## Weighted Graphs and Algorithms

A **weighted graph** is a set of tuples (V,E,W), where (V,E) is an (un)directed graph, and W is a function mapping edges into (real valued) **weights**.

In the standard graph the length of the path is the number of edges involved in the length, in this new model the *length of the path* is the sum of all its edge labels.

We can represent these graphs as adjacency lists, in which we also need to specify the weights (usually for sparse graphs), or adjacency matrix (dense graphs), in which N (Nil) represent the absence of edge (we don't use 0, since 0 can be a weight), otherwise we report the weight of the edge among the two nodes.

### Single Source Shortest Path

Idea: we want to compute all the shortest path from a single source. In the unweighted case we used BFS ($O(V| + |E|)$).

Recall:

**Lemma**: let $Q = [u_1, ..., u_n]$ be the queue during BFS. Then $u_{i-1} \cdot d \leq u_i \cdot d$ $\forall i \in [2, n]$ and $u_n \cdot d \leq u_1 \cdot d + 1$.

Hence $u_1 \cdot d + 1 \leq u_i \cdot d + 1$ $\forall i \in [2, n]$.

So if $v$ is a successor of $u_1$, any other path reaching $v$ through a node in Q is longer than $u_1 \cdot d + 1$. So BFS works properly on unweighted graphs.

We can think about set $v$'s distance to $u.d + W[(u, v)]$. However, even if $u_{i-1}d \leq u_i \cdot d$ $\forall i \in [2, n]$, there may be $(u_k, \bar{v})$ s.t. $u_1 \cdot d + W[(u_1, v)] \leq u_k \cdot d + W[(u_k, \bar{v})]$

The **Dijkstra's algorithm** solves the single-source shortest-paths problem on a weighted, directed graph G=(V,E) for the case in which all edge weights are nonnegative.

Idea is to enqueue not-discovered nodes in place of the just discovered, and nodes are pre-labeled with a candidate distance.

At each step a node having minimal candidate distance is extracted and finalised. Its outgoing neighbours are updated.

So the BFS queue has become a **priority queue** wrt candidate distance. Note that there is no more need for colouring: white nodes correspond to nodes in Q, nodes are finalised as soon as extracted from Q.

Paths are treated as distances: the predecessor of a node is updated every time a new possible minimum distance arises.

Moreover note that, if there were negative weights, the minimal path to the node extracted from Q could be not discovered.

The complexity of this algorithm is $\Theta(|V|^2 + |E|)$ for queue implemented as array, $‰(|V| + |E| \cdot \log(|V|))$ for queue implemented as binary heaps.

**All pairs shortest Path**

Want to compute the shortest path between all the pairs of nodes. Most naive strategy is to use Dijkstra algorithm for each node (compute all the shortest path in the graph). This strategy cannot be applied with negative weights, and the cost is relevant for dense graphs.

Consider $p = e_1, ..., e_h$ the shortest path from $i$ to $j$. Let $k$ be the greatest internal node in the path. So for sure there exists an index $\bar{h}$ such that $e_{\bar{h}-1}$ and $e_{\bar{h}}$ have $k$ as destination and source respectively.

So $e_1, .., e_{\bar{h}-1}$ and $e_{\bar{h},...,e_h}$ are shortest path between $i$ and $k$ and $k$ and $j$.

We can exploit this observation and reduce the problem in computing the shortest path between $i$ and $k$ and between $k$ and $j$. If $D^{(k-1)}$ contains the lengths of the shortest path whose internal nodes are smaller then $k$, we can compute $D^{(k)}$ as $D^{(k)[i,j]} = min\{D^{(k-1)}[i,k] + D^{(k-1)[k,j],D^{(k-1)}[i,j]}\}$

But we want to compute the path itself, not just its length.

We need to rely on the notion of predecessor, we need a matrix $\Pi^{(k)}$ to store the predecessor of $j$ in the path from $i$ to $j$. So $\Pi^{(k)}[i,j]$ store the predecessor of $j$ in the shortest path from $i$ to $j$.

This is called **Floyd-Warshall algorithm**, and it runs in $\Theta(|V|^3)$ time.

Summarizing we have that the Floyd-Warshall algorithm considers the intermediate vertices of a shortest path. Assume that the vertices of G are $V = \{1, 2, ..., n\}$ and consider a subset of vertices $\{1, 2..., k\}$ for some $k$. For any pair of vertices $i, j \in V$, consider all paths from $i$ to $j$ whose intermediate vertices are all drawn from $\{1, 2, ..., k\}$ and let $p$ be a minimum-weight path from among them. The matrix $D^{(k)}$ is the matrix of the shortest path weights and we can compute it bottom up.

In order to construct the shortest path we can compute the predecessor matrix $\Pi$ while the algorithm computes the matrix $D^{(k)}$.

**Routing**

Given a weighted graph G, a source $s$ and a destination $d$, we want to compute the shortest path in G from $s$ to $d$.

We can consider an heuristic distance $h$ (Euclidean, Manhattan, whatever!), the $A^\star$ **algorithm** has the same structure of the Dijkstra's algorithm, but Q is sorted according to $u.d + W(u, v) + h(v, d)$ where $u.d + W(u, v)$ is the guessed shortest path length to $v$. The accuracy of this algorithm depends on both G and $h$.

However, if the graph G is too huge to be completely stored in memory, neither Dijkstra nor $A^\star$ can be applied.

We need to shrink the graph preserving important nodes. Let $V = \{1, ..., n\}$ be sorted by ascending importance. We can then remove 1 preserving more important nodes, in that case the shortest paths $(i, 1)(1, j)$ are replaced by *shortcuts* $(i, j)$ such that $W(i, j) = W(i, 1) + W(1, j)$.

The **contraction** of the node $k$ consists in: adding the needed shortcuts and removing $k$. The resulting graph is called **overlay graph**. The sequence of the overlay graphs is a **contraction hierarchy**

Note that we can store only the edges that are about to disappear and the involved nodes at each level of the hierarchy.

By merging subsequent layers, we end up with graphs which have high probability to be disconnected. So huge graphs can be parted into subgraphs at the lowest levels and connect them by using highest levels.