Shortest paths in graphs
Remarks from previous lectures:

- Path length in unweighted graph equals to edge count on the path
- Oriented distance ($\delta(u, v)$) between vertices $u, v$ equals to the length of the shortest path from $u$ to $v$
- In an oriented graph, distance between two vertices need not to be symmetrical ($\delta(u, v) \neq \delta(v, u)$ in general)

**Figure:** In this case $\delta(u, v) \neq \delta(v, u)$. 

![Diagram](image-url)
Distance in weighted graph

In real-world applications, graph edges are weighted – e.g., distances between cities, latency of network links.

**Definition**

*Path length in weighted graph equals to sum of edge weights along the path.*

- Distance between vertices is defined as the length of the shortest path between them.
- Negative-weight cycle potentially allows some or all distances in the graph to be any negative number.

By definition, the shortest paths do not contain any nonnegative-weight cycle.
The triangle inequality holds for a graph if and only if

\[ \delta(u, w) \leq \delta(u, v) + \delta(v, w) \]

The triangle inequality does not hold in general, a graph of the shortest (not direct) distances between cities is the real-world example in which the inequality holds.
Dijkstra’s algorithm

- Well-known algorithm for finding single-source shortest paths.
- Solves the problem for both directed and undirected graphs.
- Computes shortest paths from single source vertex to all others.
- Requires non-negative weights of all edges (not only cycles).
- Linear space complexity.
- Time complexity depends on chosen data structure.
Dijkstra’s algorithm

- Denote source vertex as \( s \).
- For each vertex \( v \) in a graph, \( d[v] \) equals to length the shortest path from \( s \) to \( v \) found so far.
  - Initially, \( d[s] = 0 \) for source vertex and \( d[v] = \infty \) for the others.
  - Upon completion, \( d[v] \) equals to length of the shortest path in the graph if it exists, or \( \infty \) otherwise.
- \( p[v] \) stores the direct predecessor of vertex \( v \) on the shortest path from \( s \) found so far.
  - Initially, \( p[v] \) is undefined for all vertices except \( s \).
  - Upon completion, the shortest path to \( v \) is the sequence \( s, p[...p[v]...], ... p[p[v]], p[v], v \).
Dijkstra’s algorithm

- Vertices are split into two disjoint sets:
  - $S$ contains exactly those vertices, for which the shortest paths has already been computed and stored in $d[v]$.
  - $Q$ contains all other vertices.

- The vertices of set $Q$ are stored in a priority queue.
  - The vertex with the lowest value of $d[u]$ has the highest priority. The $d[u]$ already stores length of the shortest path to $u$.

- Following steps are taken in each iteration:
  - Remove the vertex $u$ from the queue head.
  - Move the vertex $u$ from $Q$ to $S$.
  - Relax all edges $(u, v)$ going out from $u$ to any $v$ in $Q$:
    - $w(u, v)$ denotes weight of the edge $(u, v)$. 
**Dijkstra’s algorithm – example**

**Figure:** Vertices in the set $S$ are marked blue. Content of the priority queue is depicted to the right of the graph (head on top).
Dijkstra’s algorithm – animations & illustrations

- Animation on an example graph

- Commented computation
  - http://www.youtube.com/watch?v=8Ls1RqHCOPw

- Computation allowing to input your own graph
  - http://www.cse.yorku.ca/~aaw/HFHuang/DijkstraStart.html

- Illustration of a computation
  - http://www.animal.ahrgr.de/showAnimationDetails.php3?lang=en&anim=16
Dijkstra’s algorithm – time complexity

Let’s denote $n = |V|$, $m = |E|$.

- Initialization is linear w. r. t. number of vertices.
- Each edge is traversed exactly once or twice (in case of oriented graph).
- Main loop is run $n$-times, hence
- there are $n$ delete operations on the priority queue.
- Complexity of the delete operations depends on chosen data structure:
  - **Array, vertex list** – deletion can be done in linear time, complexity of the whole algorithm is therefore in $\mathcal{O}(n^2 + m)$.
  - **Binary heap** – deletion requires $\mathcal{O}(\log(n))$ time. Moreover, each edge relaxation may require heap update ($\mathcal{O}(\log(n))$, overall complexity is in $\mathcal{O}((n + m)\log(n))$).
  - **Fibonacci’s heap** – complexity of the deletion is the same as in the case of binary heap, however update on relaxation runs in constant time – overall complexity is in $\mathcal{O}(m + n\log(n))$.

http://en.wikipedia.org/wiki/Fibonacci_heap
Link-state routing protocols make use of the Dijkstra’s algorithm.

- Each active element broadcasts its neighbors list periodically.
- Neighbors list are forwarded through the network to all active elements.
- Each active element calculates a shortest paths tree to all other AEs independently.
- Risk of loops in routing tables.

OSPF and IS-IS are the most widespread link-state protocols. They both use the Dijkstra’s algorithm.
Floyd-Warshall’s algorithm

- Computes shortest paths between each pair of vertices.
- The algorithm works with negative-weight edges correctly, however, negative-weight cycles may lead to incorrect solution.
- The shortest (so far) known distance between any two vertices is being improved gradually.
- In each step, a set of vertices which may lie on the shortest paths is defined.
- Each iteration introduces a new vertex into this set.
- In each one of $n$ iterations, shortest paths between all $n^2$ pairs of vertices are updated. The time complexity therefore equals to $O(n^3)$.
- The space complexity is $O(n^2)$. 
Floyd-Warshall’s algorithm

- Let the vertices be numbered as $1 \ldots n$.
- At first, only single-edge paths are considered. Afterwards, the algorithm searches for paths traversing through vertex 1. Subsequently, paths using vertices 1 and 2, etc.
- Between any pair of vertices $u, v$, a shortest path using vertices $1 \ldots k$ is known in $(k + 1)^{th}$ iteration.
- There are two possibilities for the shortest path (which uses vertices $1 \ldots k + 1$) between these two vertices:
  - It uses only the $1 \ldots k$ vertices.
  - It traverses vertices $1 \ldots k$ from $u$ to vertex $k + 1$ and then ends in $v$.
- Upon completion, shortest paths using all vertices in the graph are computed.
Floyd-Warshall’s algorithm – an example

Figure: Vertices which may be used for shortest paths are highlighted. Shortest paths computed so far are stored in the matrix.
Floyd-Warshall’s algorithm can be easily applied in distributed environment – among autonomous units, which communicate only through message sending

- Each vertex computes shortest paths to all other graph vertices
- Initially, only path to neighbours is known
- Similarly to the sequential case, each iteration adds single vertex which can be included in the paths
- Added vertex broadcasts its distances table to all other vertices in each iteration
- The other vertices update their distances and shortest paths according to the received table
Distributed Floyd-Warshall’s algorithm

- It is crucial for correctness of the algorithm that all vertices choose the same vertex in each iteration.

- Algorithm is inefficient in terms of transferred data amount. If $d[v] = \infty$ holds for selected vertex $v$ in any vertex, its paths are not updated at all, hence it does not need to receive any distance tables in the current iteration.

- Before broadcasting distance table, vertices may signal to each other, which of them should receive the table ⇒ Toueg’s algorithm.

- Further information:
Exercises

1. Calculate shortest paths in the graph below using Dijkstra’s and Floyd-Warshall’s algorithm.

2. Propose an implementation of the Floyd-Warshall’s algorithm (Toueg’s algorithm). Consider, that vertices can transmit messages only along graph edges (broadcasting is implemented by forwarding).
Why doesn’t Dijkstra’s algorithm work correctly on graphs with negative-weight edges? What are the possible outcomes when it is run on such graph?