Weighted graphs

- Graphs with numbers, called *weights*, attached to each edge
  - Often restricted to *non-negative*
- Directed or undirected
- Examples
  - *Distance* between cities
  - *Cost* of flight between airports
  - *Time* to send a message between routers

Adjacency matrix representation

\[
T[i, j] = \begin{cases} 
  w_{i,j} & \text{if } i, j \text{ are connected} \\
  \infty & \text{if } i \neq j \text{ are not connected} \\
  0 & \text{if } i = j 
\end{cases}
\]

- Similarly for adjacency lists

Example weighted graph

![Example weighted graph]
**Example weighted graph**

![Example weighted graph]

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>3</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>∞</td>
<td>0</td>
<td>7</td>
<td>∞</td>
<td>∞</td>
<td>10</td>
</tr>
<tr>
<td>3</td>
<td>∞</td>
<td>∞</td>
<td>0</td>
<td>5</td>
<td>1</td>
<td>∞</td>
</tr>
<tr>
<td>4</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>0</td>
<td>6</td>
<td>∞</td>
</tr>
<tr>
<td>5</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>0</td>
<td>7</td>
</tr>
<tr>
<td>6</td>
<td>∞</td>
<td>∞</td>
<td>8</td>
<td>2</td>
<td>∞</td>
<td>0</td>
</tr>
</tbody>
</table>

Adjacency matrix

**Shortest paths**

- The **length** of a path is the **sum of the weights** of its edges
- Very common problem
  - find the **shortest path** from $s$ to $d$
- Examples
  - Shortest route between cities
  - Cheapest connecting flight
  - Fastest network route
  - ...

**Shortest path from vertex 1 to vertex 5**

![Shortest path from vertex 1 to vertex 5]

**Shortest path problem**

Two main variants:

- **Single source** $s$
  - Find the shortest path from $s$ to each node
  - **Dijkstra’s** algorithm
    - Only for **non-negative** weights (important!)
- **All-pairs**
  - Find the shortest path between all pairs $s, d$
  - **Floyd-Warshall** algorithm
    - Any weights
Dijkstra's algorithm

Main ideas:
- Keep a set $W$ of visited nodes
  - Start with $W = \{s\}$ (or $W = \{\}$)
- Keep a matrix $\Delta[u]$
  - Minimum distance from $s$ to $u$ passing only through $W$
    - Start with $\Delta[u] = T[s, u]$ (or $\Delta[s] = 0, \Delta[u] = \infty$)
- At each step we enlarge $W$ by adding a new vertex $w \not\in W$
  - $w$ is the one with minimum $\Delta[w]$

Example graph

Expanding the vertex set $w$ in stages

<table>
<thead>
<tr>
<th>Stage</th>
<th>W</th>
<th>V-W</th>
<th>w</th>
<th>$\Delta(w)$</th>
<th>$\Delta(1)$</th>
<th>$\Delta(2)$</th>
<th>$\Delta(3)$</th>
<th>$\Delta(4)$</th>
<th>$\Delta(5)$</th>
<th>$\Delta(6)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Start</td>
<td>{1}</td>
<td>{2,3,4,5,6}</td>
<td>-</td>
<td>-</td>
<td>0</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>
Expanding the vertex set \( w \) in stages

\( W=2 \) is chosen for the second stage.

<table>
<thead>
<tr>
<th>Stage</th>
<th>( W )</th>
<th>( V-W )</th>
<th>( w )</th>
<th>( \Delta(w) )</th>
<th>( \Delta(1) )</th>
<th>( \Delta(2) )</th>
<th>( \Delta(3) )</th>
<th>( \Delta(4) )</th>
<th>( \Delta(5) )</th>
<th>( \Delta(6) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Start</td>
<td>{1}</td>
<td>{2,3,4,5,6}</td>
<td>-</td>
<td>-</td>
<td>0</td>
<td>( \infty )</td>
<td>( \infty )</td>
<td>( \infty )</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>{1,2}</td>
<td>{3,4,5,6}</td>
<td>2</td>
<td>3</td>
<td>0</td>
<td>3</td>
<td>10</td>
<td>( \infty )</td>
<td>( \infty )</td>
<td>5</td>
</tr>
</tbody>
</table>

Expanding the vertex set \( w \) in stages

\( W=6 \) is chosen for the third stage.

<table>
<thead>
<tr>
<th>Stage</th>
<th>( W )</th>
<th>( V-W )</th>
<th>( w )</th>
<th>( \Delta(w) )</th>
<th>( \Delta(1) )</th>
<th>( \Delta(2) )</th>
<th>( \Delta(3) )</th>
<th>( \Delta(4) )</th>
<th>( \Delta(5) )</th>
<th>( \Delta(6) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Start</td>
<td>{1}</td>
<td>{2,3,4,5,6}</td>
<td>-</td>
<td>-</td>
<td>0</td>
<td>3</td>
<td>( \infty )</td>
<td>( \infty )</td>
<td>( \infty )</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>{1,2}</td>
<td>{3,4,5,6}</td>
<td>2</td>
<td>3</td>
<td>0</td>
<td>3</td>
<td>10</td>
<td>( \infty )</td>
<td>( \infty )</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>{1,2,6}</td>
<td>{3,4}</td>
<td>6</td>
<td>5</td>
<td>0</td>
<td>3</td>
<td>10</td>
<td>7</td>
<td>( \infty )</td>
<td>5</td>
</tr>
</tbody>
</table>
Expanding the vertex set $w$ in stages

Stage $W$ $V-W$ $w$ $\Delta(w)$ $\Delta(1)$ $\Delta(2)$ $\Delta(3)$ $\Delta(4)$ $\Delta(5)$ $\Delta(6)$

Start [1] (2,3,4,5,6) - - 0 3 $\infty$ $\infty$ $\infty$ 5
2 [1,2] (3,4,5,6) 2 3 0 3 10 $\infty$ $\infty$ 5
3 [1,2,6] (3,4,5) 6 5 0 3 10 7 $\infty$ 5

$W=4$ is chosen for the fourth stage.

Stage $W$ $V-W$ $w$ $\Delta(w)$ $\Delta(1)$ $\Delta(2)$ $\Delta(3)$ $\Delta(4)$ $\Delta(5)$ $\Delta(6)$

Start [1] (2,3,4,5,6) - - 0 3 $\infty$ $\infty$ $\infty$ 5
2 [1,2] (3,4,5,6) 2 3 0 3 10 $\infty$ $\infty$ 5
3 [1,2,6] (3,4,5) 6 5 0 3 10 7 $\infty$ 5
4 [1,2,6,4] (3,5) 4 7 0 3 10 7 13 5

$W=3$ is chosen for the fifth stage.

Stage $W$ $V-W$ $w$ $\Delta(w)$ $\Delta(1)$ $\Delta(2)$ $\Delta(3)$ $\Delta(4)$ $\Delta(5)$ $\Delta(6)$

Start [1] (2,3,4,5,6) - - 0 3 $\infty$ $\infty$ $\infty$ 5
2 [1,2] (3,4,5,6) 2 3 0 3 10 $\infty$ $\infty$ 5
3 [1,2,6] (3,4,5) 6 5 0 3 10 7 $\infty$ 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

\[
\begin{array}{|c|c|c|c|c|c|c|c|}
\hline
\text{Stage} & \text{W} & \text{V-W} & \Delta(W) & \Delta(1) & \Delta(2) & \Delta(3) & \Delta(4) & \Delta(5) \\
\hline
\text{Start} & \{1\} & \{2,3,4,5,6\} & - & - & 0 & 3 & \infty & \infty & \infty & 5 \\
2 & \{1,2\} & \{3,4,5,6\} & 2 & 3 & 0 & 3 & 10 & \infty & \infty & 5 \\
3 & \{1,2,6\} & \{3,4,5\} & 6 & 5 & 0 & 3 & 10 & 7 & \infty & 5 \\
4 & \{1,2,6,4\} & \{3,5\} & 4 & 7 & 0 & 3 & 10 & 7 & 13 & 5 \\
5 & \{1,2,6,4,3\} & \{5\} & 3 & 10 & 0 & 3 & 10 & 7 & 11 & 5 \\
\hline
\end{array}
\]

\( W=5 \) is chosen for the sixth stage.

Dijkstra's algorithm in pseudocode

```plaintext
// Δεδομένα
src : αρχικός κόμβος
dest : τελικός κόμβος

// Πληροφορίες που κρατάμε για κάθε κόμβο v
W[u] : 1 αν o u είναι στο σύνολο W, 0 διαφορετικά
dist[u] : ο πίνακας Δ
prev[u] : ο προηγούμενος του v στο βέλτιστο μονοπάτι

// Αρχικοποίηση: W={} {εναλλακτικά μπορούμε και W={src}}
for each vertex u in Graph
    dist[u] = INT_MAX // infinity
    prev[u] = NULL
    W[u] = 0

dist[src] = 0
```

Expanding the vertex set \( w \) in stages

<table>
<thead>
<tr>
<th>Stage</th>
<th>W</th>
<th>V-W</th>
<th>w</th>
<th>( \Delta(W) )</th>
<th>( \Delta(1) )</th>
<th>( \Delta(2) )</th>
<th>( \Delta(3) )</th>
<th>( \Delta(4) )</th>
<th>( \Delta(5) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Start</td>
<td>{1}</td>
<td>{2,3,4,5,6}</td>
<td>-</td>
<td>-</td>
<td>0</td>
<td>3</td>
<td>\infty</td>
<td>\infty</td>
<td>\infty</td>
</tr>
<tr>
<td>2</td>
<td>{1,2}</td>
<td>{3,4,5,6}</td>
<td>2</td>
<td>3</td>
<td>0</td>
<td>3</td>
<td>10</td>
<td>\infty</td>
<td>\infty</td>
</tr>
<tr>
<td>3</td>
<td>{1,2,6}</td>
<td>{3,4,5}</td>
<td>6</td>
<td>5</td>
<td>0</td>
<td>3</td>
<td>10</td>
<td>7</td>
<td>\infty</td>
</tr>
<tr>
<td>4</td>
<td>{1,2,6,4}</td>
<td>{3,5}</td>
<td>4</td>
<td>7</td>
<td>0</td>
<td>3</td>
<td>10</td>
<td>7</td>
<td>13</td>
</tr>
<tr>
<td>5</td>
<td>{1,2,6,4,3}</td>
<td>{5}</td>
<td>3</td>
<td>10</td>
<td>0</td>
<td>3</td>
<td>10</td>
<td>7</td>
<td>11</td>
</tr>
<tr>
<td>6</td>
<td>{1,2,6,4,3,5}</td>
<td>{}</td>
<td>5</td>
<td>11</td>
<td>0</td>
<td>3</td>
<td>10</td>
<td>7</td>
<td>11</td>
</tr>
</tbody>
</table>

Dijkstra's algorithm in pseudocode

```plaintext
// Κύριας αλγόριθμος
while true
    w = vertex with minimum dist[w], among those with W[w] = 0
    W[w] = 1
    if w == dest
        stop
        // optimal cost = dist[dest]
        // optimal path = dest <- prev[dest] <- ... <- src (inverse)
    for each neighbor u of w
        if W[u] == 1
            continue
        alt = dist[w] + weight(w,u) // κόστος του src -> ... -> w
        if alt < dist[u]
            dist[u] = alt
            prev[u] = w
```

Dijkstra's algorithm in pseudocode
Using a priority queue

- Finding the $w \not\in W$ with minimum $\Delta[w]$ is slow
  - $O(n)$ time
- But we can use a priority queue for this!
  - We only keep vertices $w \not\in W$ in the queue
  - They are compared based on their $\Delta[w]$ (each $w$ has “priority” $\Delta[w]$)
- Careful when $\Delta[w]$ is modified!
  - Either use a priority queue that allows updates
  - Or insert multiple copies of each with different priorities
    - the queue might contain already visited vertices: ignore them

Dijkstra's algorithm with priority queue

```c
// κυρίως αλγόριθμος
c
// while pq is not empty
// w = pqqueue_max(pq) // w with minimal dist[u]
pqqueue_remove_max(pq)
if exists(W[w]) // το w μπορεί να υπάρχει πολλές φορές στην α
  continue // δεν κάνουμε replace), και να είναι ήδη vis
if w == dest
  stop // optimal cost/path same as before
for each neighbor u of w
  if exists(W[u])
    continue
  alt = dist[w] + weight(w,u) // cost of src->...->w->u
  if !exists(dist[u]) OR alt < dist[u]
    dist[u] = alt
    prev[u] = w
    pqqueue_insert(pq, {u,alt}) // προαιρετικά: replace an un
stop // pq άδειασε πριν βρούμε το dest => δεν υπάρχει μονοπάτι
```

Notation

- $s \rightarrow d$
  - Direct step step from $s$ to $d$
- $s \xrightarrow{W} d$
  - Multiple steps $s \rightarrow \ldots \rightarrow d$
  - All intermediate steps belong to the set $W \subseteq V$
- $s \xrightarrow{V} d$
  - Shortest path among all $s \rightarrow d$
  - So $s \xrightarrow{V} d$ is the overall shortest one
**Proof of correctness**

- We need to prove that \( \Delta[u] \) is the **minimum distance to** \( u \)
  - after the algorithm finishes
- But it’s much easier to reason **step by step**
  - we need a property that holds at **every step**
  - this is called an **invariant** (property that never changes)

---

**Invariant of Dijkstra’s algorithm**

- \( \Delta[u] \) is the cost of the shortest path **passing only through** \( W \)
- And the shortest **overall** when \( u \in W \)

Formally:

1. For all \( u \in V \) the path \( s \xrightarrow{W} u \) has cost \( \Delta[u] \)
2. For all \( u \in W \) the path \( s \xrightarrow{V} u \) has cost \( \Delta[u] \)

Proof: **induction** on the **size of** \( W \), for both (1), (2) together

---

**Base case** \( W = \{ s \} \)

- Trivial, the only path \( s \xrightarrow{W} u \) is the direct one \( s \rightarrow u \)
- For (1): its cost is exactly \( T[s, u] = \Delta[u] \)
  - initial value of \( \Delta[u] \)
- For (2): the only \( u \in W \) is \( s \) itself

---

**Inductive case**

- Assume \( |W| = k \) and (1),(2) hold
- The algorithm
  - Updates \( W \), adding a new vertex \( w \not\in W \)
  - Updates \( \Delta[u] \) for all neighbours \( u \) of \( w \)
- Let \( W', \Delta' \) be the values **after** the update
- Show that (1),(2) still hold for \( W', \Delta' \)
Proof of correctness

We start showing that (2) still holds for $W', \Delta'$

- The interesting vertex is the $w$ we just added
  - Vertices $u \neq w$ are trivial from the induction hypothesis
- Show: $s \xrightarrow{V} w$ has cost $\Delta'[w]$
  - Note: $\Delta'[w] = \Delta[w]$ (we do not update $\Delta[w]$)
  - We already know that $s \xrightarrow{W} w$ has cost $\Delta[w]$ (ind. hyp)
  - So we just need to prove that there is no better path outside $W$

Proof of correctness

It remains to show (1) for $W', \Delta'$

- Take some arbitrary $u$
  - Let $c$ be the cost of $s \xrightarrow{W} u$
  - Show: $c = \Delta'[u]$
- Three cases for the optimal path $s \xrightarrow{W'} u$
- Case 1: the path does not pass through $w$
  - So it is of the form $s \xrightarrow{W} u$
  - This path has cost $\Delta[u]$ (induction hypothesis)
  - No update: $\Delta'[u] = \Delta[u] = c$

Proof of correctness

- Case 2: $w$ is right before $u$
  - So the path is of the form $s \xrightarrow{W} w \xrightarrow{V} u$
  - The cost of $s \xrightarrow{W} w$ is $\Delta[w]$ (induction hypothesis)
  - So $c = \Delta[w] + T[w, u]$
  - So the algorithm will set $\Delta'[u] = \Delta[w] + T[w, u]$ when updating the neighbours of $w$
  - So $c = \Delta'[u]$

Proof of correctness

- Assuming such path exists, let $r$ be its first vertex outside $W$
  - So the path $s \xrightarrow{W} r \xrightarrow{V} w$ has cost $c < \Delta[w]$
  - So the path $s \xrightarrow{W} r$ has cost at most $c < \Delta[w]$ (no negative weights!)
  - So $\Delta[r] < \Delta[w]$

- Impossible! We chose $w$ to be the one with min $\Delta[w]$

Proof of correctness

- Case 2: $w$ is right before $u$
  - So the path is of the form $s \xrightarrow{W} w \xrightarrow{V} u$
  - The cost of $s \xrightarrow{W} w$ is $\Delta[w]$ (induction hypothesis)
  - So $c = \Delta[w] + T[w, u]$
  - So the algorithm will set $\Delta'[u] = \Delta[w] + T[w, u]$ when updating the neighbours of $w$
  - So $c = \Delta'[u]$
Proof of correctness

- Case 3: some other $x$ appears after $w$ in the path
  - So the path is of the form $s \xrightarrow{W} w \rightarrow x \xrightarrow{W} u$
  - But the path $s \xrightarrow{W} w \rightarrow x$ is no shorter than $s \xrightarrow{W} x$
    - From the induction hypothesis for $x \in W$
  - So $s \xrightarrow{W} x \rightarrow u$ is also optimal, reducing to case 1!

Complexity

Without a priority queue:

- Initialization stage: loop over vertices: $O(n)$
- The while-loop adds one vertex every time: $n$ iterations
- Finding the new vertex loops over vertices: $O(n)$
  - same for updating the neighbours
- So total $O(n^2)$ time

With a priority queue:

- Initialization stage: loop over vertices, so $O(n)$
- Count the number of updates (steps in the inner loop)
  - Once for every neighbour of every node: $e$ total
  - Each update is $O(\log n)$ (at most $n$ elements in the queue)
- Total $O(e \log n)$
  - Assuming a connected graph ($e \geq n$)
  - And an implementation using adjacency lists
- Only good for sparse graphs!
  - But $O(n \log n)$ can be hugely better than $O(n^2)$

The all-pairs shortest path problem

- Find the shortest path between all pairs $s, d$
- Floyd-Warshall algorithm
- Any weights
  - Even negative
  - But no negative loops (why?)
The all-pairs shortest path problem

Main idea
- At each step we compute the shortest path through a subset of vertices
  - Similarly to $W$ in Dijkstra
  - But now the set at step $k$ is $W_k = \{1, \ldots, k\}$
  - Vertices are numbered in any order
- Step $k$: the cost of $i \xrightarrow{W_k} j$ is $A_k[i,j]$ 
  - Similar to $\Delta$ in Dijkstra (but for all pairs of nodes)

Floyd-Warshall algorithm

- The algorithm at each step computes $A_k$ from $A_{k-1}$
- Initial step $k = 0$
  - Start with $A_0[i,j] = T[i,j]$ 
  - Only direct paths are allowed

$k$-th iteration: the optimal $i \xrightarrow{W_k} j$ either passes through $k$ or not.

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

Floyd-Warshall algorithm in pseudocode

```java
void floyd_warshall() {
    for (int i = 0; i < size-1; i++)
        for (int j = 0; j < size-1; j++)
            A[i][j] = weight(i, j);

    for (int i = 0; i < size-1; i++)
        A[i][i] = 0;

    for (int k = 0; k < size-1; k++)
        // Compute $A_k$ from $A_{k-1}$
        for (int i = 0; i < size-1; i++)
            for (int j = 0; j < size-1; j++)
}
```

$A$ is the current $A_k$ at every step $k$. 
**Complexity**

- Three simple loops of $n$ steps
- So $O(n^3)$
- **Not** better than $n$ executions of Dijkstra in complexity
  - But much simpler
  - Often faster in practice
  - And works for **negative** weights

**Readings**

- T. A. Standish. *Data Structures, Algorithms and Software Principles in C*. Chapter 10