

Τεχνολογίες Υλοποίησης Αλγορίθμων

Ενότητα 5

Χρήστος Δ. Ζαρολιάγκης

Καθηγητής

Τμήμα Μηχ/κων Υπολογιστών & Πληροφορικής

Πανεπιστήμιο Πατρών

email: `zaro@ceid.upatras.gr`

Ενότητα 5

- Αλγόριθμοι Συντομότερων Διαδρομών και Υλοποιήσεις
 - Γενικά Κόστη Πλευρών
- Ελεγκτής Ορθότητας Αλγορίθμων Συντομότερων Διαδρομών
- Αρχέτυπα για Αλγορίθμους Γραφημάτων

- Γενικά Κόστη Πλευρών

```
bool BELLMAN_FORD_T(const graph& G, node s,  
                    const edge_array<NT>& c,  
                    node_array<NT>& dist,  
                    node_array<edge>& pred)
```

Επιστρέφει `false` αν $\delta(v) = -\infty$ για κάποια κορυφή v , αλλιώς επιστρέφει `true`.

Χρόνος : $O(nm)$.

Γενικευμένος Αλγόριθμος ΕΣΔΑΚ

for all $\{ v \in V \mid d(v) = \infty; \pi(v) = nil; \}$

$d(s) = 0;$

while $\exists e = (u, v) \in E : d(v) > d(u) + c(e)$

$\{ \quad d(v) = d(u) + c(e); \pi(v) = e; \quad \}$

Γράφημα προκατόχων (οριζόμενο από τους δείκτες π):

$$P = \{e : e = \pi(v) \in E, v \in V\}$$

Ιδιότητες Γενικευμένου Αλγορίθμου ΕΣΔΑΚ

1. $d(s) = 0$ ανν $\pi(s) = nil$ και $d(v) < \infty$ ανν $\pi(v) \neq nil$ για $v \neq s$.
2. Αν $\pi(v) = e = (u, v)$, τότε $d(v) \geq d(u) + c(e)$.
3. Αν $\pi(v) \neq nil$, τότε η v είτε βρίσκεται σε έναν κύκλο του P , είτε είναι προσπελάσιμη από έναν κύκλο στο P , είτε είναι προσπελάσιμη από την s στο P . Αν $\pi(s) \neq nil$, τότε η s βρίσκεται σε έναν κύκλο του P .
4. Οι κύκλοι στο P έχουν αρνητικό κόστος.
5. Αν η v βρίσκεται σε έναν κύκλο του P ή είναι προσπελάσιμη από έναν κύκλο στο P , τότε $\delta(v) = -\infty$.
6. Αν $v \in V^f$ και $d(v) = \delta(v)$, τότε υπάρχει μια s - v διαδρομή στο P η οποία έχει κόστος $\delta(v)$.
7. Αν $d(v) = \delta(v)$ για όλες τις $v \in V^f$, τότε το επαγόμενο από τις κορυφές του V^f υπογράφημα του P είναι ένα ΔΣΔ.

Προβλήματα του Γενικευμένου Αλγορίθμου ΕΣΔΑΚ

- ΠΧΠ εκθετική (ακόμη και σε απουσία αρνητικών κύκλων)
- Ο αλγόριθμος δεν τερματίζει όταν υπάρχουν αρνητικοί κύκλοι.

- Διατήρηση ενός συνόλου κορυφών U που περιέχει τις κορυφές που προσωρινά δεν παραβιάζουν την τριγωνική ανισότητα:

$$U \supseteq \{u : d(u) < \infty \text{ και } \exists (u, v) \in E \text{ με } d(u) + c(u, v) < d(v)\}$$



```

for all {  $v \in V$   $d(v) = \infty$ ;  $\pi(v) = nil$ ; }
 $d(s) = 0$ ;
 $U = \{s\}$ ;
while  $U \neq \emptyset$ 
{ επέλεξε  $u \in U$ ;  $U = U - \{u\}$ ;
  for all  $e = (u, v) \in E$ 
  { if  $d(v) > d(u) + c(e)$  then
    {  $U = U \cup \{v\}$ ;  $d(v) = d(u) + c(e)$ ;  $\pi(v) = e$ ; }
  }
}

```

- Ποια κορυφή $u \in U$ πρέπει να επιλέξουμε;

Ύπαρξη Βέλτιστης Επιλογής:

- Για κάθε $v \in V^f$ με $d(v) > \delta(v)$, υπάρχει κορυφή $u \in U$ με $d(u) = \delta(u)$ και η οποία βρίσκεται σε μια s - v ΣΔ.
- Όταν μια κορυφή u με $d(u) = \delta(u)$ διαγράφεται από το U , τότε δεν προστίθεται ποτέ ξανά στο U .

- Σχέση μεταξύ (d, π) και $(dist, pred)$

$$pred[v] = \pi(v), \quad dist[v] = \begin{cases} d(v) & \text{αν } d(v) < \infty \\ \text{αυθαίρετη} & \text{αν } d(v) = +\infty \end{cases}$$

- Αναπαράσταση του $+\infty$ και σύγκριση

$$d(v) = +\infty \quad \text{αν } v \neq s \text{ και } \pi(v) = nil$$

Σύγκριση $d < d(v)$, $d \in \mathbb{R}$:

```
(pred[v] == nil && v != s) || d < dist[v]
```

Ο Αλγόριθμος Bellman-Ford

- Πως επιλέγουμε κορυφές από το U ;

- ▶ Οργάνωση του αλγορίθμου σε φάσεις.

- $U_i = U$ στην αρχή της φάσης $i, i \geq 0, U_0 = \{s\}$.

- ▶ Στην φάση i , όλες οι κορυφές του U_i διαγράφονται από το U και οι νέες κορυφές που προστίθενται στο U εισάγονται στο U_{i+1} .

- Υλοποίηση του U : με χρήση μιας ουράς Q .

- ▶ Φάση i : όλες οι κορυφές του U_i βρίσκονται στην αρχή της Q και όλες οι κορυφές του U_{i+1} βρίσκονται στο τέλος της Q (διαχωρισμένες από ένα στοιχείο με τιμή nil).

- ▶ Όταν ο διαχωριστής nil βρίσκεται στην αρχή της Q , τότε αυξάνουμε κατά 1 την `phase_count`.

- ▶ `node_array<bool> in_Q: in_Q[v] = true, αν $v \in Q$`
(αποφυγή επανεισαγωγών στην Q).

Ο Αλγόριθμος Bellman-Ford

- Ο αλγόριθμος τερματίζει όταν $Q = \emptyset$, ή όταν $i = n$.
- Στην πρώτη περίπτωση: $d(v) = \delta(v), \forall v \in V$.
- Στη δεύτερη περίπτωση: $d(v) = \delta(v), \forall v \in V^+ \cup V^f$.
- Οι κορυφές του V^- αντιμετωπίζονται σε μια μεταγενέστερη φάση.

Ο Αλγόριθμος Bellman-Ford

```
// BF: helper

template <class NT>
bool BELLMAN_FORD_T(const graph& G, node s, const edge_array<NT>& c,
                    node_array<NT>& dist, node_array<edge>& pred )

{ int n = G.number_of_nodes();
  int phase_count = 0;

  b_queue<node> Q(n+1);
  node_array<bool> in_Q(G, false);
  node u, v;
  edge e;

  forall_nodes(v, G) pred[v] = nil;

  dist[s] = 0;
  Q.append(s); in_Q[s] = true;
  Q.append((node) nil); // end marker
```

```

while( phase_count < n )
{ u = Q.pop();
  if ( u == nil)
  { phase_count++;
    if ( Q.empty() ) return true;
    Q.append((node) nil);
    continue;
  }
  else in_Q[u] = false;

  NT du = dist[u];

  forall_adj_edges(e,u)
  { v = G.opposite(u,e); // makes it also work for ugraphs
    NT d = du + c[e];
    if ( (pred[v] == nil && v != s) || d < dist[v] )
    { dist[v] = d; pred[v] = e;

      if ( !in_Q[v] ) { Q.append(v); in_Q[v] = true; }
    }
  }
}
// BF: postprocessing
return false;
}

```

Μεταγενέστερη Φάση ($Q \neq \emptyset$ μετά από n φάσεις)

- Πρόβλημα: μπορεί να υπάρχουν κορυφές στο V^- που δεν είναι ακόμη προσπελάσιμες από έναν κύκλο στο γράφημα P .
- Πως το επιτυγχάνουμε αυτό χωρίς να χρειαστεί να τρέξουμε τον αλγόριθμο για έναν μεγαλύτερο αριθμό φάσεων;

Βασικές Ιδιότητες μετά από n φάσεις

$$\delta_k(v) = \min\{c(p) : p \text{ μια } s\text{-}v \text{ διαδρομή με το πολύ } k \text{ πλευρές}\}$$

1. $d(v) \leq \delta_n(v)$ και αν $v \in U$, τότε $d(v) < \delta_{n-1}(v)$.
2. $s \in V^f$ αν $\pi(s) = nil$.
3. Κάθε κορυφή $u \in U$ βρίσκεται είτε σε έναν κύκλο του P , είτε είναι προσπελάσιμη από έναν κύκλο στο P .
4. Κάθε κορυφή $v \in V^-$ είναι προσπελάσιμη στο G από κάποια $u \in U$.
5. Αν $\pi(s) \neq nil$, τότε κάθε κορυφή προσπελάσιμη από την s βρίσκεται είτε σε έναν κύκλο του P , είτε είναι προσπελάσιμη από έναν κύκλο στο P .

Μεταγενέστερη Φάση

- Έστω $V^f \neq \emptyset$ ($\pi(s) = nil$), και $R \supseteq V^f$ το σύνολο των κορυφών που είναι προσπελάσιμες από την s στο P .
- Όλες οι κορυφές $w \in R$ που είναι προσπελάσιμες από μια κορυφή $u \in U$ ανήκουν στο V^- και επομένως πρέπει να ενημερωθούν οι $\pi(w)$.
- Εκτελούμε ΑΠΒ από κάθε $u \in U$ και όταν μια κορυφή $w \in R$ προσπελαίνεται μέσω της πλευράς (z, w) , τότε θέτουμε $\pi(w) = (z, w)$
 \implies όλες οι κορυφές του $R \cap V^-$ είναι προσπελάσιμες από τις κορυφές του U και από την (3) προσπελάσιμες από έναν κύκλο στο P .

- Πως προσδιορίζονται οι κορυφές του R ;

- ▶ ΑΠΒ από την s στο P , αφού πρώτα «κρύψουμε» τις πλευρές που δεν ανήκουν στο P (και μετά τις επαναφέρουμε στο αρχικό γράφημα).

```
// BF: postprocessing
if (pred[s] != nil) return false;
node_array<bool> in_R(G, false);
forall_edges(e, G)
    if (e != pred[G.target(e)]) ((graph*) &G)->hide_edge(e);
DFS(G, s, in_R); // sets in_R[v] = true for v in R
((graph*) &G)->restore_all_edges();
node_array<bool> reached_from_node_in_U(G, false);
forall_nodes(v, G)
    if (in_Q[v] && !reached_from_node_in_U[v])
        Update_pred(G, v, in_R, reached_from_node_in_U, pred);
```

```

// BF: helper

inline void Update_pred(const graph& G, node v,
                       const node_array<bool>& in_R,
                       node_array<bool>& reached_from_node_in_U,
                       node_array<edge>& pred)
{ reached_from_node_in_U[v] = true;
  edge e;
  forall_adj_edges(e, v)
  { node w = G.target(e);
    if ( !reached_from_node_in_U[w] )
    { if ( in_R[w] ) pred[w] = e;
      Update_pred(G, w, in_R, reached_from_node_in_U, pred);
    }
  }
}

```

- Γενικός Αλγόριθμος ΕΣΔΑΚ

```
template <class NT>
bool SHORTEST_PATH_T(const graph& G, node s, const edge_array<NT>& c,
                    node_array<NT>& dist, node_array<edge>& pred )
{
    if ( Is_Acyclic(G) )
    { ACYCLIC_SHORTEST_PATH_T(G, s, c, dist, pred);
      return true;
    }
    bool non_negative = true;
    edge e;
    forall_edges(e, G) if (c[e] < 0) non_negative = false;

    if (non_negative)
    { DIJKSTRA_T(G, s, c, dist, pred);
      return true;
    }
    return BELLMAN_FORD_T(G, s, c, dist, pred);
}
```

Αλγόριθμος ΕΟΣΔ

```
bool ALL_PAIRS_SHORTEST_PATHS_T (graph& G,  
                                  edge_array<NT> c,  
                                  node_matrix<NT> DIST)
```

Επιστρέφει `true` αν το G δεν έχει αρνητικό κύκλο, αλλιώς επιστρέφει `false`.

Στην τελευταία περίπτωση οι τιμές που επιστρέφονται από την $DIST$ είναι μη προσδιορισμένες.

Στην πρώτη περίπτωση, για όλα τα v και w ισχύει ότι: αν $\delta(v, w) < +\infty$, τότε $DIST(v, w) = \delta(v, w)$ και αν $\delta(v, w) = +\infty$, τότε η τιμή του $DIST(v, w)$ είναι μη προσδιορισμένη.

Χρόνος : $O(nm + n^2 \log n)$.

Ελεγκτής Ορθότητας Αλγορίθμων ΕΣΔΑΚ

- Ανάπτυξη προγράμματος `CHECK_SP_T(G, s, c, dist, pred)` το οποίο θα ελέγχει αν το ζεύγος $(dist, pred)$ είναι μια σωστή λύση του προβλήματος ΕΣΔΑΚ (G, s, c) .
- *Λάθος λύση*: σταματάει αμέσως η εκτέλεση του προγράμματος με διαγνωστικό μήνυμα λάθους ("assertion failed").
- *Σωστή λύση*: επιστρέφει ένα `node_array<int> label` με
$$label[v] < 0, \text{ αν } v \in V^-,$$
$$label[v] = 0, \text{ αν } v \in V^f,$$
$$label[v] > 0, \text{ αν } v \in V^+.$$

- Ορισμοί

$$U^+ = \{v; v \neq s \text{ και } \text{pred}[v] = \text{nil}\}$$

$$U^f = \begin{cases} \emptyset & \text{αν } \text{pred}[s] \neq \text{nil} \\ \{v; \exists s-v \text{ διαδομή στο } P \} & \text{αν } \text{pred}[s] = \text{nil} \end{cases}$$

$$U^- = \{v; v \text{ βρίσκεται σε έναν κύκλο του } P \\ \text{ή είναι προσπελάσιμη από έναν κύκλο στο } P\}$$

Έλεγχοι Ορθότητας

Το ζεύγος $(dist, pred)$ είναι μια σωστή λύση αν ικανοποιούνται οι παρακάτω συνθήκες/έλεγχοι.

- (1) $v \in U^+$ αν δεν υπάρχει s - v διαδρομή στο G .
- (2) Όλοι οι κύκλοι στο P έχουν αρνητικό κόστος.
- (3) Δεν υπάρχει πλευρά $(v, w) \in E$ με $v \in U^-$ και $w \in U^f$.
- (4) Για όλες τις πλευρές $e = (v, w) \in E$: αν $v \in U^f$ και $w \in U^f$ τότε $dist[v] + c[e] \geq dist[w]$.
- (5) Για όλες τις $v \in U^f$: αν $v = s$ τότε $dist[v] = 0$ και αν $v \neq s$ τότε $dist[v] = dist[u] + c[pred[v]]$ όπου u είναι η αρχική κορυφή της $pred[v]$.

Υλοποίηση - Συνθήκη (1)

- Με χρήση ΑΠΒ προσδιορίζουμε τις κορυφές που είναι προσπελάσιμες από την s και ελέγχουμε για όλα τα $v \neq s$: $pred[v] = nil$ αν δεν υπάρχει $s-v$ διαδρομή.
- Σε όλες τις κορυφές που δεν είναι προσπελάσιμες από την s δίνουμε label ίσο με PLUS.

```
// condition (1)

enum{ NEG_CYCLE = -2,  ATT_TO_CYCLE = -1,  FINITE = 0,
      PLUS = 1,  CYCLE = 2,  ON_CUR_PATH = 3,  UNKNOWN = 4  };

node_array<int> label(G, UNKNOWN);
node_array<bool> reachable(G, false);

DFS(G, s, reachable);

node v;
forall_nodes(v, G)
{ if (v != s)
  { assert( (pred[v] == nil) == (reachable[v] == false));
    if (reachable[v] == false) label[v] = PLUS;
  }
}
```

Υλοποίηση - Κατηγοριοποίηση Κορυφών

- Ακολουθούμε τη διαδρομή
 $[v, source(pred[v]), source(pred[source(pred[v]])], \dots]$
μέχρι να συναντήσουμε μια κορυφή δύο φορές (U^-) ή μέχρι να μην μπορούμε να προχωρήσουμε άλλο (U^f).
- U^- : κορυφές που ανήκουν σε έναν κύκλο στο P (label CYCLE) και κορυφές που είναι προσπελάσιμες από έναν κύκλο στο P (label ATT_TO_CYCLE).
- Η διαπέραση μιας διαδρομής δεν πρέπει να γίνεται πολλές φορές \Rightarrow η διαπέραση σταματά όταν συναντήσουμε μια κορυφή με label \neq UNKNOWN. Κατά τη διαπέραση όλες οι κορυφές της διαδρομής εισάγονται σε μία στοίβα S και παίρνουν label ON_CUR_PATH.

```

// classification of nodes

if (pred[s] == nil) label[s] = FINITE;
forall_nodes(v, G)
{ if ( label[v] == UNKNOWN )
  { stack<node> S;
    node w = v;
    while ( label[w] == UNKNOWN )
    { label[w] = ON_CUR_PATH;
      S.push(w);
      w = G.source(pred[w]);
    }
    // label all nodes on current path
  }
}

```

- Εξέταση κορυφής w με $\text{label} \neq \text{UNKNOWN}$:

- ▶ Αν $\text{label}[w] = \text{FINITE}$, τότε όλες οι κορυφές της διαδρομής ανήκουν στο U^f .
- ▶ Αν $\text{label}[w] = \text{CYCLE}$ ή $\text{label}[w] = \text{ATT_TO_CYCLE}$, τότε όλες οι κορυφές (εκτός της w) είναι προσπελάσιμες από έναν κύκλο. Αν επίσης η w ανήκει στην τρέχουσα διαδρομή, τότε ανήκει στον κύκλο.

```
// label all nodes on current path
```

```
int t = label[w];
if ( t == ON_CUR_PATH )
{ node z;
  do { z = S.pop();
      label[z] = CYCLE;
    }
  while ( z != w );
  while ( !S.empty() ) label[S.pop()] = ATT_TO_CYCLE;
}
else // t is CYCLE, ATT_TO_CYCLE, or FINITE
{ if ( t == CYCLE ) t = ATT_TO_CYCLE;
  while ( !S.empty() ) label[S.pop()] = t;
}
```

Υλοποίηση - Συνθήκη (2)

- Για κάθε κορυφή v με $label[v]=CYCLE$ διαπερνούμε τον κύκλο στον οποίο ανήκει, υπολογίζουμε το κόστος του και ελέγχουμε αν είναι αρνητικό. Αν ναι, τότε δίνουμε σε όλες τις κορυφές του κύκλου νέο $label$ ίσο με NEG_CYCLE . Αυτό εξασφαλίζει ότι κάθε κύκλος διαπερνάται μόνο μία φορά.

```
// condition (2)

forall_nodes(v, G)
{ if ( label[v] == CYCLE )
  { node w = v;
    NT cycle_length = 0;
    do
    { cycle_length += c[pred[w]];
      label[w] = NEG_CYCLE;
      w = G.source(pred[w]);
    } while (w != v);
    assert(cycle_length < 0);
  }
}
```

Υλοποίηση - Συνθήκες (3), (4) και (5)

```
//conditions (3), (4), and (5)

if ( label[s] == FINITE ) assert(dist[s] == 0);

edge e;
forall_edges(e, G)
{ node v = G.source(e);
  node w = G.target(e);
  if ( label[w] == FINITE )
  { assert( label[v] == FINITE || label[v] == PLUS);
    if ( label[v] == FINITE )
    { assert( dist[v] + c[e] >= dist[w] );
      if ( e == pred[w] )
        assert( dist[v] + c[e] == dist[w] );
    }
  }
}
```

Υλοποίηση - Συνολικό Πρόγραμμα

```
template <class NT>
node_array<int> CHECK_SP_T(const graph& G, node s,
                          const edge_array<NT>& c,
                          const node_array<NT>& dist,
                          const node_array<edge>& pred)
{
    // condition (1)
    // classification of nodes
    // condition (2)
    // conditions (3), (4), and (5)
    return label;
}
```

Αρχέτυπα για Αλγορίθμους Γραφημάτων

- Οι αλγόριθμοι γραφημάτων της LEDA μπορούν να εφαρμοσθούν σε οποιονδήποτε αριθμητικό τύπο τιμών (που πληροί κάποιες προϋποθέσεις) οι οποίες είναι συσχετισμένες με κορυφές ή πλευρές ενός γραφήματος. Ο μηχανισμός αρχετύπων χρησιμοποιείται γιαυτό το σκοπό.

Παράδειγμα: αλγόριθμος εύρεσης συντομότερων διαδρομών από μία αρχική κορυφή (ΕΣΔΑΚ – single-source shortest path).

```
template <class NT>
void DIJKSTRA_T(const graph& G, node s,
               const edge_array<NT>& c,
               node_array<NT>& dist,
               node_array<edge>& pred);
```

- Η παράμετρος αρχετύπου NT μπορεί να συγκεκριμενοποιηθεί με οποιονδήποτε αριθμητικό τύπο, ο οποίος πληροί κάποιες συντακτικές και σημασιολογικές προϋποθέσεις.

Π.χ. πρέπει να είναι γραμμικά διατεταγμένος και η λειτουργία της πρόσθεσης πρέπει να είναι μονότονη.

Επιθυμητοί Στόχοι

- Οι πιο συχνά χρησιμοποιούμενες συγκεκριμενοποιήσεις πρέπει να είναι προμεταγλωτισμένες.
- Οι προ-συγκεκριμενοποιημένες εκδοχές πρέπει να μπορούν να χρησιμοποιούνται εναλλακτικά με τις εκδοχές αρχτύπων.

Μηχανισμός Επίτευξης αυτών των Στόχων στη LEDA

- Για κάθε συνάρτηση `Alg` που υλοποιεί κάποιον αλγόριθμο υπάρχουν τρία αρχεία:
 - `Alg.h`: βρίσκεται στον κατάλογο `LEDAROOT/incl/LEDA` και περιέχει τις δηλώσεις όλων των συναρτήσεων.

Η προ-συγκεκριμενοποιημένη εκδοχή έχει όνομα `Alg`, ενώ η εκδοχή αρχετύπου έχει όνομα `Alg_T`.
 - `Alg.t`: βρίσκεται στον κατάλογο `LEDAROOT/incl/LEDA/templates` και περιέχει τον ορισμό της συνάρτησης αρχετύπου.
 - `_Alg.c`: βρίσκεται στον κατάλογο `LEDAROOT/src` και περιέχει τις υλοποιήσεις των συγκεκριμενοποιήσεων με βάση τη συνάρτηση αρχετύπου.

Το αρχείο αυτό είναι προμεταγλωπτισμένο στο αρχείο κώδικα αντικειμένου `_Alg.o` που περιέχεται σε κάποια από τις βιβλιοθήκες της LEDA.

- Χρήση των διαφορετικών εκδοχών

- *Συνάρτηση αρχετύπου:*

```
#include <LEDA/templates/Alg.t>
```

- *Προ-συγκεκριμενοποιημένη συνάρτηση:*

```
#include <LEDA/Alg.h>
```

Παράδειγμα ΕΣΔΑΚ: αλγόριθμος Dijkstra

```
// file dijkstra.h

#ifndef DIJKSTRA_H
#define DIJKSTRA_H

#include <LEDA/graph.h>

template <class NT>
void DIJKSTRA_T(const graph& G, node s,
               const edge_array<NT>& c,
               node_array<NT>& dist,
               node_array<edge>& pred);

// next come the pre-instantiated versions //

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

// and, similarly, for double, ...

#endif
```

```
// file dijkstra.t

#include <LEDA/dijkstra.h>

template <class NT>
void DIJKSTRA_T(const graph& G, node s,
                const edge_array<NT>& c,
                node_array<NT>& dist,
                node_array<edge>& pred)
{
    /* implementation of DIJKSTRA_T */
}
```

```
// file _dijkstra.c

#include <LEDA/templates/dijkstra.t>

void DIJKSTRA(const graph& G, node s,
              const edge_array<int>& c,
              node_array<int>& dist,
              node_array<edge>& pred)
{
    DIJKSTRA_T(G, s, c, dist, pred);
}

// and, similarly, for double ...
```

Χρήση προ-συγκεκριμενοποιημένης εκδοχής σε πρόγραμμα εφαρμογής

```
#include <LEDA/dijkstra.h>

// define G, s, c, dist, pred with number type int

DIJKSTRA(G, s, c, dist, pred);
```

Χρήση εκδοχής αρχετύπου σε πρόγραμμα εφαρμογής

```
#include <LEDA/templates/dijkstra.t>

// define G, s, c, dist, pred for any number type NT

DIJKSTRA_T(G, s, c, dist, pred);

// define G, s, c, dist, pred for number type int
// and use template version

DIJKSTRA_T(G, s, c, dist, pred);

// use pre-instantiated version

DIJKSTRA(G, s, c, dist, pred);
```

Ένας συνιστώμενος διαφορετικός τρόπος χρήσης

- Τα `.t`-αρχεία δεν πρέπει να συμπεριλαμβάνονται άμεσα στα προγράμματα εφαρμογής, διότι μπορεί να περιέχουν βοηθητικές συναρτήσεις οι οποίες να δημιουργούν πρόβλημα με τον χώρο ονομάτων του προγ/τος εφαρμογής.
- Δημιουργία *ενδιάμεσων* αρχείων.

Π.χ. συγκεκριμενοποίηση της `DIJKSTRA_T` με τον τύπο δεδομένων `real` της LEDA.

```
// real_dijkstra.h

#include <LEDA/real.h>

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

```

// real_dijkstra.c
#include "real_dijkstra.h"
#include <LEDA/templates/dijkstra.t>

void DIJKSTRA(const graph& G, node s,
              const edge_array<real>& c,
              node_array<real>& dist,
              node_array<edge>& pred)
{
    DIJKSTRA_T(G, s, c, dist, pred);
}

```

- Το αρχείο `real_dijkstra.h` συμπεριλαμβάνεται στο πρόγραμμα εφαρμογής, ενώ το αρχείο `real_dijkstra.c` προμεταγλωπίζεται και προστίθεται στη βιβλιοθήκη κώδικα αντικειμένου από τον διασυνδετή.