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, BFS works because of this lemma:

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

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

Moving from a grey node, if I can move from the head of the queue to discover a new node, for sure it is the shortest path (in the unweighted case).

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 $u.d + W[(u, v)]$. It could be the case that an edge as a wight that is smaller than 1. 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 finalized. Its outgoing neighbours are updated.

Summarizing, the idea is to move the attention from the just discovered node to the undiscovered node, we want to extend the path from the node whose candidate distance is the smallest one.

So the BFS queue has become a **priority queue** wrt candidate distance (at each extraction we select the one having minimal 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 (every time we update the candidate distance, we update also the predecessor). This is not a real algorithm, it is *meta* algorithm, we don't know how to handle the queue exactly (the complexity of the algorithm will depend on the complexity of the operations).

Moreover note that, if there were negative weights, the minimal path to the node extracted from Q could be not discovered (e.g. if the shortest path involves more than one edge). This is a limitation of this algorithm.

Moreover if we have cyclic graphs, we don't have any shortest path if we allow negative weights (the shortest path cannot be defined, we can loop around until we reach the wanted length).

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

Suppose that nodes are natural numbers. Consider $p = e_1, ..., e_h$ the shortest path from i to j. Let k be the greatest internal node in the path. All nodes in the path are smaller than k, apart for k itself. 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 (and these edges belong to the shortest path from i to j).

So $e_1, .., e_{\bar{h}-1}$ (prefix of the shortest past) and $e_{\bar{h}, ..., e_h}$ (last part of the shortest path) are shortest path between i and k and k and j (otherwise we can replace them).

We can exploit this observation and reduce the problem in computing the shortest path between i and k and between k and j. All internal nodes in those path should be smaller than k.

If $D^{(k-1)}$ contains the lengths of the shortest path whose internal nodes are smaller then k, we can compute the matrix $D^{(k)}$ (lengths of the shortest path whose internal nodes are smaller or equal to k) as $D^{(k)[i,j]} = min\{D^{(k-1)}[i,k] + D^{(k-1)[k,j], D^{(k-1)}[i,j]}\}$ either the smallest path does not contain k, or by allowing the path to have k, we find the shortest path (iteratively we are computing the distances between nodes and store them in matrices). The overall length of p is given by the sum of the length of the first part and the length of the second part, it could be the case that the shortest path does not pass through k.

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

We need to rely on the notion of predecessor (as we did for BFS and Dijkstra), we need a matrix $\Pi^{(k)}$ to store the predecessor of j in the path from i to j (we cannot just write $v.pred$ as we did in SSSP). So $\Pi^{(k)}[i,j]$ store the predecessor of j in the shortest path from i to j. As in Dijkstra we can handle the predecessor in the same way we handled the candidate distance.

This is called **Floyd-Warshall algorithm**, and it runs in $\Theta(|V|^3)$ time. It allows negative weights, provided that they do not belong to a loop (the definition itself of shortest path is empty if there are loops involving negative weights).

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 Π 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. It is similar to the SSSP, but we are not searching the shortest path among two nodes. So Dijkstra is suitable, but it computes much more than what is needed.

The idea is to deal with greeds that represent graph, our goal is to improve the Dijkstra algorithm to deal with this specific problem. The problem is that Dijkstra algorithm “explores too much”. Indeed we have in mind something that is not encoded in the graph, namely the “geographic” position of the nodes, the geography of the nodes in the graph itself.

We want to extract from the queue the candidate shortest path.

We can consider an heuristic distance h (Euclidean, Manhattan, whatever!), the distance h represents the estimated distance from the node we are extracting from the queue and the destination, so that we are able to estimate the overall length of the path from the source to the destination. We are expanding the guessed shortest path.

The $A^\star$ **algorithm** has the same structure of the Dijkstra’s algorithm (we embed in the Dijkstra algorithm the heuristic distance), 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. The priority queue is no more sorted according to the distance from the source, but according to the guessed distance of the overall path from source to destination, through the node included in the queue.

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

A possible way to deal with them can be to sort the nodes according to their "importance".

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 $W(i, j) = W(i, 1) + W(1, j)$ (a shortcut is an edge that replace a pair of edges, we can introduce them since we are removing nodes, whose length is the one of the path we had before removing the node, we don't to loose paths, so we introduce shortcuts). So we have a lighter graph, but reachability properties are preserved, between nodes that are left in the graph.

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** (obtained removing the nodes that are less important at each step, until we end up with a single node). If we look at hierarchy from the top, we find the original graph plus the shortcuts.

It could be that moving along the shortest path, we move downward to a less important node. We want to avoid this, so we are guaranteed the shortest path moves upward in the first part, then downward without taking care of the nodes in the middle, introducing shortcuts.

The first part can be obtained with a Dijkstra algorithm investigating paths from a low importance node to a higher importance node, then again on the reverse graph from node whose importance is high to node whose importance is low.

So the shortest path have the form $v_1, ..., v_k, ..., v_d$ where:

- $v_{i-1} < v_i \ \forall i \leq k$;
- $v_{i-1} > v_i \ \forall i > k$.

And we can find them by building $G \uparrow = (V, E \uparrow)$ with $E \uparrow = \{(v, w) \in E | v < w\}$ and $G \downarrow = (V, E \downarrow)$ with $E \downarrow = \{(v, w) \in E | v > w\}$. We can apply a kind of bidirectional version of the Dijkstra algorithm, interleaving the two, until the two searches finalize the same node.

Many of the overlay graphs share the same edges. So that we can store only the edges that are about to disappear and the involved nodes at each level of the hierarchy (all nodes reachable by the node we are about to contract by a singe edge, and all the edges connecting to it). Edges are contained exclusively in one level, nodes can be repeated. Looking from top we find the same graph that in the case of full contraction 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 (this is the idea behind contraction hierarchies). Levels are a way to connect levels in some sense. We can use modified Dijkstra thanks to the contraction hierarchy.