17. WEIGHT GRAPHS

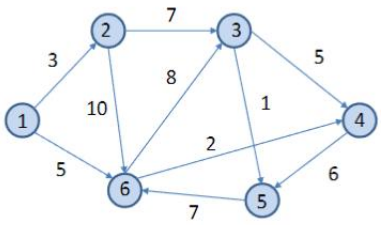
Σταθμισμένα Γραφήματα

- Τα σταθμισμένα γραφήματα μπορεί να είναι κατευθυνόμενα ή μη κατευθυνόμενα.
- Σε αυτά, κάθε ακμή συνοδεύεται από έναν αριθμό που ονομάζεται **βάρος**.
- **Παράδειγμα**: Αν οι κορυφές του γραφήματος αντιστοιχούν σε πόλεις σε έναν χάρτη, το βάρος μιας ακμής μπορεί να είναι:
  - η απόσταση μεταξύ δύο πόλεων,
  - το κόστος ενός εισιτηρίου για τη διαδρομή,
  - ή ο χρόνος που απαιτείται για το ταξίδι.

Αναπαράσταση Σταθμισμένων Γραφημάτων

- Ένας τρόπος αναπαράστασης ενός σταθμισμένου γραφήματος G είναι μέσω ενός **πίνακα γειτνίασης** (adjacency matrix) T, όπου:
  - $T[i, j] = wij$ αν υπάρχει ακμή από την κορυφή vi στην κορυφή vj με βάρος wij,
  - $T[i, i] = 0$ (το βάρος από μια κορυφή στον εαυτό της είναι 0),
  - $T[i, j] = \infty$ αν δεν υπάρχει ακμή από το vi στο vj.
- Υποθέτουμε ότι όλα τα βάρη είναι **μη αρνητικοί αριθμοί**.

Example Weighted Directed Graph



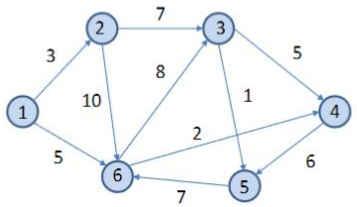
Adjacency Matrix for the Example Graph

	1	2	3	4	5	6
1	0	3	∞	∞	∞	5
2	∞	0	7	∞	∞	10
3	∞	∞	0	5	1	∞
4	∞	∞	∞	0	6	∞
5	∞	∞	∞	∞	0	7
6	∞	∞	8	2	∞	0

Αναπαράσταση Σταθμισμένων Γραφημάτων (συνέχεια)

- Η **αναπαράσταση με λίστες γειτνίασης** μπορεί εύκολα να επεκταθεί ώστε να υποστηρίζει και σταθμισμένα γραφήματα.
- Αν υπάρχει ακμή (vi, vj) με βάρος wij, τότε η λίστα γειτνίασης της κορυφής vi θα περιλαμβάνει το ζεύγος $(\boldsymbol{vj}, \boldsymbol{wij})$.
- Με αυτόν τον τρόπο, κάθε στοιχείο της λίστας γειτνίασης δεν περιέχει μόνο την κορυφή-γειτονά αλλά και το αντίστοιχο βάρος της ακμής.

Example Weighted Directed Graph



Adjacency List Representation for the Example Graph

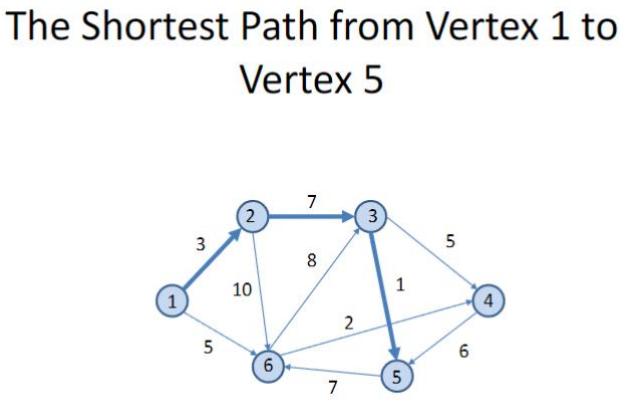
Vertices	Adjacency List
1	(2,3) (6,5)
2	(3,7) (6,10)
3	(4,5) (5,1)
4	(5,6)
5	(6,7)
6	(3,8) (4,2)

Κατευθυνόμενα Σταθμισμένα Γραφήματα

- Σε αυτό το πλαίσιο, εξετάζουμε **μόνο κατευθυνόμενα σταθμισμένα γραφήματα** (δηλαδή γραφήματα όπου οι ακμές έχουν κατεύθυνση και βάρος).

Συντομότερα Μονοπάτια

- Το **μήκος** (ή το **βάρος**) ενός μονοπατιού p είναι το **άθροισμα των βαρών των ακμών** που περιλαμβάνει το p.
- Ένα πολύ ενδιαφέρον πρόβλημα σε κατευθυνόμενο σταθμισμένο γράφημα είναι ο εντοπισμός του **συντομότερου μονοπατιού** από μια κορυφή s προς μια άλλη κορυφή t.
- Το **συντομότερο μονοπάτι** μεταξύ δύο κορυφών s και t είναι ένα **κατευθυνόμενο απλό μονοπάτι** από το s στο t, τέτοιο ώστε **κανένα άλλο μονοπάτι** να μην έχει μικρότερο συνολικό βάρος.



Πρόβλημα Συντομότερων Μονοπατιών με Κοινή Αφετηρία

- Έστω $G = (V, E)$ ένα κατευθυνόμενο σταθμισμένο γράφημα, στο οποίο κάθε ακμή έχει **μη αρνητικό βάρος**.
- Μία κορυφή του γράφου ορίζεται ως **αφετηρία** (source).
- Το **πρόβλημα των συντομότερων μονοπατιών κοινής αφετηρίας** είναι να υπολογιστεί το **μήκος του συντομότερου μονοπατιού** από την αφετηρία προς **κάθε κορυφή** του συνόλου V.

Άπληστοι Αλγόριθμοι (Greedy Algorithms)

- Οι αλγόριθμοι για **προβλήματα βελτιστοποίησης** εκτελούνται συνήθως σε **διαδοχικά βήματα**, όπου σε κάθε βήμα υπάρχουν πολλαπλές πιθανές επιλογές.
- Το πρόβλημα των συντομότερων μονοπατιών με κοινή αφετηρία είναι **πρόβλημα βελτιστοποίησης**.
- Ένας **άπληστος αλγόριθμος** επιλέγει κάθε φορά αυτό που φαίνεται **καλύτερο εκείνη τη στιγμή** (δηλαδή, την **τοπικά βέλτιστη επιλογή**), ελπίζοντας ότι αυτό θα οδηγήσει στη **συνολικά βέλτιστη λύση**.
- Οι άπληστοι αλγόριθμοι **δεν εγγυώνται πάντα τη βέλτιστη λύση**, αλλά σε πολλές περιπτώσεις αποδίδουν σωστά αποτελέσματα.

Ο Άπληστος Αλγόριθμος του Dijkstra για το Πρόβλημα Συντομότερων Μονοπατιών Κοινής Αφετηρίας

- Έστω $G = (V, E)$ το γράφημα μας.
- Ξεκινάμε με ένα σύνολο κορυφών $W = \{s\}$, το οποίο περιέχει **μόνο την αφετηρία**.
- Σταδιακά, προσθέτουμε **μία νέα κορυφή κάθε φορά** στο W, μέχρι να περιλαμβάνει **όλες τις κορυφές** του συνόλου V.
- Σε κάθε βήμα, επιλέγουμε την κορυφή $\boldsymbol{w \in V - W}$ που έχει **τη μικρότερη απόσταση από την αφετηρία**, μεταξύ όλων των κορυφών που δεν έχουν ακόμα προστεθεί στο W.
  👉 Αυτή είναι η **άπληστη επιλογή** του αλγορίθμου.

Αλγόριθμος Dijkstra (συνέχεια)

- Παρακολουθούμε τη **μικρότερη απόσταση** από την αφετηρία s προς κάθε κορυφή χρησιμοποιώντας έναν **πίνακα** `ShortestDistance[u] = Δ(u)`, που καταγράφει:
  - τη συντομότερη απόσταση από το s σε κάθε κορυφή $\boldsymbol{u \in W}$ (το σύνολο των κορυφών που έχουμε ήδη επεξεργαστεί),
  - και επίσης τη μικρότερη γνωστή απόσταση από s προς κάθε $\boldsymbol{u \in V - W}$, ακολουθώντας ένα μονοπάτι που ξεκινά από s και περνά μόνο από κορυφές του W (εκτός από την τελευταία κορυφή u που δεν ανήκει ακόμη στο W).

Ενημέρωση Αποστάσεων (Edge Relaxation)

- Κάθε φορά που προσθέτουμε μια νέα κορυφή $\boldsymbol{w}$ στο W, ενημερώνουμε τον πίνακα `ShortestDistance[u]` για κάθε κορυφή $\boldsymbol{u \in V - W}$.
- Ελέγχουμε αν η τρέχουσα καταγεγραμμένη απόσταση για το u είναι μεγαλύτερη από την απόσταση μέσω της νέας κορυφής w:

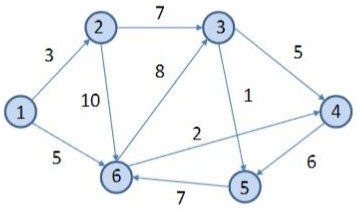
$$ShortestDistance[u] > ShortestDistance[w] + T[w, u]$$

- Αν ισχύει, τότε **ενημερώνουμε** την απόσταση:

$$ShortestDistance[u] = ShortestDistance[w] + T[w, u]$$

- Αυτή η διαδικασία ονομάζεται **χαλάρωση ακμής** (*edge relaxation*).
- Αν και λέγεται "χαλάρωση", στην πραγματικότητα *"***σφίγγουμε" το όριο** της απόστασης, βελτιώνοντας τη γνωστή τιμή.

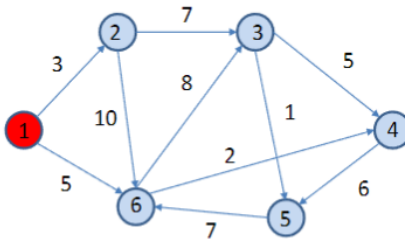
Example Graph



- We will show how Dijkstra's algorithm works on this graph with source vertex 1.

Expanding the Vertex Set W in Stages

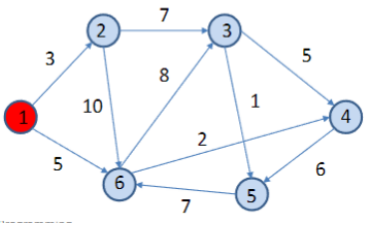
Stage	W	V-W	w	Δ(w)	Δ(1)	Δ(2)	Δ(3)	Δ(4)	Δ(5)	Δ(6)
Start	{1}	{2,3,4,5,6}	-	-	0	3	∞	∞	∞	5



Expanding the Vertex Set W in Stages (cont'd)

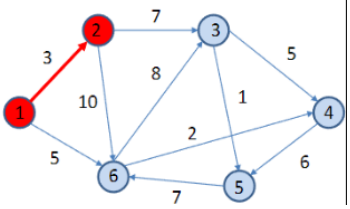
Stage	W	V-W	w	Δ(w)	Δ(1)	Δ(2)	Δ(3)	Δ(4)	Δ(5)	Δ(6)
Start	{1}	{2,3,4,5,6}	-	-	0	3	∞	∞	∞	5

w=2 is chosen for the second stage.



Expanding the Vertex Set W in Stages (cont'd)

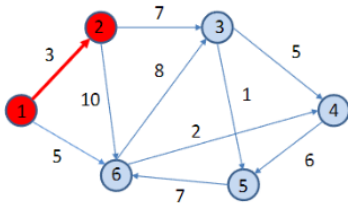
Stage	W	V-W	w	Δ(w)	Δ(1)	Δ(2)	Δ(3)	Δ(4)	Δ(5)	Δ(6)
Start	{1}	{2,3,4,5,6}	-	-	0	3	∞	∞	∞	5
2	{1,2}	{3,4,5,6}	2	3	0	3	10	∞	∞	5



Expanding the Vertex Set W in Stages (cont'd)

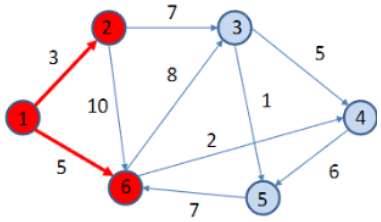
Stage	W	V-W	w	Δ(w)	Δ(1)	Δ(2)	Δ(3)	Δ(4)	Δ(5)	Δ(6)
Start	{1}	{2,3,4,5,6}	-	-	0	3	∞	∞	∞	5
2	{1,2}	{3,4,5,6}	2	3	0	3	10	∞	∞	5

w=6 is chosen for the third stage.



Expanding the Vertex Set W in Stages (cont'd)

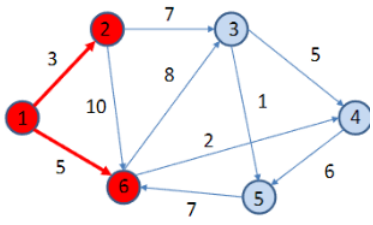
Stage	W	V-W	w	Δ(w)	Δ(1)	Δ(2)	Δ(3)	Δ(4)	Δ(5)	Δ(6)
Start	{1}	{2,3,4,5,6}	-	-	0	3	∞	∞	∞	5
2	{1,2}	{3,4,5,6}	2	3	0	3	10	∞	∞	5
3	{1,2,6}	{3,4,5}	6	5	0	3	10	7	∞	5



Expanding the Vertex Set W in Stages (cont'd)

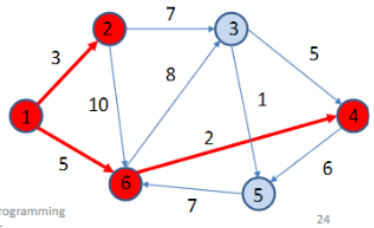
Stage	W	V-W	w	Δ(w)	Δ(1)	Δ(2)	Δ(3)	Δ(4)	Δ(5)	Δ(6)
Start	{1}	{2,3,4,5,6}	-	-	0	3	∞	∞	∞	5
2	{1,2}	{3,4,5,6}	2	3	0	3	10	∞	∞	5
3	{1,2,6}	{3,4,5}	6	5	0	3	10	7	∞	5

w=4 is chosen for the fourth stage.



Expanding the Vertex Set W in Stages (cont'd)

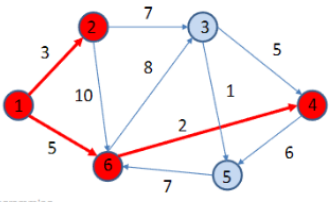
Stage	W	V-W	w	Δ(w)	Δ(1)	Δ(2)	Δ(3)	Δ(4)	Δ(5)	Δ(6)
Start	{1}	{2,3,4,5,6}	-	-	0	3	∞	∞	∞	5
2	{1,2}	{3,4,5,6}	2	3	0	3	10	∞	∞	5
3	{1,2,6}	{3,4,5}	6	5	0	3	10	7	∞	5
4	{1,2,6,4}	{3,5}	4	7	0	3	10	7	13	5



Expanding the Vertex Set W in Stages (cont'd)

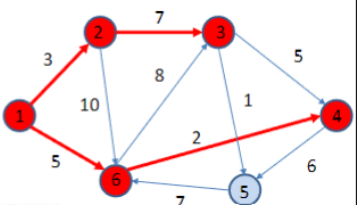
Stage	W	V-W	w	Δ(w)	Δ(1)	Δ(2)	Δ(3)	Δ(4)	Δ(5)	Δ(6)
Start	{1}	{2,3,4,5,6}	-	-	0	3	∞	∞	∞	5
2	{1,2}	{3,4,5,6}	2	3	0	3	10	∞	∞	5
3	{1,2,6}	{3,4,5}	6	5	0	3	10	7	∞	5
4	{1,2,6,4}	{3,5}	4	7	0	3	10	7	13	5

w=3 is chosen for the fifth stage.



Expanding the Vertex Set W in Stages (cont'd)

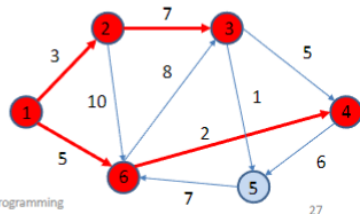
Stage	W	V-W	w	Δ(w)	Δ(1)	Δ(2)	Δ(3)	Δ(4)	Δ(5)	Δ(6)
Start	{1}	{2,3,4,5,6}	-	-	0	3	∞	∞	∞	5
2	{1,2}	{3,4,5,6}	2	3	0	3	10	∞	∞	5
3	{1,2,6}	{3,4,5}	6	5	0	3	10	7	∞	5
4	{1,2,6,4}	{3,5}	4	7	0	3	10	7	13	5
5	{1,2,6,4,3}	{5}	3	10	0	3	10	7	11	5



Expanding the Vertex Set W in Stages (cont'd)

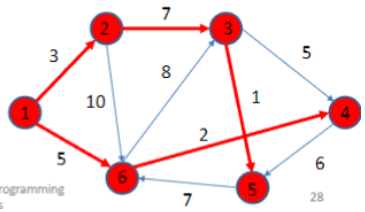
Stage	W	V-W	w	Δ(w)	Δ(1)	Δ(2)	Δ(3)	Δ(4)	Δ(5)	Δ(6)
Start	{1}	{2,3,4,5,6}	-	-	0	3	∞	∞	∞	5
2	{1,2}	{3,4,5,6}	2	3	0	3	10	∞	∞	5
3	{1,2,6}	{3,4,5}	6	5	0	3	10	7	∞	5
4	{1,2,6,4}	{3,5}	4	7	0	3	10	7	13	5
5	{1,2,6,4,3}	{5}	3	10	0	3	10	7	11	5

w=5 is chosen for the sixth stage.



Expanding the Vertex Set W in Stages (cont'd)

Stage	W	V-W	w	Δ(w)	Δ(1)	Δ(2)	Δ(3)	Δ(4)	Δ(5)	Δ(6)
Start	{1}	{2,3,4,5,6}	-	-	0	3	∞	∞	∞	5
2	{1,2}	{3,4,5,6}	2	3	0	3	10	∞	∞	5
3	{1,2,6}	{3,4,5}	6	5	0	3	10	7	∞	5
4	{1,2,6,4}	{3,5}	4	7	0	3	10	7	13	5
5	{1,2,6,4,3}	{5}	3	10	0	3	10	7	11	5
6	{1,2,6,4,3,5}	{}	5	11	0	3	10	7	11	5



```c
#include <stdio.h>
#include <limits.h>

#define MAX_VERTICES 100
#define INF INT_MAX

// Function to find the vertex with the minimum distance that hasn't been visited yet
int minDistance(int distance[], int visited[], int V) {
    int min = INF, minIndex = -1;

    for (int v = 0; v < V; v++) {
        if (!visited[v] && distance[v] <= min) {
            min = distance[v];
            minIndex = v;
        }
    }
    return minIndex;
}

void Dijkstra(int graph[MAX_VERTICES][MAX_VERTICES], int V, int source) {
    int distance[MAX_VERTICES];   // Shortest distances from source
    int visited[MAX_VERTICES];    // Visited vertices
    int W[MAX_VERTICES];          // Set of visited vertices (W)

    // Step 1: Initialize distances and visited array
    for (int i = 0; i < V; i++) {
        distance[i] = INF;
        visited[i] = 0;
    }
    distance[source] = 0;

    // Step 2: Add the source vertex to W
    W[0] = source;
    visited[source] = 1;

    // Step 3: Start the loop to find the shortest paths
    for (int count = 0; count < V - 1; count++) {
        // Step 4: Find the vertex with the smallest distance that hasn't been visited yet
        int u = minDistance(distance, visited, V);

        // Mark the vertex u as visited
        visited[u] = 1;

        // Step 5: Update the shortest distance for all unvisited neighbors of u
        for (int v = 0; v < V; v++) {
            if (!visited[v] && graph[u][v] != INF && distance[u] != INF && distance[u] + graph[u][v] < distance[v]) {
                distance[v] = distance[u] + graph[u][v];
            }
        }
    }

    // Step 6: Output the shortest distances from the source
    printf("Shortest distances from vertex %d:\n", source);
    for (int i = 0; i < V; i++) {
        if (distance[i] == INF) {
            printf("Vertex %d is unreachable\n", i);
        } else {
            printf("Distance to vertex %d is %d\n", i, distance[i]);
        }
    }
}
```

```c
int main() {
    int V, E, u, v, weight, source;

    // Input the number of vertices and edges
    printf("Enter the number of vertices: ");
    scanf("%d", &V);
    printf("Enter the number of edges: ");
    scanf("%d", &E);

    int graph[MAX_VERTICES][MAX_VERTICES];

    // Initialize the graph with INF
    for (int i = 0; i < V; i++) {
        for (int j = 0; j < V; j++) {
            graph[i][j] = INF;
        }
    }

    // Input the edges and their weights
    printf("Enter the edges (u, v, weight):\n");
    for (int i = 0; i < E; i++) {
        scanf("%d %d %d", &u, &v, &weight);
        graph[u][v] = weight;
        graph[v][u] = weight; // For undirected graphs, remove this line for directed graphs
    }

    // Input the source vertex
    printf("Enter the source vertex: ");
    scanf("%d", &source);

    // Call Dijkstra's algorithm to find the shortest path
    Dijkstra(graph, V, source);

    return 0;
}
```

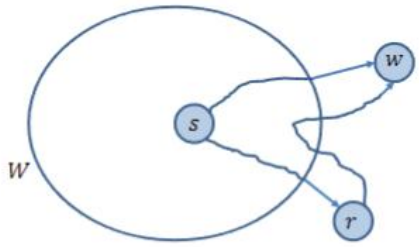
Απόδειξη Ορθότητας του Αλγορίθμου Dijkstra

- Θα αποδείξουμε ότι **σε κάθε στάδιο του αλγορίθμου**, όταν επιλέγουμε την κορυφή w, η τιμή `ShortestDistance[w]` είναι πράγματι το **μήκος του συντομότερου μονοπατιού** από την αφετηρία s προς την κορυφή w.

Απόδειξη (συνέχεια)

- Έστω ότι αυτό **δεν ισχύει** — δηλαδή, ότι `ShortestDistance[w]` **δεν είναι** το μήκος του συντομότερου μονοπατιού από το s στο w.
- Τότε, πρέπει να υπάρχει κάποιο **συντομότερο μονοπάτι** p από το s προς το w, το οποίο **περνάει από κάποια κορυφή** του συνόλου $V - W$, διαφορετική από το w.
- Ξεκινώντας από την αφετηρία s, μπορούμε να ακολουθήσουμε το μονοπάτι p περνώντας από κορυφές που ανήκουν στο W, μέχρι να συναντήσουμε την **πρώτη κορυφή** r που **δεν ανήκει** στο W.
- Σε αυτό το σημείο, όμως, σύμφωνα με την επιλογή του αλγορίθμου, **το w έχει ήδη μικρότερη ή ίση απόσταση** από την αφετηρία από κάθε τέτοια κορυφή $r \in V - W$.
- Άρα, η υπόθεση ότι υπάρχει συντομότερο μονοπάτι δεν μπορεί να ισχύει — οδηγούμαστε σε **αντίφαση**.
- Συνεπώς, το `ShortestDistance[w]` **είναι πράγματι η ελάχιστη δυνατή απόσταση** από το s προς το w, κάθε φορά που το w επιλέγεται.

Hypothetical Shorter Path to w



Απόδειξη Ορθότητας του Αλγορίθμου Dijkstra (Συνέχεια)

- Τώρα παρατηρούμε ότι το **μήκος της αρχικής τμήματος του μονοπατιού p** από το s προς το r είναι **μικρότερο** από το μήκος του **συνολικού μονοπατιού p** από το s προς το w.
- Δεδομένου ότι υποθέσαμε ότι το **μήκος του μονοπατιού p** είναι μικρότερο από το `ShortestDistance[w]`, το **μήκος του μονοπατιού από το s προς το r** είναι επίσης μικρότερο από `ShortestDistance[w]`.
- Επιπλέον, το μονοπάτι από το s προς το r έχει **όλες τις κορυφές του** εκτός από την κορυφή r να ανήκουν στο σύνολο W.
- Έτσι, θα έπρεπε να ισχύει ότι:

$$ShortestDistance[r] < ShortestDistance[w]$$

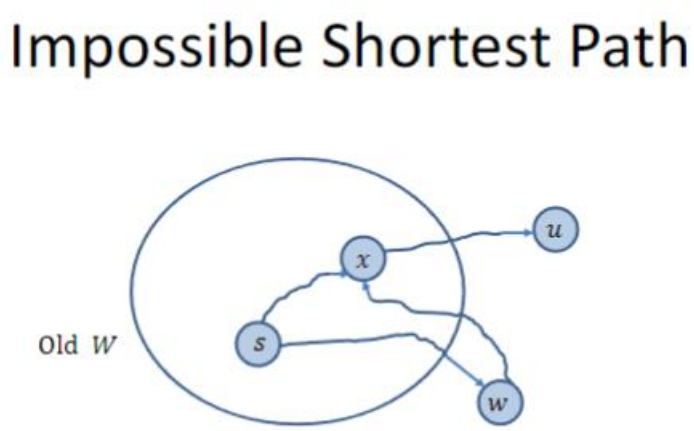
  όταν το w επιλέγεται για να προστεθεί στο σύνολο W.
- Αλλά αυτό αντιφάσκει με την επιλογή του w, καθώς θα έπρεπε να έχουμε επιλέξει το r αντί για το w.
- Επομένως, η υπόθεση είναι λάθος, και συνεπώς `ShortestDistance[w]` **είναι το μήκος του συντομότερου μονοπατιού από το s στο w**.

Απόδειξη Ορθότητας του Αλγορίθμου Dijkstra (Συνέχεια)

- Τώρα θα αποδείξουμε ότι **σε κάθε στάδιο του αλγορίθμου**, μετά την προσθήκη του νέου συνόλου W με την προσθήκη της κορυφής w και την ενημέρωση των συντομότερων αποστάσεων, η τιμή `ShortestDistance[u]` δίνει την απόσταση του συντομότερου μονοπατιού από το s προς κάθε κορυφή u στο σύνολο $V - W$ μέσω ενδιάμεσων κορυφών που **ανήκουν πλήρως στο W**.

Ανανέωση των Αποστάσεων

- Παρατηρούμε ότι όταν προσθέτουμε μια νέα κορυφή w στο σύνολο W, προσαρμόζουμε τις συντομότερες αποστάσεις για να λάβουμε υπόψη τη δυνατότητα ότι τώρα υπάρχει **ένα συντομότερο μονοπάτι προς το u** που περνάει από το w.
- Αν το μονοπάτι περνάει από το παλιό W προς το w και στη συνέχεια κατευθύνεται αμέσως προς το u, το μήκος του μονοπατιού θα συγκριθεί με το `ShortestDistance[u]`, και αν είναι μικρότερο, το `ShortestDistance[u]` θα μειωθεί.
- Η μόνη άλλη περίπτωση για ένα συντομότερο μονοπάτι είναι αν το μονοπάτι πηγαίνει από το s προς το w, επιστρέφει στο παλιό W, φτάνει σε κάποια κορυφή x του παλιού W, και τελικά καταλήγει στο u.



Απόδειξη (Συνέχεια)

- Ωστόσο, **δεν μπορεί να υπάρχει τέτοιο μονοπάτι**. Δεδομένου ότι το x τοποθετήθηκε στο σύνολο $\boldsymbol{W}$ πριν από το w, το συντομότερο από όλα τα μονοπάτια από την **αφετηρία προς το $\boldsymbol{x}$** περνάει μόνο από το παλιό σύνολο $\boldsymbol{W}$.
- Επομένως, το μονοπάτι προς το x μέσω του w, όπως φαίνεται στην εικόνα, **δεν είναι συντομότερο** από το μονοπάτι που πηγαίνει απευθείας προς το x μέσω του W.
- Ως αποτέλεσμα, το μήκος του μονοπατιού από την αφετηρία προς το w, το x και το u **δεν μειώνεται** σε σχέση με την προηγούμενη τιμή του `ShortestDistance[u]`.
- Έτσι, το `ShortestDistance[u]` **δεν μπορεί να μειωθεί από τον αλγόριθμο λόγω ενός μονοπατιού μέσω του $\boldsymbol{w}$ και του $\boldsymbol{x}$**, και συνεπώς δεν χρειάζεται να εξετάσουμε το μήκος αυτών των μονοπατιών.

Χρόνος Εκτέλεσης

- Εάν χρησιμοποιούμε **μήτρα γειτνίασης** για την αναπαράσταση του κατευθυνόμενου γράφου, ο αλγόριθμος Dijkstra εκτελείται σε $\boldsymbol{O(n^2)}$ χρόνο, όπου n είναι ο αριθμός των κορυφών του γράφου.
  - Στο στάδιο αρχικοποίησης, ο αλγόριθμος διατρέχει $\boldsymbol{n}$ − 1 κορυφές και απαιτεί χρόνο $\boldsymbol{O(n)}$.
  - Η επανάληψη **while-loop** διατρέχει τις $\boldsymbol{n}$ − 1 κορυφές του $\boldsymbol{V - \{s\}}$ μία κάθε φορά. Για κάθε τέτοια κορυφή, η επιλογή της νέας κορυφής με την ελάχιστη απόσταση, καθώς και η ενημέρωση των αποστάσεων, απαιτούν χρόνο που είναι αναλογικός με τον αριθμό των κορυφών στο $\boldsymbol{V - W}$. Επομένως, η επανάληψη απαιτεί $\boldsymbol{O(n^2)}$ χρόνο.

Χρόνος Εκτέλεσης (Συνέχεια)

- Εάν ο αριθμός των **ακμών** του γράφου e είναι πολύ μικρότερος από $\boldsymbol{n^2}$ (π.χ. $\boldsymbol{O(n)}$, δηλαδή ο γράφος είναι αραιός), είναι καλύτερο να χρησιμοποιηθεί η **αναπαράσταση με λίστα γειτνίασης** και μια **ουρά προτεραιότητας** για να οργανώσουμε τις κορυφές στο $\boldsymbol{V - W}$ σύμφωνα με τις τιμές του πίνακα `ShortestDistance`.
  - Στη συνέχεια, η ενημέρωση του πίνακα `ShortestDistance` μπορεί να γίνει περνώντας από τη λίστα γειτνίασης του w και ενημερώνοντας τις αποστάσεις στην ουρά προτεραιότητας. Συνολικά, θα πραγματοποιηθούν $\boldsymbol{e}$ ενημερώσεις, κάθε μία με κόστος $\boldsymbol{O(\log n)}$ αν η ουρά προτεραιότητας υλοποιείται ως **min heap**, άρα ο συνολικός χρόνος για τις ενημερώσεις είναι $\boldsymbol{O(e \log n)}$.

Χρόνος Εκτέλεσης (Συνέχεια)

- Ο χρόνος για την αρχικοποίηση της ουράς προτεραιότητας είναι $\boldsymbol{O(n)}$.
- Ο χρόνος που απαιτείται για την επιλογή του w είναι $\boldsymbol{O(\log n)}$, καθώς περιλαμβάνει την εύρεση και την αφαίρεση του ελάχιστου στοιχείου σε ένα σωρό.
- Επομένως, ο συνολικός χρόνος του αλγορίθμου είναι $\boldsymbol{O(n + e \log n)}$, ο οποίος είναι **πολύ καλύτερος από το $\boldsymbol{O(n^2)}$** για αραιούς γράφους.

The All-Pairs Shortest Path Problem (Συνέχεια)

- Αν έχουμε έναν κατευθυνόμενο, βαρυμένο γράφο που αντιπροσωπεύει το χρόνο πτήσης σε διάφορες διαδρομές μεταξύ πόλεων, και θέλουμε να κατασκευάσουμε έναν πίνακα που να δίνει τον συντομότερο χρόνο πτήσης από οποιαδήποτε πόλη σε οποιαδήποτε άλλη, τότε αυτό αποτελεί παράδειγμα του **προβλήματος των συντομότερων μονοπατιών για όλα τα ζεύγη**.

Τύπος του Προβλήματος

- Περισσότερο τυπικά, έστω ότι έχουμε έναν κατευθυνόμενο γράφο $G = (V, E)$, όπου κάθε ακμή (v, w) έχει μη αρνητικό βάρος $\boldsymbol{C[v, w]}$.
- Το πρόβλημα των συντομότερων μονοπατιών για όλα τα ζεύγη είναι να βρούμε, για κάθε ζεύγος κορυφών v και w, το συντομότερο μονοπάτι από την v στην w.

Προσέγγιση με τον Αλγόριθμο Dijkstra

- Ένας τρόπος για να λύσουμε το πρόβλημα είναι να τρέξουμε τον αλγόριθμο Dijkstra για κάθε κορυφή ξεχωριστά ως **αφετηρία**.
- Ωστόσο, η μέθοδος αυτή μπορεί να μην είναι η πιο αποτελεσματική για όλα τα ζεύγη κορυφών, γι' αυτό θα παρουσιάσουμε μια πιο άμεση λύση με τον αλγόριθμο **Floyd** (R.W. Floyd).

Αλγόριθμος Floyd

- Έστω ότι οι κορυφές του γράφου G είναι αριθμημένες από 0, 1, 2, ..., $n-1$.
- Ο αλγόριθμος χρησιμοποιεί έναν πίνακα $\boldsymbol{n \times n}$ $\boldsymbol{A}$ για να υπολογίσει τα μήκη των συντομότερων μονοπατιών.
  - Αρχικά, θέτουμε $\boldsymbol{A[i, j] = C[i, j]}$, όπου $\boldsymbol{C}$ είναι η μήτρα γειτνίασης του γράφου G.
  - Έτσι, αν δεν υπάρχει ακμή από το i στο j, τότε $\boldsymbol{A[i, j] = \infty}$.
  - Επίσης, κάθε διαγώνιο στοιχείο του πίνακα $\boldsymbol{A}$ είναι 0 (δηλαδή, $\boldsymbol{A[i, i] = 0}$).

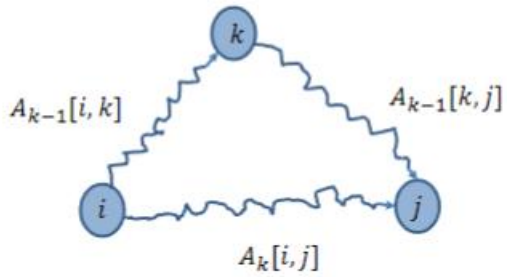
Αλγόριθμος Floyd (Συνέχεια)

- Ο αλγόριθμος εκτελεί $\boldsymbol{n}$ επαναλήψεις πάνω από τον πίνακα $\boldsymbol{A}$.
  - Μετά την $\boldsymbol{k}$-**οστή** επανάληψη, το στοιχείο $\boldsymbol{A[i, j]}$ θα περιέχει τη μικρότερη απόσταση από την κορυφή i στην κορυφή j που **δεν περνά από καμία κορυφή μεγαλύτερη από την** $\boldsymbol{k}$.
  - Στην $\boldsymbol{k}$-**οστή** επανάληψη, χρησιμοποιούμε τον εξής τύπο για τον υπολογισμό των στοιχείων του πίνακα $\boldsymbol{A}$:

$$A_k[i, j] = \min\left(A_{k-1}[i, j], A_{k-1}[i, k] + A_{k-1}[k, j]\right)$$

  - Στην ουσία, για κάθε ζεύγος κορυφών $\boldsymbol{i, j}$, εξετάζουμε αν υπάρχει μικρότερη απόσταση περνώντας από την **κορυφή** $\boldsymbol{k}$ ή αν η καλύτερη διαδρομή είναι η υπάρχουσα.

The k-th Iteration Graphically




```
void APSP(void)
{
    int i, j, k;
    int A[MAX][MAX], C[MAX][MAX];

    // Αντιγραφή της μήτρας C στην A
    for (i = 0; i <= MAX - 1; i++)
        for (j = 0; j <= MAX - 1; j++)
            A[i][j] = C[i][j];

    // Εκτέλεση του αλγορίθμου Floyd-Warshall για να βρούμε το APSP
    for (k = 0; k <= MAX - 1; k++)
        for (i = 0; i <= MAX - 1; i++)
            for (j = 0; j <= MAX - 1; j++)
                if (A[i][k] + A[k][j] < A[i][j])
                    A[i][j] = A[i][k] + A[k][j];
}
```

- **Αρχικοποίηση**: Η μήτρα **C** αναπαριστά τις άμεσες αποστάσεις μεταξύ των κορυφών. Η μήτρα **A** αρχικοποιείται να είναι ίδια με την **C**. Στη συνέχεια, εκτελείται ο αλγόριθμος Floyd-Warshall σε τρία ενσωματωμένα βρόχους για να βρεθούν οι πιο σύντομες αποστάσεις μεταξύ όλων των ζευγών κορυφών.
- **Κύριο Μέρος του Αλγορίθμου**: Για κάθε ενδιάμεση κορυφή **k**, ο αλγόριθμος ελέγχει αν η άμεση διαδρομή μεταξύ των κορυφών **i** και **j** είναι μικρότερη από τη διαδρομή που περνά από την κορυφή **k**. Αν είναι, ενημερώνει την απόσταση στη μήτρα **A**.

Χρονική Πολυπλοκότητα

- **Χρονική Πολυπλοκότητα του Αλγορίθμου Floyd**: Ο χρόνος εκτέλεσης του αλγορίθμου Floyd είναι $O(n^3)$, όπου n είναι ο αριθμός των κορυφών στο γράφημα. Αυτό συμβαίνει επειδή υπάρχουν τρεις ενσωματωμένοι βρόχοι που επαναλαμβάνονται για τον αριθμό των κορυφών.

Υπαρξή Δρομολογίων (Paths)

Σε κάποια προβλήματα μπορεί να μας ενδιαφέρει απλώς να καθορίσουμε αν υπάρχει διαδρομή από την κορυφή **i** στην κορυφή **j** σε έναν κατευθυνόμενο γράφο, χωρίς να λαμβάνουμε υπόψη τα βάρη των ακμών ή εάν τα βάρη δεν υπάρχουν καθόλου.

Αυτό μπορεί να επιτευχθεί με την τροποποίηση του **αλγορίθμου Floyd** σε **αλγόριθμο Warshall**. Ο αλγόριθμος Warshall είναι πιο παλιός και αναφέρεται στη διαδικασία που καθορίζει αν υπάρχει οποιαδήποτε διαδρομή μεταξύ δύο κορυφών, ανεξαρτήτως του μήκους της διαδρομής.

Αλγόριθμος Warshall (για Υπαρξή Δρομολογίων)

- **Μήτρα Βαρών**: Υποθέτουμε ότι η μήτρα βαρών C είναι απλώς η **μήτρα γειτνίασης** του γράφου G:
  - C**[i, j]** = **1** αν υπάρχει ακμή από την κορυφή **i** στην κορυφή **j**, και **0** αλλιώς.
- **Στόχος**: Θέλουμε να υπολογίσουμε τη μήτρα A, όπου:
  - A**[i, j]** = **1** αν υπάρχει διαδρομή από την κορυφή **i** στην κορυφή **j** με μήκος 1 ή μεγαλύτερο.
  - A**[i, j]** = **0** αν δεν υπάρχει τέτοια διαδρομή.

  Αυτή η μήτρα A είναι η **μεταβατική κλειστότητα** (transitive closure) της μήτρας γειτνίασης C.

Μεταβατική Κλειστότητα

Η **μεταβατική κλειστότητα** ενός γράφου μας λέει αν υπάρχει διαδρομή μεταξύ οποιωνδήποτε δύο κορυφών, χωρίς να λαμβάνουμε υπόψη το μήκος της διαδρομής. Μπορεί να υπολογιστεί με μια διαδικασία παρόμοια με αυτή του αλγορίθμου Floyd:

Αλγόριθμος για τη Μεταβατική Κλειστότητα:

Για κάθε **k** από 0 έως **n-1**, ενημερώνουμε τη μήτρα A χρησιμοποιώντας τον παρακάτω κανόνα:

$$A_k[i, j] = A_{k-1}[i, j] \text{ ή } (A_{k-1}[i, k] \text{ και } A_{k-1}[k, j])$$

Εξήγηση:

- **Αν** υπάρχει ήδη διαδρομή από την κορυφή **i** στην κορυφή **j** (`A[i, j] = 1`), δεν απαιτείται καμία ενημέρωση.
- **Αλλιώς**, ελέγχουμε αν υπάρχει διαδρομή από την κορυφή **i** στην κορυφή **k** και από την κορυφή **k** στην κορυφή **j** (`A[i, k] = 1` και `A[k, j] = 1`). Αν ναι, ορίζουμε `A[i, j] = 1`.

Η ενημερωμένη μήτρα **A** αντιπροσωπεύει τη **μεταβατική κλειστότητα** της μήτρας γειτνίασης, δείχνοντας πού υπάρχουν διαδρομές μεταξύ κορυφών.

Περίληψη του Αλγορίθμου Warshall:

- **Στόχος**: Να καθορίσουμε αν υπάρχει διαδρομή μεταξύ ζευγαριών κορυφών.
- **Χρονική Πολυπλοκότητα**: Ο αλγόριθμος εκτελείται σε $O(n^3)$ χρόνο, όπως και ο αλγόριθμος Floyd, καθώς χρησιμοποιεί τρεις ενσωματωμένους βρόχους για την ενημέρωση της μήτρας γειτνίασης.

```c
void TransitiveClosure(void)
{
    int i, j, k;
    int A[MAX][MAX], C[MAX][MAX];

    // Αντιγραφή της μήτρας C στην A
    for (i = 0; i <= MAX - 1; i++)
        for (j = 0; j <= MAX - 1; j++)
            A[i][j] = C[i][j];

    // Εκτέλεση του αλγορίθμου Warshall για υπολογισμό της μεταβατικής κλειστότητας
    for (k = 0; k <= MAX - 1; k++)
        for (i = 0; i <= MAX - 1; i++)
            for (j = 0; j <= MAX - 1; j++)
                if (!A[i][j])
                    A[i][j] = A[i][k] && A[k][j];
}
```

Εξήγηση του Κώδικα:

- **Αρχικοποίηση**: Η μήτρα **C** είναι η μήτρα γειτνίασης του γράφου, όπου κάθε **C[i, j] = 1** αν υπάρχει ακμή από την κορυφή **i** στην κορυφή **j**, αλλιώς **C[i, j] = 0**.
- Η μήτρα **A** αρχικοποιείται με την τιμή της μήτρας γειτνίασης **C**.
- **Κύριος Αλγόριθμος**: Ο αλγόριθμος Warshall χρησιμοποιεί τρεις βρόχους για να υπολογίσει την **μεταβατική κλειστότητα**. Για κάθε ενδιάμεση κορυφή **k**, ελέγχει αν υπάρχει διαδρομή από την κορυφή **i** στην κορυφή **j** μέσω της κορυφής **k**. Αν υπάρχει τέτοια διαδρομή, τότε θέτει το στοιχείο **A[i, j] = 1**.

  Η γραμμή `A[i][j] = A[i][k] && A[k][j]` είναι υπεύθυνη για τον έλεγχο αν υπάρχει διαδρομή μέσω της κορυφής **k**. Αν η **A[i, k] = 1** και η **A[k, j] = 1**, τότε υπάρχει διαδρομή από **i** σε **j** μέσω του **k**.

Χρονική Πολυπλοκότητα

- **Χρονική Πολυπλοκότητα του Αλγορίθμου Warshall**: Ο αλγόριθμος εκτελείται σε $O(n^3)$ χρόνο, όπου n είναι ο αριθμός των κορυφών στο γράφημα. Αυτό συμβαίνει επειδή ο αλγόριθμος έχει τρεις ενσωματωμένους βρόχους που εξετάζουν όλα τα ζευγάρια κορυφών για κάθε ενδιάμεση κορυφή.

Αυτό καθιστά τον αλγόριθμο Warshall κατάλληλο για την υπολογισμό της μεταβατικής κλειστότητας σε μικρούς ή μεσαίου μεγέθους γράφους.