Δομές Δεδομένων και Αλγόριθμοι

Γραφήματα - αλγόριθμοι γραφημάτων (V1.0)

Χρήστος Γκόγκος

Πανεπιστήμιο Ιωαννίνων, Τμήμα Πληροφορικής και Τηλεπικοινωνιών (2019-2020)

Ένας γρίφος: "Οι γέφυρες του Κένιξμπεργκ"

Στο δέκατο-όγδοο αιώνα η πόλη Königsberg είχε 7 γέφυρες που σύνδεαν διαφορετικά τμήματα της πόλης. Υπήρχε δε η διαμάχη για το εάν υπήρχε διαδρομή που διένυε κάθε γέφυρα μια μόνο φορά και περνούσε από όλες τις γέφυρες. Ο διάσημος μαθηματικός Leonard Euler απέδειξε ότι κάτι τέτοιο ήταν αδύνατο και ταυτόχρονα θεμελίωσε τη θεωρία γραφημάτων.

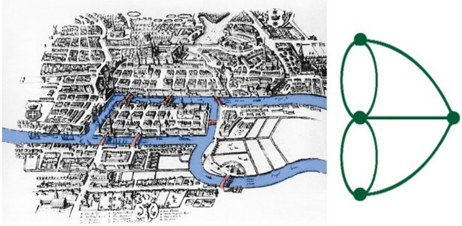


Figure 1: Königsberg

https://www.mathscareers.org.uk/article/bridges-of-konigsberg-and-graph-theory/

Γραφήματα - Γράφοι (Graphs)

Ένα γράφημα είναι ένα σύνολο από σημεία - θέσεις που ονομάζονται κορυφές (vertices) ή κόμβοι (nodes) για τα οποία ισχύει ότι κάποιες κορυφές είναι συνδεδεμένες μεταξύ τους με συνδέσμους που ονομάζονται ακμές (edges ή arcs). Συνήθως ένα γράφημα συμβολίζεται ως $G = (V, E)$ όπου V είναι το σύνολο των κορυφών και E είναι το σύνολο των ακμών.

- Ένα πλήρες γράφημα (που όλες οι κορυφές συνδέονται απευθείας με όλες τις άλλες κορυφές) έχει $\frac{|V||V-1|}{2}$ ακμές ($|V|$ είναι το πλήθος των κορυφών του γραφήματος).
- Μια ακμή αναπαρίσταται ως ένα ζεύγος (v_1, v_2) με v_1, v_2 κορυφές του γραφήματος.

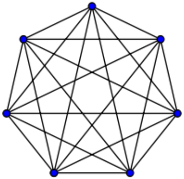


Figure 2: Πλήρες γράφημα με 7 κορυφές και 21 ακμές

Παραδείγματα γραφημάτων για προβλήματα του πραγματικού κόσμου

- Σύστημα πλοήγησης: κορυφές είναι οι διευθύνσεις των κτιρίων και οι διασταυρώσεις των δρόμων και ακμές οι δρόμοι.
- Δίκτυο πτήσεων: κορυφές είναι τα αεροδρόμια και ακμές είναι οι πτήσεις.
- Δίκτυο ύδρευσης: κορυφές είναι οι καταναλωτές και τα υδραγωγεία και ακμές είναι οι σωληνώσεις
- Κοινωνικό δίκτυο: κορυφές είναι τα άτομα που συμμετέχουν στο κοινωνικό δίκτυο και ακμές είναι οι σχέσεις φιλίας μεταξύ των ατόμων.

Είδη γραφημάτων (1/3)

- Μη κατευθυνόμενα γραφήματα (undirected graphs): Οι ακμές του γραφήματος δεν έχουν κατεύθυνση.
- Κατευθυνόμενα γραφήματα (directed graphs): Οι ακμές του γραφήματος έχουν κατεύθυνση.
- Γραφήματα με βάρη (weighted graphs): Οι ακμές έχουν βάρη (συναντώνται συχνά σε εφαρμογές του πραγματικού κόσμου π.χ. προβλήματα μεταφορών).

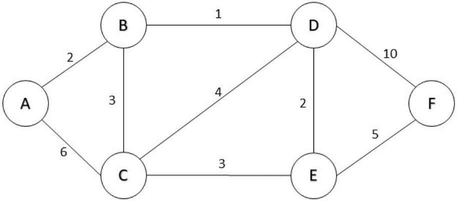


Figure 3: Μη κατευθυνόμενο γράφημα με βάρη

Είδη γραφημάτων (2/3)

- Κατευθυνόμενα ακυκλικά γραφήματα (directed acyclic graphs): Δεν υπάρχει κύκλος από μια κορυφή προς την ίδια κορυφή μέσω ακμών του γραφήματος.

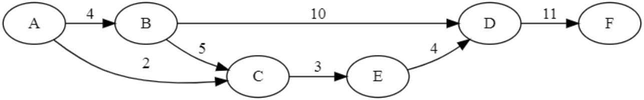


Figure 4: Κατευθυνόμενο ακυκλικό γράφημα με βάρη

Είδη γραφημάτων (3/3)

- Αραιά γραφήματα (sparse graphs): γραφήματα με μικρό αριθμό ακμών σε σχέση με τον αριθμό ακμών που θα είχε το πλήρες γράφημα.
- Πυκνά γραφήματα (dense graphs): γραφήματα με αριθμό ακμών με συγκρίσιμο μέγεθος σε σχέση με τον αριθμό ακμών που θα είχε το πλήρες γράφημα ονομάζονται πυκνά γραφήματα.

Αναπαραστήσεις γραφημάτων

- Για να αναπαρασταθεί ένα γράφημα θα πρέπει να αποθηκευτεί τόσο το σύνολο των κορυφών όσο και το σύνολο των ακμών του γραφήματος.
- Λειτουργίες που πρέπει να υποστηρίζονται:
    - Εντοπισμός όλων των ακμών μιας κορυφής.
    - Έλεγχος αν δύο κορυφές συνδέονται απευθείας.
- Η αποθήκευση των ακμών μπορεί να γίνει είτε με πίνακες γειτνίασης (adjacency matrix) είτε με λίστες γειτνίασης (adjacency list).

Πίνακες γειτνίασης

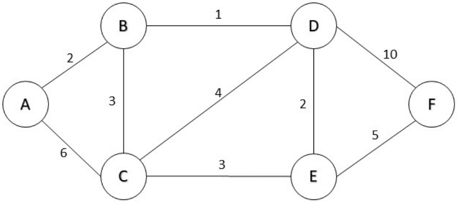
- Ένας πίνακας γειτνίασης είναι ένας $|V| \times |V|$ πίνακας A για τον οποίο ισχύει:
  - A[i,j] = 1 αν υπάρχει ακμή από την κορυφή i στην κορυφή j (αν το γράφημα είναι με βάρη τότε στη θέση της μονάδας είναι το βάρος της ακμής)
  - A[i,j] = 0 αλλιώς
- Η γραμμή i του πίνακα αναπαριστά όλες τις ακμές που εξέρχονται από την κορυφή i.
- Η στήλη j του πίνακα αναπαριστά όλες τις ακμές που καταλήγουν στην κορυφή j.
- Πρόκειται για έναν απλό τρόπο αποθήκευσης της πληροφορίας του γραφήματος:
  - Ο έλεγχος αν δύο κορυφές συνδέονται απευθείας γίνεται σε χρόνο $\mathrm{O}(1)$.
  - Ο χώρος που απαιτεί είναι $\mathrm{O}(|V|^2)$.
  - Οι πίνακες γειτνίασης δεν είναι καλή λύση για αραιά γραφήματα λόγω της σπατάλης χώρου που συνεπάγεται η αποθήκευση των μηδενικών που υποδηλώνουν ότι συγκεκριμένες κορυφές δεν συνδέονται απευθείας.

Παράδειγμα πίνακα γειτνίασης

```
#define V 6

string vertices[V] = {"A", "B", "C", "D", "E", "F"};

int adj_matrix[V][V] = {
    {0, 2, 6, 0, 0, 0},
    {2, 0, 3, 1, 0, 0},
    {6, 3, 0, 4, 3, 0},
    {0, 1, 4, 0, 2, 10},
    {0, 0, 3, 2, 0, 5},
    {0, 0, 0, 10, 5, 0}};
```



Βασικές λειτουργίες σε πίνακα γειτνίασης

- get_vertex_index(v): επιστροφή του δείκτη της κορυφής v.
- get_edges(v): επιστροφή των ακμών (βάρος, κορυφή) με τις οποίες συνδέεται απευθείας η κορυφή v.
- are_directly_connected(v1,v2): έλεγχος αν οι κορυφές v1 και v2 συνδέονται απευθείας.

```
int get_vertex_index(string v) {
    for (int i = 0; i < V; i++)
        if (vertices[i] == v)
            return i;
    return -1;
}
typedef pair<int, string> w_v;
vector<w_v> get_edges(string v) {
    vector<w_v> edges;
    int pos = get_vertex_index(v);
    for (int j = 0; j < V; j++)
        if (adj_matrix[pos][j] != 0)
            edges.push_back({adj_matrix[pos][j], vertices[j]});
    return edges;
}
bool are_directly_connected(string v1, string v2) {
    int i = get_vertex_index(v1);
    int j = get_vertex_index(v2);
    return adj_matrix[i][j] != 0;
}
```

Λίστες γειτνίασης

- Για κάθε κορυφή διατηρείται μια λίστα με τις κορυφές που εξέρχονται από αυτή.
  - Διευκολύνει το πέρασμα από όλες τις ακμές που εξέρχονται από μια κορυφή.
  - Έχει μικρές απαιτήσεις χώρου $O(|V| + |E|)$.
- Μια πιθανή υλοποίηση λίστας γειτνίασης είναι με χρήση ενός πίνακα διανυσμάτων. Για κάθε κορυφή διατηρείται ένα διάνυσμα με όλες τις κορυφές με τις οποίες η κορυφή είναι απευθείας συνδεδεμένη.
  - Η “γραμμή” i είναι μια λίστα που αναπαριστά τις ακμές που εξέρχονται από την κορυφή i.
  - Η j στη σειρά τιμή σε μια δεδομένη “γραμμή” i αντιστοιχεί στη j ακμή που εξέρχεται από την κορυφή i.

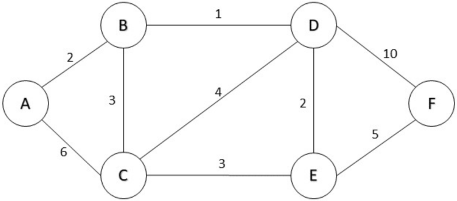
Παράδειγμα λίστας γειτνίασης

```
#define V 6

string vertices[V] = {"A", "B", "C", "D", "E", "F"};

typedef pair<int, string> w_v;

map<string, vector<w_v>> g = {
    {"A", {{2, "B"}, {6, "C"}}},
    {"B", {{2, "A"}, {3, "C"}, {1, "D"}}},
    {"C", {{6, "A"}, {3, "B"}, {4, "D"}, {3, "E"}}},
    {"D", {{1, "B"}, {4, "C"}, {2, "E"}, {10, "F"}}},
    {"E", {{3, "C"}, {2, "D"}, {5, "F"}}},
    {"F", {{10, "D"}, {5, "E"}}}};
```



Βασικές λειτουργίες σε λίστα γειτνίασης

- get_vertex_index(v): επιστροφή του δείκτη της κορυφής v.
- get_edges(v): επιστροφή των ακμών (βάρος, κορυφή) με τις οποίες συνδέεται απευθείας η κορυφή v.
- are_directly_connected(v1,v2): έλεγχος αν οι κορυφές v1 και v2 συνδέονται απευθείας.

```
int get_vertex_index(string v) {
    for (int i = 0; i < V; i++)
        if (vertices[i] == v)
            return i;
    return -1;
}

vector<w_v> get_edges(string v) {
    return g[v];
}

bool are_directly_connected(string v1, string v2) {
    for (auto p : g[v1])
        if (p.second == v2)
            return true;
    return false;
}
```

Διάσχιση γραφήματος

Οι δύο βασικοί αλγόριθμοι διάσχισης γραφημάτων είναι η αναζήτηση κατά βάθος και η αναζήτηση κατά πλάτος. Συχνά χρησιμοποιούνται ως υπορουτίνες σε άλλους αλγορίθμους.

Αναζήτηση κατά βάθος

- Η βασική ιδέα της αναζήτησης κατά βάθος (DFS=Depth First Search) είναι ότι εξετάζονται κατά προτεραιότητα οι κορυφές που είναι προσβάσιμες από την τρέχουσα κάθε φορά κορυφή, προχωρώντας βαθιά στο γράφημα πριν εξεταστούν όλες οι κορυφές που βρίσκονται σε μικρή απόσταση από την αφετηρία.
- Για την υλοποίηση της DFS χρησιμοποιείται η δομή δεδομένων στοίβα (stack) έτσι ώστε να καθοριστεί η σειρά με την οποία θα πραγματοποιηθεί η εξερεύνηση των κορυφών.
- Η χρονική πολυπλοκότητα της DFS είναι $O(|V|+|E|)$.

Αναζήτηση κατά βάθος - κωδικοποίηση

```
bool visited[V];
void clear_visited() {
    for (int i = 0; i < V; i++)
        visited[i] = false;
}
void dfs(string start) {
    stack<string> a_stack;
    a_stack.push(start);
    while (!a_stack.empty()) {
        string curr = a_stack.top();
        a_stack.pop();
        if (visited[get_vertex_index(curr)])
            continue;
        cout << curr << " ";
        visited[get_vertex_index(curr)] = true;
        for (w_v v : get_edges(curr))
            a_stack.push(v.second);
    }
    cout << endl;
}
int main() {
    clear_visited();
    dfs("A");
}
```

αποτέλεσμα εκτέλεσης

```
A C E F D B
```

Αναζήτηση κατά πλάτος

- Η βασική ιδέα της αναζήτησης κατά πλάτος (BFS=Breadth First Search) είναι ότι εξετάζονται όλες οι κορυφές που είναι απευθείας προσβάσιμες από την τρέχουσα κορυφή πριν πραγματοποιηθεί μετακίνηση στην επόμενη κορυφή.
- Για την υλοποίηση της BFS χρησιμοποιείται η δομή δεδομένων ουρά (queue) έτσι ώστε να καθοριστεί η σειρά με την οποία θα πραγματοποιηθεί η εξερεύνηση των κορυφών.
- Η χρονική πολυπλοκότητα της BFS είναι $O(|V| + |E|)$.

Αναζήτηση κατά πλάτος - κωδικοποίηση

```
bool visited[V];
void clear_visited() {
    for (int i = 0; i < V; i++)
        visited[i] = false;
}
void bfs(string start) {
    queue<string> a_queue;
    a_queue.push(start);
    while (!a_queue.empty()) {
        string curr = a_queue.front();
        a_queue.pop();
        if (visited[get_vertex_index(curr)])
            continue;
        cout << curr << " ";
        visited[get_vertex_index(curr)] = true;
        for (w_v v : get_edges(curr))
            a_queue.push(v.second);
    }
    cout << endl;
}
int main() {
    clear_visited();
    bfs("A");
}
```

αποτέλεσμα εκτέλεσης

```
A B C D E F
```

Αλγόριθμος του Dijkstra για εύρεση συντομότερης διαδρομής

- Η βασική ιδέα του αλγορίθμου του Dijkstra είναι ότι συνεχώς αναζητείται ανάμεσα στις διαθέσιμες διαδρομές που επεκτείνουν το σύνολο των κόμβων που έχουν επισκεφθεί, η διαδρομή με το μικρότερο συνολικό βάρος.
- Μια καλή υλοποίηση του αλγορίθμου του Dijkstra (με χρήση λίστας γειτνίασης και ουράς προτεραιότητας για την αποθήκευση των διαδρομών βάσει μήκους διαδρομής) έχει πολυπλοκότητα χρόνου $O(|E| \log |V|)$.
- Ο αλγόριθμος του Dijkstra δεν λειτουργεί ορθά αν τα βάρη των ακμών είναι αρνητικά.

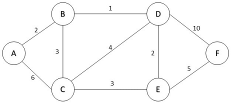
Περιγραφή του αλγορίθμου

Ο αλγόριθμος εντοπίζει τις συντομότερες διαδρομές προς τις κορυφές του γραφήματος σε σειρά απόστασης από την κορυφή αφετηρία. Σε κάθε βήμα του αλγορίθμου η αφετηρία και οι ακμές προς τις κορυφές για τις οποίες έχει ήδη βρεθεί συντομότερο μονοπάτι σχηματίζουν το υποδένδρο S του γραφήματος. Οι κορυφές που είναι προσπελάσιμες με 1 ακμή από το υποδένδρο S είναι υποψήφιες να αποτελέσουν την επόμενη κορυφή που θα εισέλθει στο υποδένδρο. Επιλέγεται μεταξύ τους η κορυφή που βρίσκεται στη μικρότερη απόσταση από την αφετηρία. Για κάθε υποψήφια κορυφή u υπολογίζεται το άθροισμα της απόστασής της από την πλησιέστερη κορυφή v του δένδρου συν το μήκος της συντομότερης διαδρομής από την αφετηρία s προς την κορυφή v. Στη συνέχεια επιλέγεται η κορυφή με το μικρότερο άθροισμα και προσαρτάται στο σύνολο των κορυφών που απαρτίζουν το υποδένδρο S. Για κάθε μία από τις υποψήφιες κορυφές που συνδέονται με μια ακμή με την κορυφή που επιλέχθηκε ενημερώνεται η απόστασή της από το υποδένδρο εφόσον προκύψει μικρότερη τιμή.

Ψευδοκώδικας

Το σύνολο S περιέχει τις κορυφές για τις οποίες έχει προσδιοριστεί η συντομότερη διαδρομή από την κορυφή s (αφετηρία) ενώ το διάνυσμα d περιέχει τις αποστάσεις από την κορυφή s

1. Αρχικά $S = s$, $d_s = 0$ και για όλες τις κορυφές $i \neq s, d_i = \infty$
2. Μέχρι να γίνει $S = V$
3. Εντοπισμός του στοιχείου $v \notin S$ με τη μικρότερη τιμή d_v και προσθήκη του στο S
4. Για κάθε ακμή από την κορυφή v στην κορυφή u με βάρος w ενημερώνεται η τιμή d_u έτσι ώστε:
$$d_u = min(d_u, d_v + w)$$
5. Επιστροφή στο βήμα 2.



$S = \{A\}, d_A = 0, d_B = 2, d_C = 6, d_D = \infty, d_E = \infty, d_F = \infty$	Από το S μπορούμε να φτάσουμε στις κορυφές B και C με μήκος διαδρομής 2 και 6 αντίστοιχα. Επιλέγεται η κορυφή B.
$S = \{A, B\}, d_A = 0, d_B = 2, d_C = 5, d_D = 3, d_E = \infty, d_F = \infty$	Από το S μπορούμε να φτάσουμε στις κορυφές C και D με μήκος διαδρομής 5 και 3 αντίστοιχα. Επιλέγεται η κορυφή D.
$S = \{A, B, D\}, d_A = 0, d_B = 2, d_C = 5, d_D = 3, d_E = 5, d_F = 13$	Από το S μπορούμε να φτάσουμε στις κορυφές C, E και F με μήκος διαδρομής 5, 5 και 13 αντίστοιχα. Επιλέγεται (με τυχαίο τρόπο) ανάμεσα στις κορυφές C και E η κορυφή C.
$S = \{A, B, D, C\}, d_A = 0, d_B = 2, d_C = 5, d_D = 3, d_E = 5, d_F = 13$	Από το S μπορούμε να φτάσουμε στις κορυφές E και F με μήκος διαδρομής 5 και 13 αντίστοιχα. Επιλέγεται η κορυφή E.
$S = \{A, B, D, C, E\}, d_A = 0, d_B = 2, d_C = 5, d_D = 3, d_E = 5, d_F = 10$	Η μοναδική κορυφή στην οποία μένει να φτάσουμε από το S είναι η κορυφή F και το μήκος της συντομότερης διαδρομής από την A στην F είναι 10.
$S = \{A, B, D, C, E, F\}, d_A = 0, d_B = 2, d_C = 5, d_D = 3, d_E = 5, d_F = 10$	

Figure 5: Αναλυτική περιγραφή εκτέλεσης αλγορίθμου Dijkstra

Κωδικοποίηση αλγορίθμου (shortest path Dijkstra)

```
struct path_info { string path; int cost; };
void dijkstra(string source) {
    map<string, path_info> paths; vector<string> S = {source}; set<string> NS;
    for (string v : vertices)
        if (v == source) paths[v] = {source, 0};
        else { NS.insert(v); paths[v] = {"", numeric_limits<int>::max()}; }
    while (!NS.empty()) {
        string v1 = S.back();
        for (w_v wv : get_edges(v1)) {
            int w = wv.first; string v2 = wv.second;
            if (NS.find(v2) != NS.end())
                if (paths[v1].cost + w < paths[v2].cost) {
                    paths[v2].path = paths[v1].path + " " + v2;
                    paths[v2].cost = paths[v1].cost + w;
                }
        }
        int min = numeric_limits<int>::max(); string pmin = "None";
        for (string v2 : NS)
            if (paths[v2].cost < min) { min = paths[v2].cost; pmin = v2; }
        if (pmin == "None") break;
        S.push_back(pmin); NS.erase(pmin);
    }
    for (auto &p : paths)
        cout << "Shortest path from vertex " << source << " to vertex " << p.first
             << " is {" << p.second.path << "} having length " << p.second.cost
             << endl;
}
```

Παράδειγμα εκτέλεσης (shortest path Dijkstra)

```
int main()
{
    cout << "source = A" << endl;
    dijkstra("A");
    cout << "source = F" << endl;
    dijkstra("F");
}
```

αποτελέσματα

```
source = A
Shortest path from vertex A to vertex A is {A} having length 0
Shortest path from vertex A to vertex B is {A B} having length 2
Shortest path from vertex A to vertex C is {A B C} having length 5
Shortest path from vertex A to vertex D is {A B D} having length 3
Shortest path from vertex A to vertex E is {A B D E} having length 5
Shortest path from vertex A to vertex F is {A B D E F} having length 10
source = F
Shortest path from vertex F to vertex A is {F E D B A} having length 10
Shortest path from vertex F to vertex B is {F E D B} having length 8
Shortest path from vertex F to vertex C is {F E C} having length 8
Shortest path from vertex F to vertex D is {F E D} having length 7
Shortest path from vertex F to vertex E is {F E} having length 5
Shortest path from vertex F to vertex F is {F} having length 0
```

Η τοπολογική ταξινόμηση (Topological Sort) εφαρμόζεται σε DAGs και παράγει μια σειρά κορυφών του γραφήματος για την οποία ισχύει ότι για κάθε κατευθυνόμενη ακμή από την κορυφή u στην κορυφή v στη σειρά των κορυφών η κορυφή u προηγείται της κορυφής v.

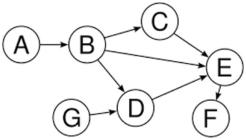


Figure 6: Ακυκλικό κατευθυνόμενο γράφημα (DAG)

```
#define V 7
string vertices[V] = {"A", "B", "C", "D", "E", "F", "G"};
int adj_matrix[V][V] = {
    {0, 1, 0, 0, 0, 0, 0},
    {0, 0, 1, 1, 1, 0, 0},
    {0, 0, 0, 0, 1, 0, 0},
    {0, 0, 0, 0, 1, 0, 0},
    {0, 0, 0, 0, 0, 1, 0},
    {0, 0, 0, 0, 0, 0, 0},
    {0, 0, 0, 1, 0, 0, 0}};
```

Τοπολογική ταξινόμηση με τον αλγόριθμο του Kahn

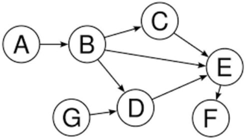
Ο αλγόριθμος επιλέγει την κορυφή που δεν έχει εισερχόμενες ακμές. Αν υπάρχουν περισσότερες από μια τότε επιλέγεται κάποια από αυτές με τυχαίο τρόπο. Στη συνέχεια αφαιρείται η κορυφή από το γράφημα καθώς και οι ακμές που εξέρχονται από αυτή και καταλήγουν σε άλλες κορυφές. Η διαδικασία επαναλαμβάνεται μέχρι να εξαντληθούν όλες οι κορυφές.

Τοπολογική ταξινόμηση με τον αλγόριθμο του Kahn (κωδικοποίηση)

```
void topological_sort() {
    vector<int> in_degree(V, 0);
    for (int j = 0; j < V; j++)
        for (int i = 0; i < V; i++)
            if (adj_matrix[i][j]) in_degree[j]++;
    queue<int> q;
    for (int i = 0; i < V; i++)
        if (in_degree[i] == 0) q.push(i);
    int visited = 0; vector<int> top_order;
    while (!q.empty()) {
        int u = q.front(); q.pop(); top_order.push_back(u);
        for (int j = 0; j < V; j++)
            if (adj_matrix[u][j] == 1) {
                in_degree[j]--;
                if (in_degree[j] == 0) q.push(j);
            }
        visited++;
    }
    if (visited != V) { cerr << "Cycles exist!!!" << endl; return; }
    for (int i = 0; i < top_order.size(); i++)
        cout << vertices[top_order[i]] << " ";
    cout << endl;
}
```

Τοπολογική ταξινόμηση με τον αλγόριθμο του Kahn (εκτέλεση κώδικα)

Κάθε DAG έχει τουλάχιστον μια τοπολογική σειρά για τις κορυφές του.




```
int main() {
    topological_sort();
}
```

αποτελέσματα

```
A G B C D E F
```