



Τεχνολογίες Υλοποίησης Αλγορίθμων

Χρήστος Ζαρολιάγκης

Καθηγητής

Τμήμα Μηχ/κων Η/Υ & Πληροφορικής

Πανεπιστήμιο Πατρών

email: zaro@ceid.upatras.gr

Γρηγόρης Πράσιнос

Υποψήφιος Διδάκτωρ

Τμήμα Μηχ/κων Η/Υ & Πληροφορικής

Πανεπιστήμιο Πατρών

Boost Graph Library

Γενικευμένος προγραμματισμός και αλγόριθμοι γραφημάτων



Ευρωπαϊκή Ένωση
Ευρωπαϊκό Κοινωνικό Ταμείο



ΕΠΙΧΕΙΡΗΣΙΑΚΟ ΠΡΟΓΡΑΜΜΑ
ΕΚΠΑΙΔΕΥΣΗ ΚΑΙ ΔΙΑ ΒΙΟΥ ΜΑΘΗΣΗ
επένδυση στην κοινωνία της γνώσης

ΥΠΟΥΡΓΕΙΟ ΠΑΙΔΕΙΑΣ & ΘΡΗΣΚΕΥΜΑΤΩΝ, ΠΟΛΙΤΙΣΜΟΥ & ΑΘΛΗΤΙΣΜΟΥ
ΕΙΔΙΚΗ ΥΠΗΡΕΣΙΑ ΔΙΑΧΕΙΡΙΣΗΣ

Με τη συγχρηματοδότηση της Ελλάδας και της Ευρωπαϊκής Ένωσης

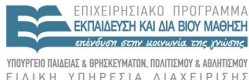


ΕΥΡΩΠΑΙΚΟ ΚΟΙΝΩΝΙΚΟ ΤΑΜΕΙΟ

- Το παρόν εκπαιδευτικό υλικό υπόκειται σε άδειες χρήσης Creative Commons.
- Για εκπαιδευτικό υλικό, όπως εικόνες, που υπόκειται σε άλλου τύπου άδειας χρήσης, η άδεια χρήσης αναφέρεται ρητώς.



- Το παρόν εκπαιδευτικό υλικό έχει αναπτυχθεί στα πλαίσια του εκπαιδευτικού έργου του διδάσκοντα.
- Το έργο «**Ανοικτά Ακαδημαϊκά Μαθήματα στο Πανεπιστήμιο Πατρών**» έχει χρηματοδοτήσει μόνο τη αναδιαμόρφωση του εκπαιδευτικού υλικού.
- Το έργο υλοποιείται στο πλαίσιο του Επιχειρησιακού Προγράμματος «Εκπαίδευση και Δια Βίου Μάθηση» και συγχρηματοδοτείται από την Ευρωπαϊκή Ένωση (Ευρωπαϊκό Κοινωνικό Ταμείο) και από εθνικούς πόρους.



Με τη συγχρηματοδότηση της Ελλάδας και της Ευρωπαϊκής Ένωσης



- Εισαγωγή στη Boost Graph Library
- Κατασκευή ενός γραφήματος
- Τρόποι διάτρεξης ενός γραφήματος
- Παραδείγματα εφαρμογής αλγορίθμων σε γραφήματα

Ο σκοπός της Boost Graph Library είναι η υλοποίηση γενικευμένων αλγορίθμων για γραφήματα.

Παρόμοια με την STL, οι ιδιότητες και οι διάφοροι τρόποι αναπαράστασης ενός γραφήματος περιγράφονται γενικευμένα και οι αλγόριθμοι προσπελαίνουν τα γραφήματα μέσω ειδικών iterators χωρίς να εξαρτώνται από την εσωτερική τους αναπαράσταση.

Με την περιγραφή των concepts που παρουσιάζονται στους αλγόριθμους γραφημάτων η BGL φιλοδοξεί να είναι μια ανοικτή και εύκολα επεκτάσιμη βιβλιοθήκη χωρίς να θυσιάζει ορθότητα και αποδοτικότητα.

Το απλούστερο concept στη BGL είναι το *Graph* που ορίζει ότι ένα γράφημα πρέπει να έχει κόμβους και ακμές.

Η αναγνώριση και η προσπέλαση των κόμβων και των ακμών ενός γραφήματος πραγματοποιείται μέσω των *descriptors*.

Οι *descriptors* παρέχονται από την κλάση *graph_traits* που αντιστοιχεί στην εκάστοτε κλάση γραφήματος. Οι *descriptors* κόμβων και ακμών παρέχουν έναν τελεστή ανάθεσης και έλεγχο ισότητας.

Παράδειγμα: ανίχνευση self-loops.

```
template<typename Graph>
bool isSelfLoop(typename graph_traits<Graph>
                ::edge_descriptor e,
                const Graph &g)
{
    typename graph_traits<Graph>
        ::vertex_descriptor u, v;
    u = source(e, g);
    v = target(e, g);
    return u == v;
}
```

Συνήθως σε ένα γράφημα υπάρχει κάποια πληροφορία (float συντεταγμένες, int αποστάσεις) που συσχετίζεται με τους κόμβους και τις ακμές του. Μία τέτοια συσχέτιση μπορεί να υλοποιηθεί με πολλούς τρόπους (π.χ. με `node_arrays` και `edge_arrays` στη LEDA).

Για τη συγγραφή γενικευμένων αλγορίθμων απαιτείται ένας γενικευμένος τρόπος πρόσβασης σε αυτή τη πληροφορία που να είναι ανεξάρτητος από την υλοποίηση. Στη BGL χρησιμοποιείται το concept του *property map*.

Π.χ. Το concept του `LvaluePropertyMap` ορίζει τρεις μεθόδους:

- `get(p_map, key)`: επιστρέφει την τιμή που αντιστοιχεί στο κλειδί `key`
- `put(p_map, key, value)`: εισάγει ή ενημερώνει το `(key, value)`
- `p_map[key]`: επιστρέφει μία αναφορά στην αντίστοιχη τιμή `value`

Για να μπορούμε να προσπελάσουμε ένα γράφημα, η BGL ορίζει 5 βασικά είδη iterators:

- `vertex_iterator`: Διάσχιση κόμβων του γραφήματος.
Παρέχει πρόσβαση σε `vertex descriptors`.
- `edge_iterator`: Διάσχιση ακμών του γραφήματος.
Παρέχει πρόσβαση σε `edge descriptors`.
- `out_edge_iterator`: Διάσχιση εξερχόμενων ακμών ενός κόμβου.
- `in_edge_iterator`: Διάσχιση εισερχόμενων ακμών ενός κόμβου.
- `adjacency_iterator`: Διάσχιση προσκείμενων κόμβων ενός κόμβου.

- Κάθε τύπος γραφήματος παρέχει τους δικούς του iterators μέσω της κλάσης `graph_traits`.
- Για να οριοθετηθεί το πρώτο και το τελευταίο στοιχείο, είτε για κόμβους `std::pair<vertex_iterator, vertex_iterator>` είτε για ακμές `std::pair<edge_iterator, edge_iterator>`, από τις παρακάτω συναρτήσεις παρέχονται οι iterators που δείχνουν στην αρχή και στο τέλος της συλλογής των στοιχείων :
 - `vertices(g)`
 - `edges(g)`
 - `out_edges(v, g)`
 - `in_edges(v, g)`
 - `adjacent_vertices(v, g)`

- Το concept που περιγράφει την προσθήκη και διαγραφή κόμβων και ακμών είναι το *MutableGraph*. Οι αντίστοιχες συναρτήσεις είναι:
 - `add_vertex(g)`
 - `remove_vertex(g)`
 - `add_edge(s, t, g)`
 - `remove_edge(s, t, g)`
- Η `add_vertex()` επιστρέφει τον αντίστοιχο `vertex descriptor` ενώ η `add_edge()` ένα `std::pair` που περιέχει τον αντίστοιχο `edge descriptor` και ένα `flag` που υποδεικνύει αν έγινε η εισαγωγή ή όχι.

- Η STL παρέχει τη δυνατότητα προσαρμογής της συμπεριφοράς των αλγορίθμων με χρήση του κατάλληλου functor (π.χ. `sort`).
- Η BGL παρέχει έναν παρόμοιο μηχανισμό για την προσαρμογή αλγορίθμων γραφημάτων, τους *algorithm visitors*. Ένας visitor δεν παρέχει τον `operator()` αλλά πολλές μεθόδους οι οποίες καλούνται σε συγκεκριμένα σημεία του αλγορίθμου (π.χ. κατά την ανακάλυψη ή εγκατάλειψη ενός κόμβου).
- Η BGL ορίζει 4 (προς το παρόν) είδη visitors και τις μεθόδους τους: BFS, DFS, Dijkstra, BellmanFord.

Η ιδέα της `επίσκεψης` - Παράδειγμα

```
template<typename VertexNameMap>
class bfs_name_printer
    : public default_bfs_visitor
{
public:
    bfs_name_printer(VertexNameMap n_map)
        : m_name_map(n_map) { }

    template<typename Vertex, typename Graph>
    void discover_vertex(Vertex u, const Graph &) const
    {
        std::cout << m_name_map[u] << " ";
    }
private:
    VertexNameMap m_name_map;
};
```

Η ιδέα της `επίσκεψης` - Παράδειγμα

Δημιουργείται ένα στιγμιότυπο του `bfs_name_printer` και δίνεται σαν όρισμα στη `breadth_first_search()`:

```
std::map<Vertex, std::string> names;
// Assume "names" is filled with values
names[u1]="IQPatras2015"; names[u2]="ALGO2015";

// Make a property map out of the std::map
typedef associative_property_map<
    std::map<Vertex, std::string> > VertexNameMap;
VertexNameMap name_map(names);

bfs_name_printer<VertexNameMap> vis(name_map);
// Using named parameters version of bfs
breadth_first_search(g, s, visitor(vis));
```

Η BGL παρέχει τρεις τύπους γραφημάτων: `adjacency_list`, `adjacency_matrix`, και `compressed_sparse_row_graph`.

Οι τρεις τύποι μοντελοποιούν διαφορετικά concepts οπότε προσφέρουν διαφορετική πρόσβαση στους κόμβους και τις ακμές του γραφηματος. Π.χ. το `adjacency_list` δεν μοντελοποιεί το `BidirectionalGraph` (πρόσβαση σε εισερχόμενες ακμές) οπότε δεν παρέχει τον iterator `in_edges()` ενώ το `compressed_sparse_row_graph` δεν μοντελοποιεί το `MutableGraph`, προσφέρει δηλαδή ένα στατικό γράφημα.

Η κλάση `adjacency_list` είναι template με παραμέτρους :

- `VertexList` και `EdgeList` που ορίζουν τις κλάσεις που θα χρησιμοποιηθούν για την αποθήκευση των κόμβων και των ακμών.
- `Directed` για τον τύπο του γραφήματος
- `VertexProperties`, `EdgeProperties`, `GraphProperties` που ορίζουν τις κλάσεις που θα χρησιμοποιηθούν για την αντιστοίχιση τιμών με τους κόμβους, τις ακμές και το ίδιο το γράφημα.

Η επιλογή των παραμέτρων `VertexList` και `EdgeList` καθορίζει την χρονική πολυπλοκότητα των λειτουργιών του γραφήματος και πρέπει να συμβαδίζει με τις απαιτήσεις του προβλήματος.

Παράδειγμα χρήσης γραφήματος

```
#include <boost/graph/adjacency_list.hpp>
#include <boost/graph/dijkstra_shortest_paths.hpp>
#include <iostream>
#include <string>
using namespace boost;
using namespace std;

int main(int argc, char *argv[]) {
    typedef adjacency_list<vecS, vecS, undirectedS
        string, int> Graph;
    typedef graph_traits<Graph>::vertex_descriptor Vertex;
    typedef graph_traits<Graph>::edge_descriptor Edge;
    Graph G;
    Vertex v1 = add_vertex(G);
    Vertex v2 = add_vertex(G);
    Edge e1 = add_edge(v1, v2, G).first;
    G[v1] = "Athens";
    G[v2] = "Patra";
    G[e1] = 215;
```

Παράδειγμα χρήσης γραφήματος (συνέχεια)

```
// vector for storing distance property
vector<edge_bundle_type<Graph>::type >
    dist(num_vertices(G));

// invoke "named parameters" variant of Dijkstra
dijkstra_shortest_paths(G, v1,
    weight_map(get(edge_bundle,G))
    .distance_map(&dist[0]));

cout << "distances from start vertex:" << endl;
graph_traits<Graph>::vertex_iterator vi;
for(vi = vertices(G).first;
    vi != vertices(G).second;
    ++vi)
    cout << "distance(" << G[*vi] << ") = "
        << dist[*vi] << endl;
}
```

Παρέχονται διάφοροι προσαρμογείς (adaptors), π.χ.:

- `reverse_graph` που αντιστρέφει τη φορά των ακμών
- `filtered_graph` που παρέχει μία περιορισμένη όψη του γραφήματος
- Προσαρμογέας για χρήση του τύπου γραφήματος της LEDA
- Προσαρμογέας για χρήση του τύπου `std::vector< std::list<int> >` ως γράφημα

Η τοπολογική διάταξη μπορεί να υλοποιηθεί με χρήση του `depth_first_search()`.

Θα δημιουργήσουμε ένα κατάλληλο visitor με υλοποιημένη τη μέθοδο `finish_vertex` ο οποίος θα δοθεί σαν όρισμα στην `depth_first_search()`.

Η υλοποίηση του αλγορίθμου δεν απαιτεί σχεδόν καμμία γνώση για την εσωτερική υλοποίηση του γραφήματος.

Παράδειγμα: τοπολογική διάταξη

```
// Outputs nodes in reverse topological order
template<typename OutputIterator>
class topo_sort_visitor: public default_dfs_visitor
{
    template <typename Vertex, typename Graph>
    void finish_vertex(Vertex u, const Graph &)
    { *m_iter++ = u; }
    template <typename Edge, typename Graph>
    void back_edge(const Edge& u, Graph&)
    { // throw an exception - no topo sort possible }
    // ... constructor and private iterator copy
};

template <typename Graph, typename OutputIterator>
void topological_sort(Graph &g, OutputIterator result)
{
    topo_sort_visitor<OutputIterator> vis(result);
    depth_first_search(g, visitor(vis));
}
```

Υπάρχουν αλγόριθμοι για:

- Αναζήτηση
- Συνεκτικές συνιστώσες
- Ελάχιστα γεννητικά δένδρα
- Συντομότερες διαδρομές
- Ροές
- Επίπεδα γραφήματα

Πλεονεκτήματα της BGL:

- Γενικότητα και ευελιξία καθώς βασίζεται στο ισχυρότερο μοντέλο του γενικευμένου προγραμματισμού.
- Πλήρης τεκμηρίωση.
- Αποδοτικότητα (δεν χρησιμοποιούνται π.χ. εικονικές μέθοδοι).

Μειονεκτήματα:

- Μικρότερο πλήθος αλγορίθμων.
- Αρχική δυσκολία στην εκμάθηση λόγω του διαφορετικού μοντέλου προγραμματισμού.
- Δυσκολία στην εύρεση σφαλμάτων λόγω χρήσης προχωρημένων τεχνικών της C++.
- Ορισμένοι αλγόριθμοι δεν έχουν την ποιότητα υλοποίησης που έχει η LEDA.

Ο ιστοχώρος του Boost Project: <http://www.boost.org>

Boost graph library book:

<http://www.informit.com/store/product.aspx?isbn=0201729148>

Τέλος Ενότητας



Ευρωπαϊκή Ένωση
Ευρωπαϊκό Κοινωνικό Ταμείο



ΕΠΙΧΕΙΡΗΣΙΑΚΟ ΠΡΟΓΡΑΜΜΑ
ΕΚΠΑΙΔΕΥΣΗ ΚΑΙ ΔΙΑ ΒΙΟΥ ΜΑΘΗΣΗ
επένδυση στην κοινωνία της γνώσης

ΥΠΟΥΡΓΕΙΟ ΠΑΙΔΕΙΑΣ & ΘΡΗΣΚΕΥΜΑΤΩΝ, ΠΟΛΙΤΙΣΜΟΥ & ΑΘΛΗΤΙΣΜΟΥ
ΕΙΔΙΚΗ ΥΠΗΡΕΣΙΑ ΔΙΑΧΕΙΡΙΣΗΣ

Με τη συγχρηματοδότηση της Ελλάδας και της Ευρωπαϊκής Ένωσης



Το παρόν έργο αποτελεί την έκδοση **1.0**.

Copyright Πανεπιστήμιο Πατρών, Χρήστος Ζαρολιάγκης, 2014. «Τεχνολογίες Υλοποίησης Αλγορίθμων». Έκδοση: 1.0. Πάτρα 2014. Διαθέσιμο από τη δικτυακή διεύθυνση:

<https://eclass.upatras.gr/courses/CEID1084>

Το παρόν υλικό διατίθεται με τους όρους της άδειας χρήσης Creative Commons Αναφορά, Μη Εμπορική Χρήση, Όχι Παράγωγα Έργα 4.0 [1] ή μεταγενέστερη, Διεθνής Έκδοση. Εξαιρούνται τα αυτοτελή έργα τρίτων π.χ. φωτογραφίες, διαγράμματα κ.λ.π., τα οποία εμπεριέχονται σε αυτό.



[1] <http://creativecommons.org/licenses/by-nc-nd/4.0>

Ως **Μη Εμπορική** ορίζεται η χρήση:

- που δεν περιλαμβάνει άμεσο ή έμμεσο οικονομικό όφελος από την χρήση του έργου, για το διανομέα του έργου και αδειοδόχο
- που δεν περιλαμβάνει οικονομική συναλλαγή ως προϋπόθεση για τη χρήση ή πρόσβαση στο έργο
- που δεν προσπορίζει στο διανομέα του έργου και αδειοδόχο έμμεσο οικονομικό όφελος (π.χ. διαφημίσεις) από την προβολή του έργου σε διαδικτυακό τόπο

Ο δικαιούχος μπορεί να παρέχει στον αδειοδόχο ξεχωριστή άδεια να χρησιμοποιεί το έργο για εμπορική χρήση, εφόσον αυτό του ζητηθεί.

Οποιαδήποτε αναπαραγωγή ή διασκευή του υλικού θα πρέπει να συμπεριλαμβάνει :

- το Σημείωμα Αναφοράς
- το Σημείωμα Αδειοδότησης
- τη δήλωση Διατήρησης Σημειωμάτων
- το Σημείωμα Χρήσης Έργων Τρίτων (εφόσον υπάρχει) μαζί με τους συνοδευόμενους υπερσυνδέσμους