<point> L1 = L;  
  
print_statistics();
```

παράγει την ακόλουθη έξοδο στο σύστημα Zenon

STD_MEMORY_MGR (memory status)

size	used	free	blocks	bytes
8	27	995	1	8176
12	200000	214	294	2402568
20	39	369	1	8160
28	1	260	1	8148
40	100002	79	493	4003160
> 255	-	-	3	392
time: 0.52 sec		space: 6317.30 kb		