Weighted Graphs (Γράφοι με Βάρη)

Manolis Koubarakis

Weighted Graphs

- **Weighted graphs** are directed or undirected graphs in which numbers called **weights** are attached to the edges.

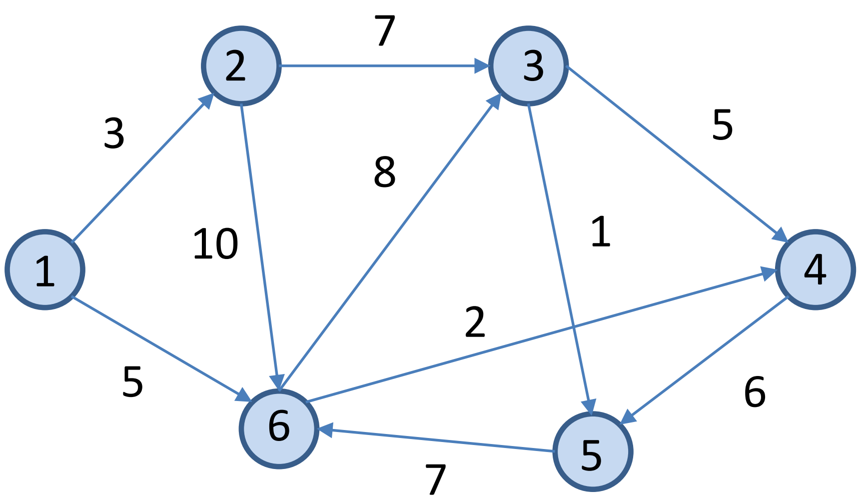
- **Example**: Let the vertices of a graph represent cities on a map. The weight on an edge connecting city A to city B can be the travel distance from A to B, the cost of an airline ticket to go from A to B, or the time required to travel from A to B.

Representations of Weighted Graphs

- To represent a weighted graph G, we can use an adjacency matrix T in which:
  - $T[i, j] = w_{ij}$ if there exists an edge $e = (v_i, v_j)$ of weight w_{ij}.
  - $T[i, i] = 0$
  - $T[i, j] = \infty$ if there is no edge from v_i to v_j .

- We will assume that all weights w_{ij} are **non-negative numbers.**

Example Weighted Directed Graph



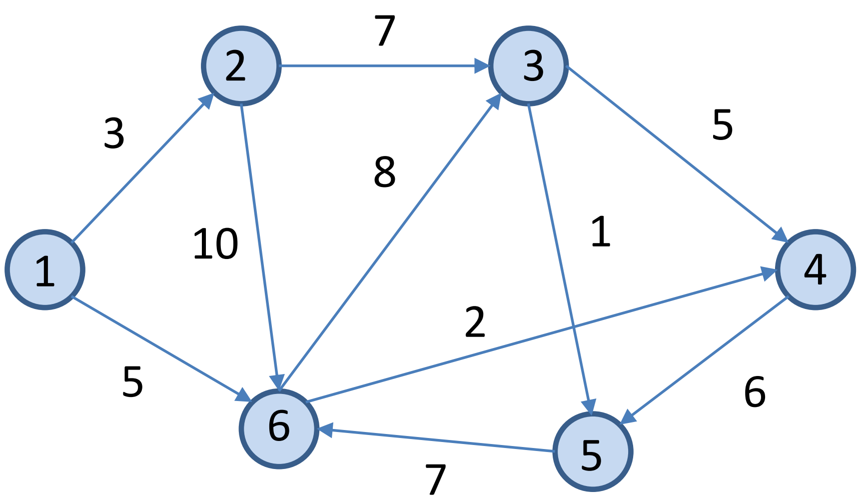
Adjacency Matrix for the Example Graph

	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

Representations of Weighted Graphs (cont'd)

- We can easily extend the adjacency list representation to be used for weighted graphs too.
- If (v_i, v_j) is an edge in the graph with weight w_{ij} then the adjacency list of v_i will contain the pair (v_j, w_{ij}).

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)

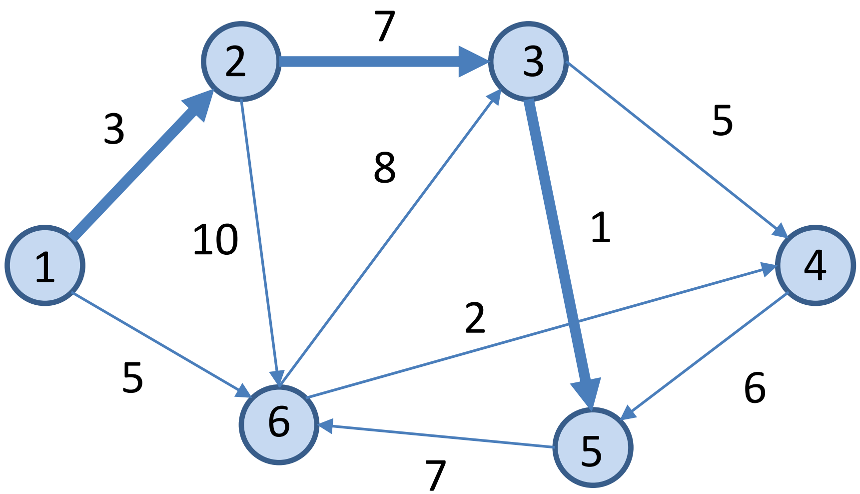
Directed Weighted Graphs

- We will consider only **directed weighted graphs** in this lecture.

Shortest Paths (Συντομότερα Μονοπάτια)

- The **length** (or **weight**) of a path p is the sum of the weights of the edges of p.
- A very interesting problem in a directed weighted graph is to find the shortest path from a vertex s to a vertex t.
- A **shortest path (συντομότερο μονοπάτι)** between to vertices s and t in a weighted directed graph is a directed simple path from s to t with the property that no other path has a lower length.

The Shortest Path from Vertex 1 to Vertex 5



The Single Source Shortest Paths Problem

- Let $G = (V, E)$ be a weighted directed graph in which each edge has a non-negative weight, and one vertex is specified as the **source (αφετηρία).**
- The **single source shortest paths** problem (το πρόβλημα των **συντομότερων μονοπατιών κοινής αφετηρίας**) is to determine the length of the shortest path from the source to each vertex in V.

Greedy Algorithms

- Algorithms for **optimization problems (προβλήματα βελτιστοποίησης)** typically go through a sequence of steps, with a set of choices at each step.
- The single source shortest path problem presented earlier is an optimization problem.
- A **greedy (άπληστος)** algorithm always makes the choice that looks best at the moment. That is, it makes a locally optimal choice in the hope that this choice will lead to a globally optimal solution.
- Greedy algorithms do not always yield optimal solutions, but for many problems they do.

Dijkstra's Greedy Algorithm for the Single Source Shortest Paths Problem

- Let $G = (V, E)$ our graph.
- We **start** with a vertex set $W = \{s\}$ containing only the source.
- We will **progressively enlarge** W by adding one new vertex at a time, until W includes all vertices of V.
- The vertex we add at each stage is the vertex w in $V - W$, which is at a **minimum distance** from the source among all vertices in $V - W$ that have not been added to W (this is a **greedy** choice).

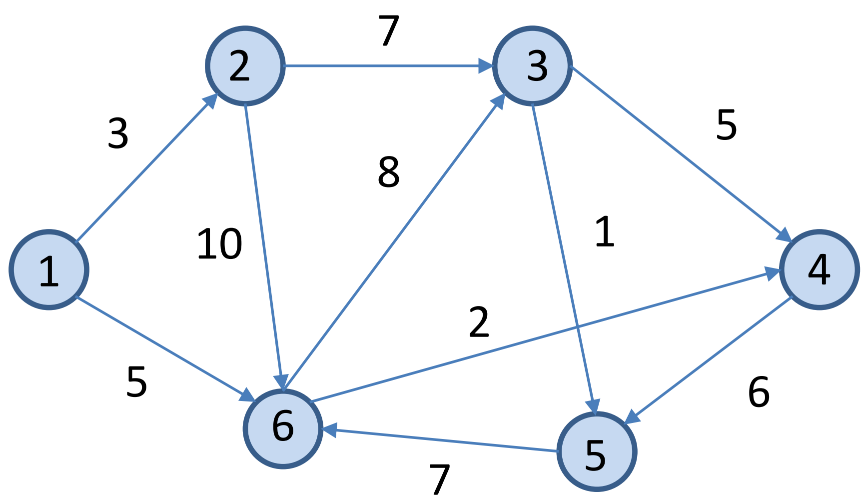
Dijkstra's Algorithm (cont'd)

- We keep track of the **minimum distance from the source** s at each stage by using an array $ShortestDistance[u] = \Delta[u]$ which keeps track of the shortest distance from s to each vertex u in W.
- It also keeps track of the shortest distance from s to each vertex u in $V - W$ using a path p starting at s, such that all vertices of path p lie in W, except the last vertex u which lies outside W.

Dijkstra’s Algorithm (cont’d)

- Every time we add a new vertex w to W, we update the array $ShortestDistance[u]$ for all u in $V - W$.
- This distance is updated in case it is bigger than the length of the path from the source to u going through w which is $ShortestDistance[w] + T[w, u]$. This operation is called **edge relaxation (χαλάρωση ακμής)** for the edge (w, u).
- The term relaxation is historical. In fact, what we do here is “tighten” the edge.

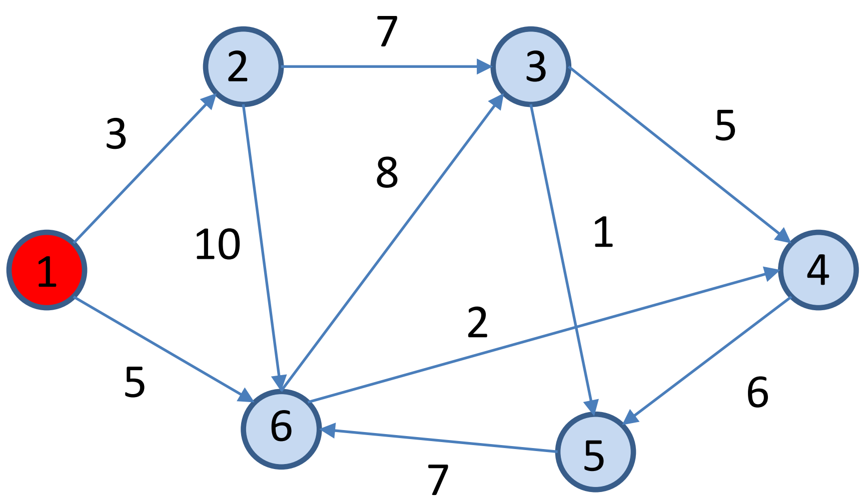
Example Graph



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

Expanding the Vertex Set W in Stages

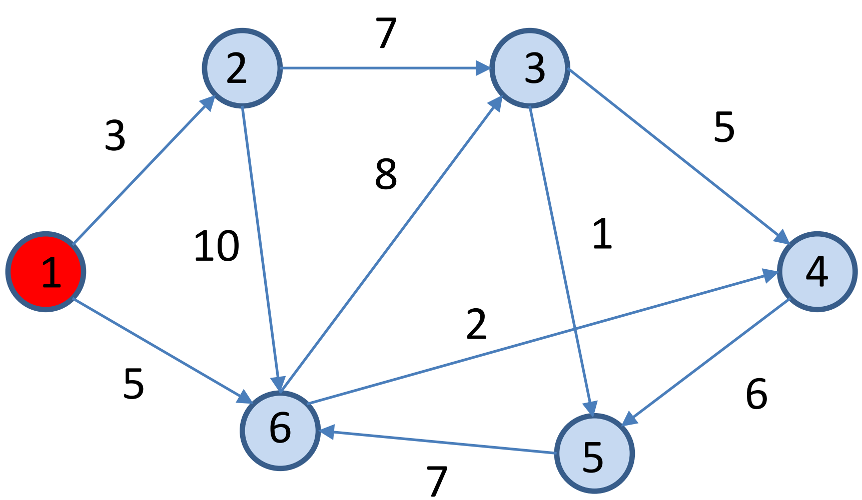
Stage	W	V-W	w	Δ(w)	Δ(1)	Δ(2)	Δ(3)	Δ(4)	Δ(5)	Δ(6)
Start	{1}	{2,3,4,5,6}	-	-	0	3	∞	∞	∞	5



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

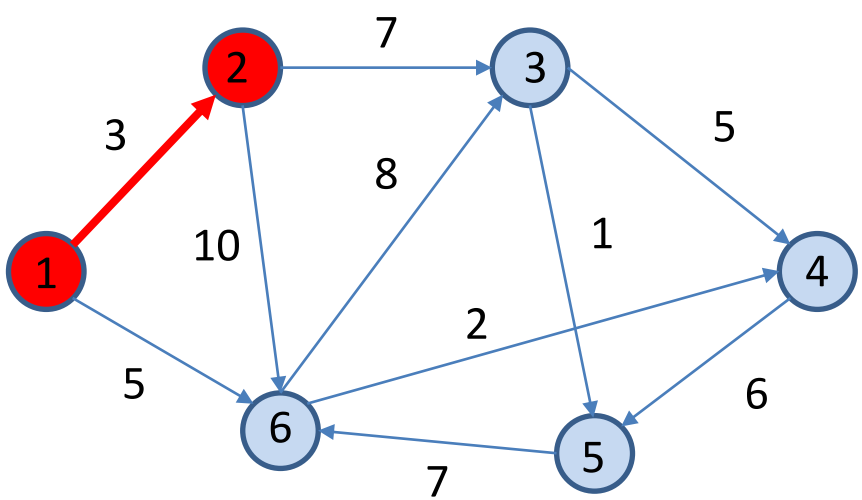
Stage	W	V-W	w	Δ(w)	Δ(1)	Δ(2)	Δ(3)	Δ(4)	Δ(5)	Δ(6)
Start	{1}	{2,3,4,5,6}	-	-	0	3	∞	∞	∞	5

w=2 is chosen for the second stage.



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

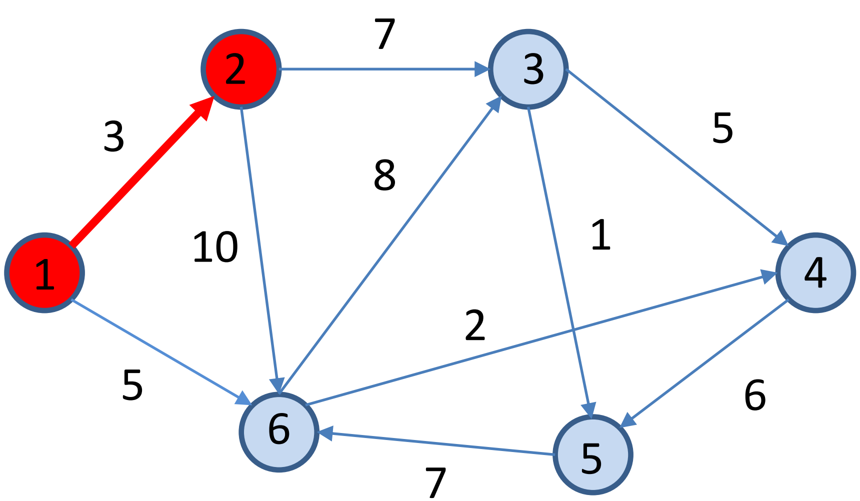
Stage	W	V-W	w	Δ(w)	Δ(1)	Δ(2)	Δ(3)	Δ(4)	Δ(5)	Δ(6)
Start	{1}	{2,3,4,5,6}	-	-	0	3	∞	∞	∞	5
2	{1,2}	{3,4,5,6}	2	3	0	3	10	∞	∞	5



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

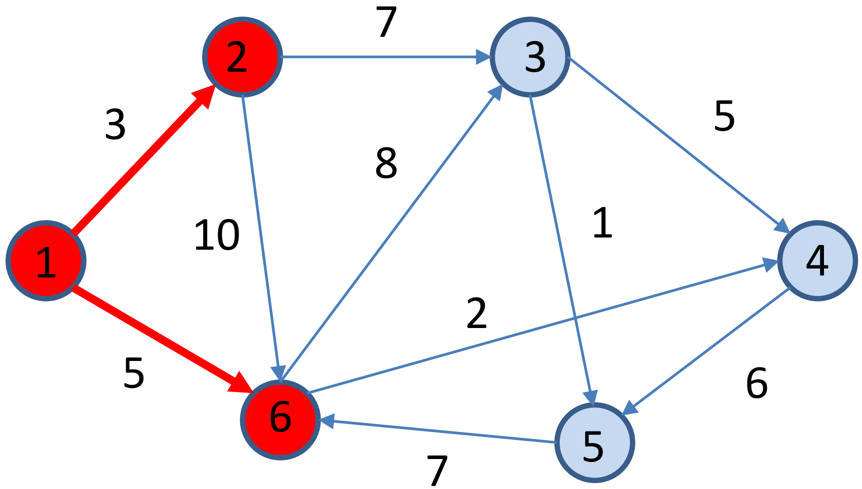
Stage	W	V-W	w	Δ(w)	Δ(1)	Δ(2)	Δ(3)	Δ(4)	Δ(5)	Δ(6)
Start	{1}	{2,3,4,5,6}	-	-	0	3	∞	∞	∞	5
2	{1,2}	{3,4,5,6}	2	3	0	3	10	∞	∞	5

w=6 is chosen for the third stage.



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

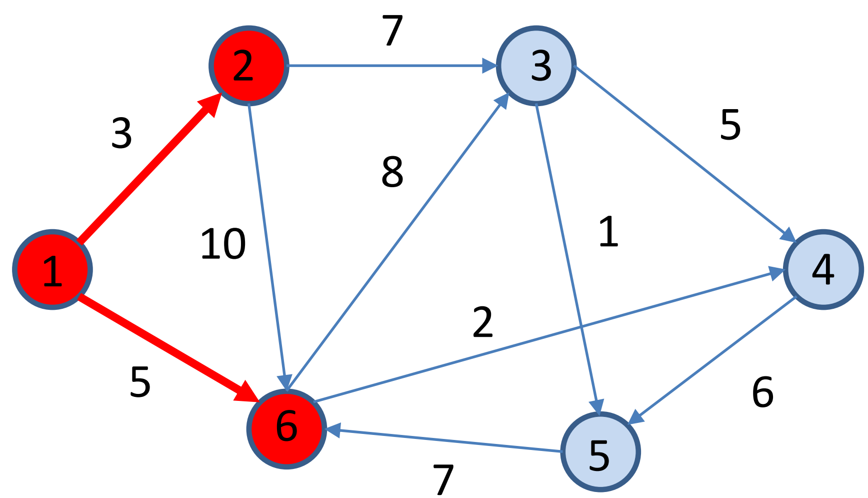
Stage	W	V-W	w	Δ(w)	Δ(1)	Δ(2)	Δ(3)	Δ(4)	Δ(5)	Δ(6)
Start	{1}	{2,3,4,5,6}	-	-	0	3	∞	∞	∞	5
2	{1,2}	{3,4,5,6}	2	3	0	3	10	∞	∞	5
3	{1,2,6}	{3,4,5}	6	5	0	3	10	7	∞	5



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

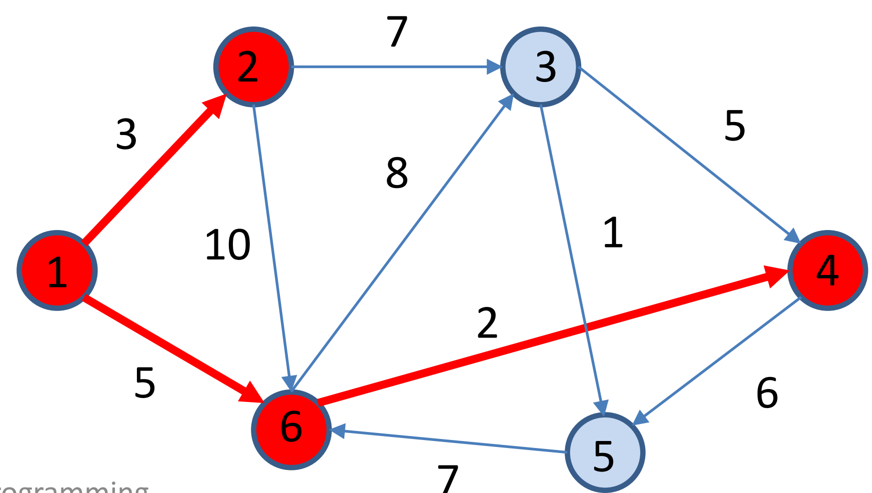
Stage	W	V-W	w	Δ(w)	Δ(1)	Δ(2)	Δ(3)	Δ(4)	Δ(5)	Δ(6)
Start	{1}	{2,3,4,5,6}	-	-	0	3	∞	∞	∞	5
2	{1,2}	{3,4,5,6}	2	3	0	3	10	∞	∞	5
3	{1,2,6}	{3,4,5}	6	5	0	3	10	7	∞	5

w=4 is chosen for the fourth stage.



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

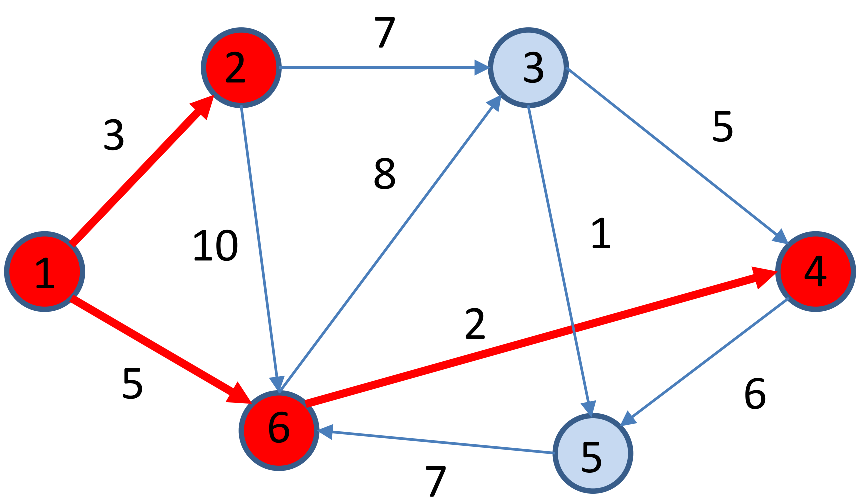
Stage	W	V-W	w	Δ(w)	Δ(1)	Δ(2)	Δ(3)	Δ(4)	Δ(5)	Δ(6)
Start	{1}	{2,3,4,5,6}	-	-	0	3	∞	∞	∞	5
2	{1,2}	{3,4,5,6}	2	3	0	3	10	∞	∞	5
3	{1,2,6}	{3,4,5}	6	5	0	3	10	7	∞	5
4	{1,2,6,4}	{3,5}	4	7	0	3	10	7	13	5



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

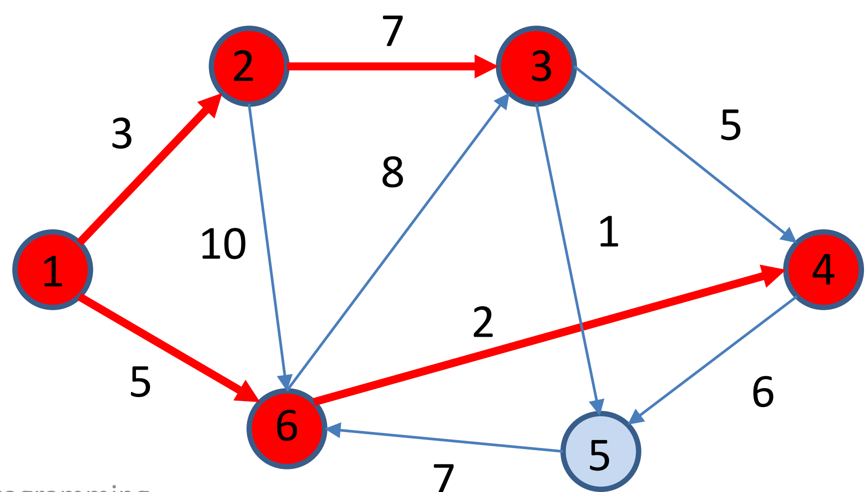
Stage	W	V-W	w	Δ(w)	Δ(1)	Δ(2)	Δ(3)	Δ(4)	Δ(5)	Δ(6)
Start	{1}	{2,3,4,5,6}	-	-	0	3	∞	∞	∞	5
2	{1,2}	{3,4,5,6}	2	3	0	3	10	∞	∞	5
3	{1,2,6}	{3,4,5}	6	5	0	3	10	7	∞	5
4	{1,2,6,4}	{3,5}	4	7	0	3	10	7	13	5

w=3 is chosen for the fifth stage.



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

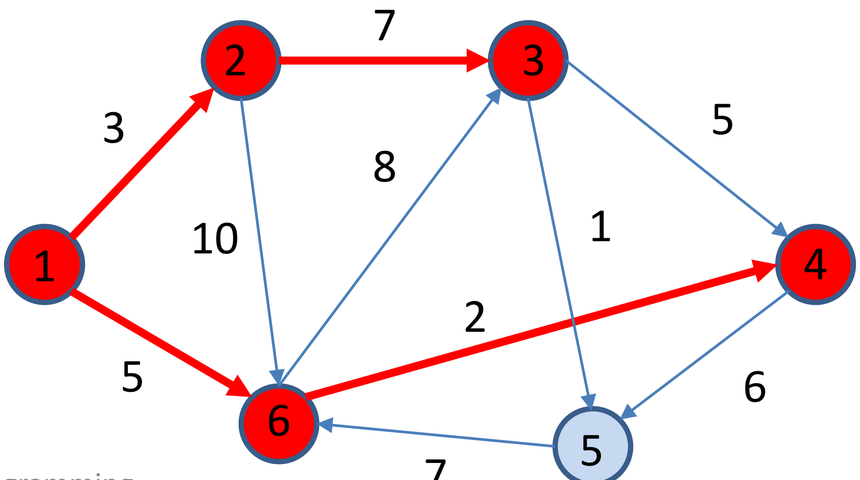
Stage	W	V-W	w	Δ(w)	Δ(1)	Δ(2)	Δ(3)	Δ(4)	Δ(5)	Δ(6)
Start	{1}	{2,3,4,5,6}	-	-	0	3	∞	∞	∞	5
2	{1,2}	{3,4,5,6}	2	3	0	3	10	∞	∞	5
3	{1,2,6}	{3,4,5}	6	5	0	3	10	7	∞	5
4	{1,2,6,4}	{3,5}	4	7	0	3	10	7	13	5
5	{1,2,6,4,3}	{5}	3	10	0	3	10	7	11	5



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

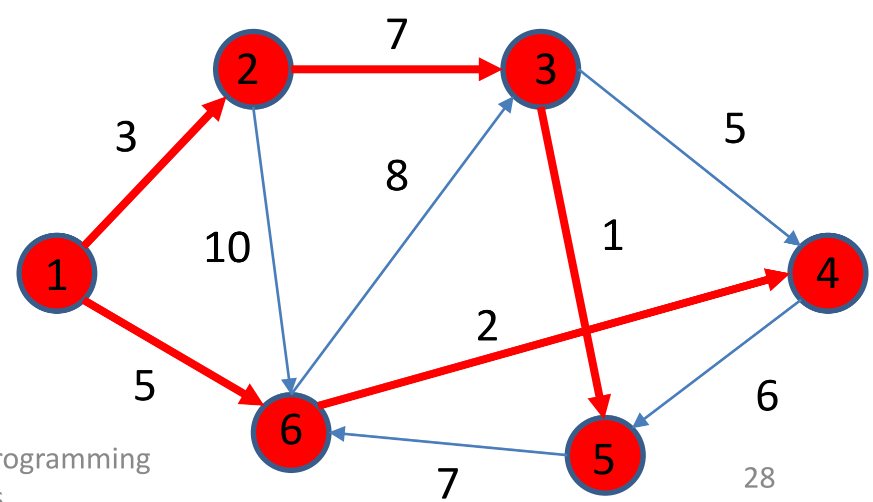
Stage	W	V-W	w	Δ(w)	Δ(1)	Δ(2)	Δ(3)	Δ(4)	Δ(5)	Δ(6)
Start	{1}	{2,3,4,5,6}	-	-	0	3	∞	∞	∞	5
2	{1,2}	{3,4,5,6}	2	3	0	3	10	∞	∞	5
3	{1,2,6}	{3,4,5}	6	5	0	3	10	7	∞	5
4	{1,2,6,4}	{3,5}	4	7	0	3	10	7	13	5
5	{1,2,6,4,3}	{5}	3	10	0	3	10	7	11	5

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



Dijkstra’s Algorithm in Pseudocode

```
void ShortestPath(void)
{
  Let T be the adjacency matrix of graph G.
  Let MinDistance be a variable that takes edge
  weights as values.
  Let Minimum(x,y) be a function whose value is the lesser
  of x and y.

  /* Let s in V be the source vertex at which the
     shortest paths starts. */
  /* Initialize W and ShortestDistance[u] as follows: */
  W={s};
  ShortestDistance[s]=0;
  for (each u in V-{s}) ShortestDistance[u]=T[s][u];
```

Dijkstra’s Algorithm (cont’d)

```
  /*Now repeatedly enlarge W until W includes all vertices in V */
  while (W!=V){
    /* find the vertex w in V-W at the minimum distance from s */
    MinDistance=∞;
    for (each v in V-W){
      if (ShortestDistance[v] < MinDistance){
        MinDistance=ShortestDistance[v];
        w=v;
      }
    }

    /* add w to W */
    W=W ∪ {w};

   /* relaxation step: update the shortest distance to vertices in V-W */
    for (each u in V-W){
      ShortestDistance[u]=Minimum(ShortestDistance[u],
                                  ShortestDistance[w]+T[w][u]);
    }
  }
}
```

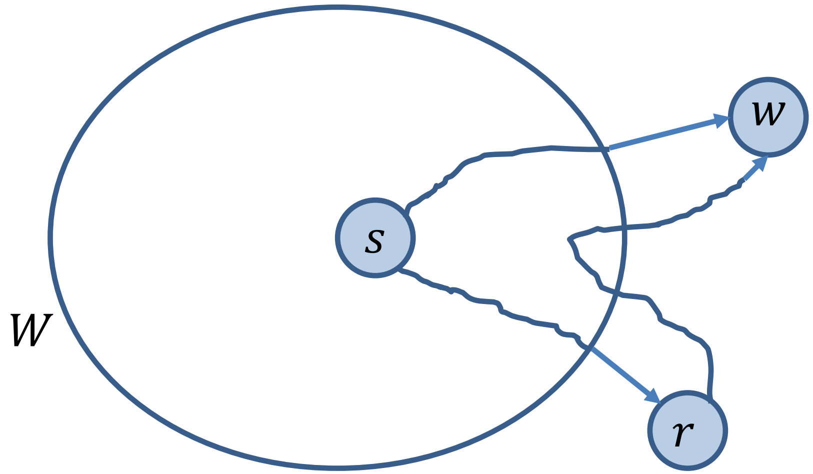
Proof of Correctness for Dijkstra's Algorithm

- We will first prove that at each stage of the algorithm, when w is selected, $ShortestDistance[w]$ gives us the length of the shortest path from the source to w.

Proof (cont'd)

- Let us assume that this is not the case i.e., $ShortestDistance[w]$ is **not** the length of the shortest path from s to w.
- Then, there must exist some shorter path p, which starts at s and contains a vertex in $V - W$ other than w.
- We can start at the source s and proceed along path p, passing through vertices in W, until we come to the first vertex r, that is not in W as the next figure shows.

Hypothetical Shorter Path to w



Proof (cont'd)

- Now notice that the length of the initial portion of the path p from s to r is shorter than the length of the entire path p from s to w.
- Since we assumed that the length of path p was shorter than $ShorterDistance[w]$, the length of the path from s to r is shorter than $ShorterDistance[w]$ also.
- Moreover, the path from s to r has all its vertices except for r lying in W.
- Thus we would have $ShortestDistance[r] < ShortestDistance[w]$ when w was chosen as the next vertex to add to W.
- But this contradicts the choice of w and would have meant that we would have chosen r instead.
- Since we reached a contradiction, $ShorterDistance[w]$ is the length of the shortest path from s to w.

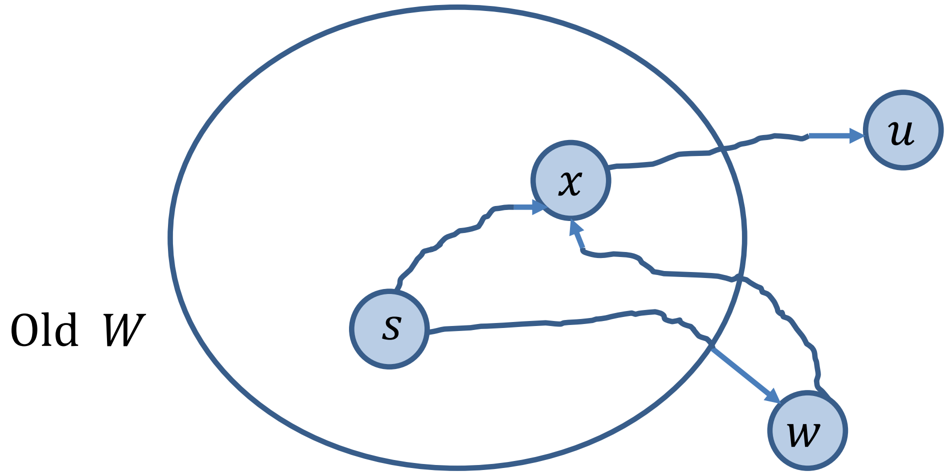
Proof (cont'd)

- We will now prove that, at each stage, after W is enlarged by the addition of w and shortest distances updated, $ShortestDistance[u]$ gives the distance of the shortest path from s to every vertex u in $V - W$ via intermediaries lying wholly in W.

Proof (cont'd)

- Observe that when we add a new vertex w to W, we adjust the shortest distances to take into account of the possibility that there is now a shorter path to u going through w.
- If that path goes through the old W to w and then immediately to u, its length will be compared with $ShorterDistance[u]$ and $ShorterDistance[u]$ will be reduced if the new path is shorter.
- The only other possibility for a shorter path is shown on the next slide where the path travels to w, then back into the old W, to some member x of the old W, then to u.

Impossible Shortest Path



Proof (cont'd)

- But there really cannot be such a path. Since x was placed in W before w, the shortest of all paths from the source to x runs through the old W alone.
- Therefore, the path to x through w shown on the figure is no shorter than the path directly to x through W.
- As a result, the length of the path from the source to w, x and u is no less from the old value of $ShorterDistance[u]$.
- Thus, $ShorterDistance[u]$ cannot be reduced by the algorithm due to a path through w and x, and we need not consider the length of such paths.

Time Complexity

- If we use an adjacency matrix to represent the digraph, Dijkstra's algorithm runs in $\boldsymbol{O(n^2)}$ **time** where n is the number of vertices of the graph.
- The initialization stage runs through $n-1$ vertices and takes time $O(n)$.
- The while-loop runs through the $n-1$ vertices of $V-\{s\}$ one at a time, and for each such vertex, the selection of the new vertex at minimum distance, as well as the updating of the distances takes time proportional to the number of vertices in $V-W$. Therefore, the loop takes $O(n^2)$ time.

Time Complexity (cont'd)

- If the number of edges in the graph $\boldsymbol{e}$ **is much less than** $\boldsymbol{n^2}$ **e.g.** $\boldsymbol{O(n)}$**(i.e., the graph is sparse)** it is better to use the **adjacency list** representation of the graph and a **priority queue** to organize the vertices in $V - W$ according to the values of array $ShortestDistance$.
- Then, the updating of the array $ShortestDistance$ can be done by going down the adjacency list of w and updating the distances in the priority queue. A total of e updates will be made, each at cost $O(\log n)$ if the priority queue is implemented as a **min heap**, so the total time for updates is $O(e \log n)$.

Time Complexity (cont'd)

- The time to initialize the priority queue is $O(n)$.
- The time needed to select w is $O(\log n)$ since it involves finding and removing the minimum element in a heap.
- Thus, the total time of the algorithm is $\boldsymbol{O(n + e \log n)}$ which is considerably better than $O(n^2)$ for sparse graphs.

The All-Pairs Shortest Path Problem

- Suppose we have a weighted digraph that gives the flying time on certain routes containing cities, and we wish to construct a table that gives the shortest time required to fly from any one city to any other.
- This is an instance of the **all-pairs shortest path problem.**

The All-Pairs Shortest Path Problem (cont'd)

- More formally, let $G = (V, E)$ be a weighted directed graph in which each edge (v, w) has a non-negative weight $C[v, w]$. The **all-pairs shortest path problem** is to find for each pair of vertices v, w, the shortest path from v to w.
- We could solve this problem by running Dijkstra's algorithm with each vertex in turn as a source.
- We will present a more direct way of solving the problem due to **R. W. Floyd.**

Floyd's Algorithm

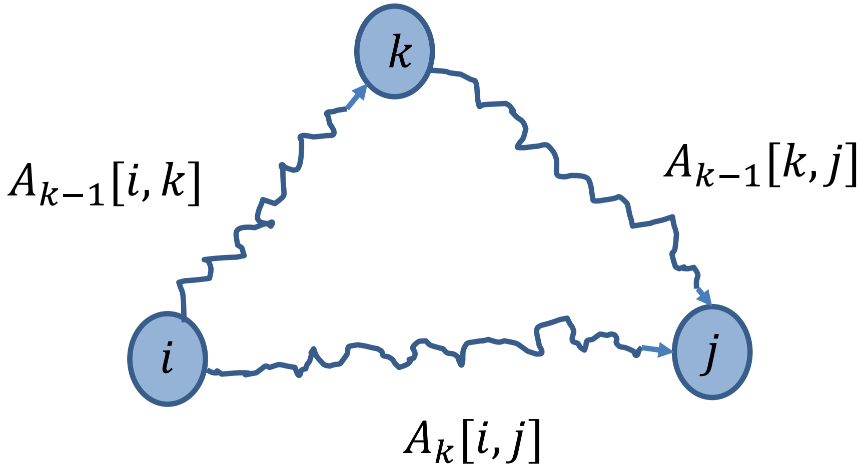
- Let us assume that vertices in V are numbered with $0, 1, 2, \ldots, n-1$. The algorithm uses an $n \times n$ matrix A in which to compute the lengths of the shortest paths.
- We initially set $A[i,j] = C[i,j]$ where C is the adjacency matrix of G.
- As a result, if there is no edge from i to j, we have $A[i,j] = \infty$.
- Also, each diagonal element of A is 0.

Floyd’s Algorithm (cont’d)

- The algorithm makes n iterations over the matrix A.
- After the ***k*-th iteration**, $A[i, j]$ will have as value the smallest length of any path from vertex i to vertex j that does not pass through a vertex numbered higher than k.
- In the k-th iteration, we use the following formulas to compute A:

$$A_k[i, j] = min\begin{cases} A_{k-1}[i, j] \\ A_{k-1}[i, k] + A_{k-1}[k, j] \end{cases}$$

The k-th Iteration Graphically



Floyd’s Algorithm (cont’d)

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

  for (i=0; i<=MAX-1; i++)
    for (j=0; j<=MAX-1; j++)
       A[i][j]=C[i][j];


  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];
}
```

Time Complexity

- The running time of Floyd's algorithm is $\boldsymbol{O(n^3)}$ where n is the number of vertices.

Existence of Paths

- In some problems we may be interested in determining only **whether there exists a path** of length one or more from vertex i to vertex j of directed graph G (the weights are not considered or weights do not exist).
- The algorithm for this problem is a modification of Floyd's algorithm, which historically predates Floyd's algorithm, called **Warshall's algorithm.**

Existence of Paths (cont'd)

- Suppose our weight matrix C is just the **adjacency matrix** of graph G. That is, $C[i,j] = 1$ if there is an edge from i to j, and 0 otherwise.
- We wish to compute the matrix A such that $A[i,j] = 1$ if there is a path of length one or more from i to j, and 0 otherwise.
- A is the **transitive closure (μεταβατική κλειστότητα)** of the adjacency matrix.

Transitive Closure

- The transitive closure can be computed using a procedure similar to the one we used for the all-pairs shortest path problem.
- We apply the following formula in the k-th pass over the Boolean matrix A:

$$A_k[i,j] = A_{k-1}[i,j] \ \boldsymbol{or} \ (A_{k-1}[i,k] \ \boldsymbol{and} \ A_{k-1}[k,j])$$

- The formula states that there is a path from i to j not passing through a vertex numbered higher than k if
  - there is already a path from i to j not passing through a vertex number higher than $k-1$ or
  - there is a path from i to k not passing through a vertex numbered higher than $k-1$ and a path from k to j not passing through a vertex numbered higher than $k-1$.

Transitive Closure (cont'd)

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

  for (i=0; i<=MAX-1; i++)
    for (j=0; j<=MAX-1; j++)
       A[i][j]=C[i][j];


  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];
}
```

Time Complexity

- The running time of Warshall's algorithm is $\boldsymbol{O(n^3)}$ where n is the number of vertices.

Readings

- T. A. Standish. *Data Structures , Algorithms and Software Principles in C.*
  - Chapter 10
- A. V. Aho, J. E. Hopcroft and J. D. Ullman. *Data Structures and Algorithms.*
  - Chapters 6 and 7
- T. H Cormen, C. E. Leiserson and R.L. Rivest. *Introduction to Algorithms*.
  - Chapters 25 and 26.