Minimal Spanning Tree and Shortest Path Problems

Youri Tamitegama

March 11, 2020

This is the transcript of a lecture given by Luc Devroye in COMP 252 (Data Structures and Algorithms) at McGill University.

Suppose we are given an undirected graph $G = (V, E)$ where each edge $(u, v) \in E$ has an associated **weight** $w[u, v]$. The **length** of a path $(v_1, v_2, \ldots, v_k)$ is the sum of the weights along the edges of the path,

$$\sum_{i=1}^{n-1} w[v_i, v_{i+1}] \; .$$

For such a graph, it is natural to minimize the length of the paths we take. This gives rise to several problems of a similar flavour, the most important of which we list below. Their solution is the main topic of these notes.

1. Given two vertices $s, t \in V$, find the shortest path from s to t.
2. Given a starting vertex $s \in V$, find the shortest path from s to v for every other vertex $v \in V$.
3. Find the shortest paths for all pairs $(s, t) \in V^2$.
4. Find the spanning tree of smallest total weight. (For a definition, see below.)

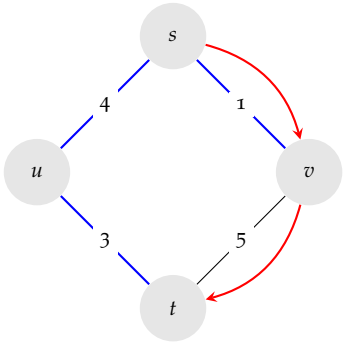

Figure 1: The path between two nodes in the minimum spanning tree is not necessarily the shortest path between them in the graph. In blue the minimum spanning tree, in red the shortest path s to t.

Shortest Path Problem

In this section we treat questions 1 and 2. The collection of all shortest paths to a node s forms a tree that we will refer to as the **shortest path tree**.

Example 1. A classical example is when the nodes of the graph $G = (V, E)$ correspond to points $(x, y) \in \mathbb{R}^2$. Any two nodes are connected by an edge of E and the weight between two nodes is the Euclidean distance between them:

$$w[(x_1, y_1), (x_2, y_2)] = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

In this setup, the shortest path question has a simple answer, the shortest path s to t is to take the edge $\{s, t\}$[1]. It follows that the shortest path tree is a star tree with center s, as illustrated in figure 1.

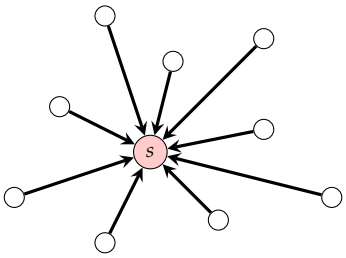

Figure 2: The shortest path tree for Example 1.

[1] This is because the Euclidean distance satisfies the triangle inequality: $w[s, t] \leq w[s, u] + w[u, t]$

Example 2. A twist on the above example is to work with a more restricted set of edges, $G = (V, E^*)$, where $E^* \subseteq E$ is a strict subset of all possible edges. This model can for example be used to study road networks: some cities are connected by a road, others are not, but the distance between connected cities is usually Euclidean.

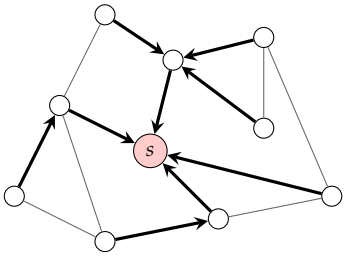

Figure 3: A shortest path tree in a network with Euclidean distances as weights, (Example 2).

[2] Edsger W. Dijkstra et al. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(1): 269–271, 1959

[3] Who came up with the algorithm while having coffee with his fiancée on a shopping date in Amsterdam.

Dijkstra's algorithm

The standard solution to the shortest path problem was found in the 1950's and is due to Dijkstra[2,3]. He initially gave an algorithm for finding the shortest path between two specified nodes, but effectively that algorithm computes the shortest path tree to a specified vertex s.

Informally, the algorithm is to maintain a partial shortest path tree A for s and iteratively add to it the node with the shortest path to s via nodes of A out of the remaining nodes $V - A$. We let $d[v]$ denote the shortest distance from v to s while visiting only nodes in A. In other words, we grow the solution one node at a time. Dijkstra's algorithm is a great example of the greedy method.

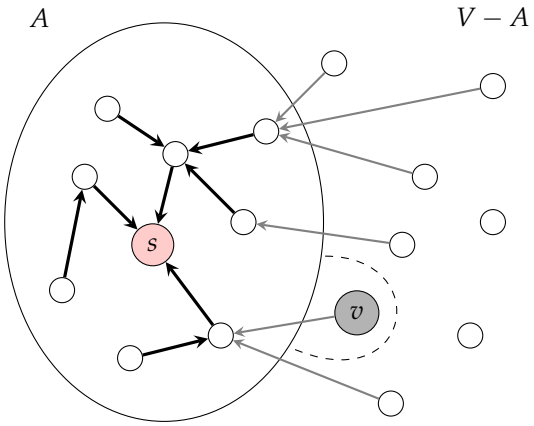

Figure 4: A step of Dijkstra's algorithm. The node v has the minimum $d[v]$ and is to be added to the partial solution A. In black, the partial shortest path tree; in grey the parent pointers $p[u]$ for $u \in V - A$.

Proposition 3. *If A is a partial solution and $v = \arg\min_{u \notin S} d[u]$, then $d[v]$ is the shortest path distance from v to s.*

Proof. Assume not, *i.e.*, that there is a path from v to s that has length less than $d[v]$. Since this path ends with $s \in A$ and starts with $v \notin A$, if we follow it from s to v, it must exit A for the first time at some point[4]. Call u the first node of $V - S$ that we encounter. By the definition of v it must be that

$$d[u] \geq d[v] \ .$$

In turn, the length of the path must be at least $d[v]$, which contradicts our assumption. □

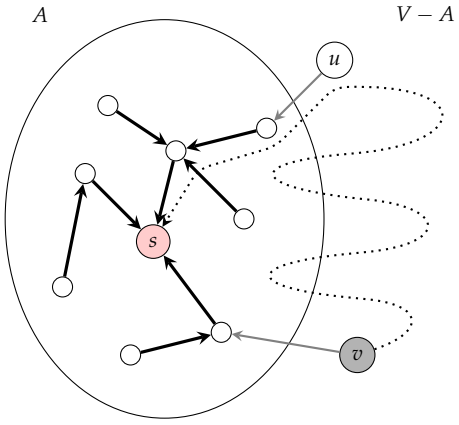

Figure 5: The node v has the smallest d value of all nodes in $V - A$. In dots, the presumed shorter path from v to s.

[4] Note that it may re-enter A later before arriving at v.

Concretely, we will progressively compute the following attributes for each node:

$color[v]$	WHITE: unprocessed node outside of A GREY: node currently being processed BLACK: node of A
$d[v]$	length of the shortest path from s to v via nodes of A
$p[v]$	parent of v in the shortest path tree of s

In order to easily access the node with smallest $d[v]$ value, the nodes that are not in the partial solution A will be stored in a priority queue Q[5] with keys $d[v]$.

[5] With the standard priority queue operations MAKENULL(Q), INSERT(v, Q), DELETEMIN(Q), DECREASEKEY((v,$d[v]$), Q) which, in respective order, initialize a new priority queue Q, insert v in Q, remove the node x in Q with smallest value and returns x, decreases the key of v to $d[v]$ and modifies Q accordingly.

```
DIJKSTRA(s)
 1  for v ∈ V
 2      p[v] = NIL
 3      d[v] = ∞
 4      color[v] = WHITE
 5  d[s] = 0
 6  MAKENULL(Q)
 7  for each v in V
 8      INSERT(v, Q)
        // Note that s has the smallest key
 9  while |Q| > 0
10      u = DELETEMIN(Q)
11      color[u] = GREY
        // At this point, d[u] is correct and final for u
12      for each neighbor v of u
13          if color[v] == WHITE
14              if d[u] + w[u,v] < d[v]
                    // Update d[v] with the new value
15                  d[v] = d[u] + w[u,v]
16                  p[v] = u
17                  DECREASEKEY((v, d[v]), Q)
18      color[u] ← BLACK
```

Analysis

Lines 8 and 10 are called once for each node of the graph, and lines 13 through 17 will be visited at most once per edge in the graph. Hence, INSERT and DELETEMIN will be called at most $O(|V|)$ times and DECREASEKEY $O(|E|)$ times.

The final complexity of the algorithm depends heavily on the running time of the operations in the priority queue implementation we choose to use. Below is a summary of these complexities for various data structures.

Data structure	INSERT	DELETEMIN	DECREASEKEY	DIJKSTRA
Array	1	$\|V\|$	1	$O(\|V\|^2)$
Binary heap	1	$\log \|V\|$	$\log \|V\|$	$O(\|E\| \log \|V\|)$
Fibonacci heap*	1	$\log \|V\|$	1	$O(\|E\| + \|V\| \log \|V\|)$
k-ary heap	1	$k \log_k \|V\|$	$\log_k \|V\|$	$O((\|E\| + k\|V\|) \log_k \|V\|)$

The table above shows that Fibonacci heaps are best for Dijkstra's algorithm.

* The complexity of the individual operations is in an amortized, not worst-case, sense.

Remark 4. If we have a k-ary heap with $k := \min\left(2, \left\lceil |E|/|V| \right\rceil\right)$, then for dense graphs having $|E| \geq |V|^{1+\varepsilon}$ for some constant $\varepsilon > 0$, we have

$$\left(|E| + k|V|\right) \log_k |V| \simeq 2|E| \frac{\log |V|}{\log(|E|/|V|)} = O(|E|) \, .$$

Minimal Spanning Tree (MST)

Let $G = (V, E)$ be a connected graph where every edge (u, v) has a weight $w[u, v]$.

Definition 5. A **spanning tree** of G is a connected graph[6] (V, E^*) where E^* is a subset of the edges such that $|E^*| = |V| - 1$.

[6] Equivalently, one could say it is a connected tree that is a subgraph of G and has all its vertices.

We are interested in finding the **minimal spanning tree** of G, that is

$$MST = \underset{\substack{\text{spanning} \\ \text{tree } (V,E^*)}}{\arg\min} \sum_{(u,v)\in E^*} w[u, v] \, .$$

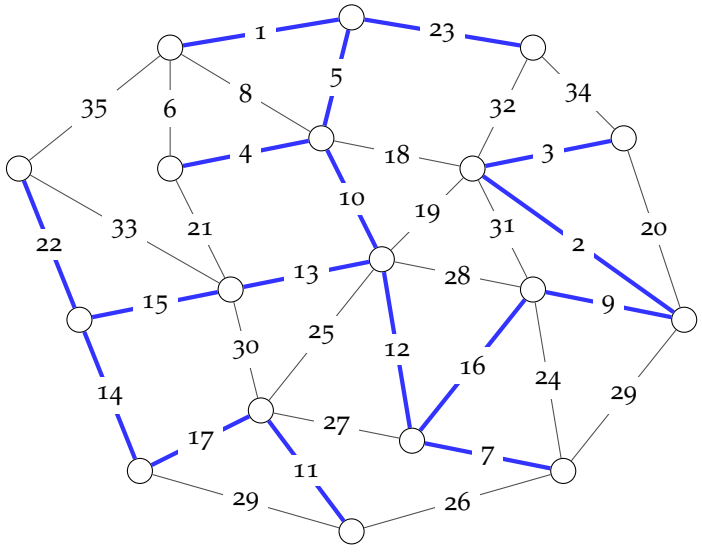

Figure 6: An MST (in blue) in a weighted network.

Basic Property

Definition 6. Given a partition of a graph (A, B), we say that an edge $(u, v) \in E$ is a **bridge** between A and B if $u \in A$, $v \in B$.

The backbone of the solution is the following observation.

Proposition 7. *Suppose we have a partial solution*[7]*. Then for any partition A, B of its connected components, the smallest bridge between the two parts is an edge of the solution*[8]*.*

Proof. Suppose for contradiction that an MST does not contain the smallest bridge e. Then if we add the smallest bridge to the MST, we create a cycle between A and B. Since this cycle is between A and B, there must be a bridge f different from e along it. By removing f we obtain once again a spanning tree. Since further $w(e) < w(f)$ by assumption, and all we did was to swap f and e, we obtain a spanning tree with smaller weight, which is a contradiction. □

From this we see that in an MST, every node is connected to its nearest neighbor. Indeed, we can pick our partition to be a single node $A = \{v\}$ and the rest of the tree $B = V - \{v\}$ with some partial MST. In that set-up, the smallest bridge between A and B is precisely the nearest neighbor of v.

Strategies

The main tool is in our hands, but there are different ways of using it to solve the problem. Several approaches exist, although we only list here some of the most efficient and commonly used solutions. We will discuss the algorithm of Prim and Dijkstra in more detail in the next section.

Prim-Dijkstra. [9] Initially discovered by Jarnik in 1930, it was rediscovered by Prim in 1957 and Dijkstra in 1959. The idea is to grow one component A repeatedly by adding the smallest weight edge between A and its complement in the graph.

Kruskal. [10] Discovered by Kruskal in 1956, this is a classical instance of a greedy algorithm. The idea is to go through the edges from small to large and add in edges as long as what we obtain remains a forest. The MST is obtained once we have added $n - 1$ edges.

Boruvka-Choquet. [11] Initially discovered by Borůvka in 1926 to find an efficient electrical coverage for the region of Moravia and later rediscovered by Choquet in 1938. The idea is to maintain components of similar sizes in a queue, find the component in the queue with the smallest bridge to the first component in the queue, add that bridge and enqueue the result.

[7] *i.e.*, a partition of V into subsets $V_1, \ldots, V_k$ and minimal spanning trees T_i for each subset V_i.

[8] Note that in general there may be multiple bridges of smallest weight. We assume that all weights are distinct.

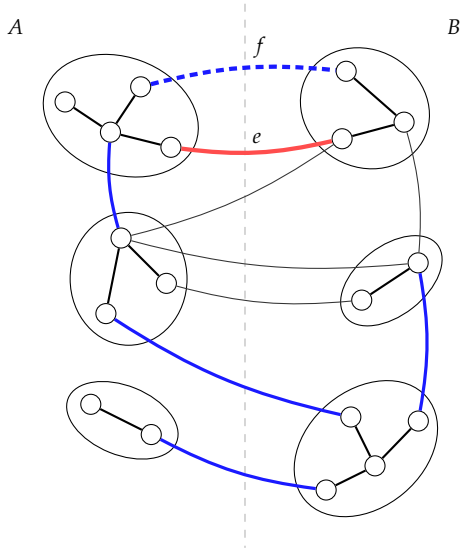

Figure 7: In black the partial solution, in blue the alleged MST and in red the smallest bridge e. The blue dashed bridge f can be swapped with the red one to obtain an even smaller spanning tree.

[9] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, 2009

[10] Joseph B. Kruskal. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical Society*, 7(1): 48–50, 1956

[11] Jaroslav Nešetřil, Eva Milková, and Helena Nešetřilová. Otakar Borůvka on minimum spanning tree problem translation of both the 1926 papers, comments, history. *Discrete Mathematics*, 233(1-3):3–36, 2001

Prim-Dijkstra's algorithm

The algorithm will be very similar to Dijkstra's algorithm for shortest paths. We compute the same set of attributes for each node, with some slight changes for d and p.

$color[v]$	WHITE, unprocessed node outside of A GREY, node currently being processed BLACK, node of A
$d[v]$	weight of the shortest edge from v to A
$p[v]$	parent of v in the MST

The choice of starting vertex s is made arbitrarily.

```
PRIM-DIJKSTRA(s)
 1  for v ∈ V
 2      p[v] = NIL
 3      d[v] = ∞
 4      color[v] = WHITE
 5  d[s] = 0
 6  MAKENULL(Q)
 7  for each v in V
 8      INSERT(v, Q)
 9  while |Q| > 0
10      u = DELETEMIN(Q)
11      color[u] ← GREY
        // At this point, d[u] is correct and final for u
12      for each v in u.neighbors
13          if color[v] == WHITE
14              if w[u, v] < d[v]
                    // Update d[v] with the new value
15                  d[v] = w[u, v]
16                  p[v] = u
17                  DECREASEKEY((v, d[v]), Q)
18      color[u] ← BLACK
```

Complexity

The time complexities of the algorithms we mentioned are summarized below. We will not discuss the complexity of Boruvka-Choquet.

Prim-Dijkstra	$O(\|E\| + \|V\| \log \|V\|)$
Kruskal	$O(\|E\| \log \|E\|)$
Boruvka-Choquet	$O(\|E\| \log \|V\|)$

It is apparent from the previous section that the time complexity for Prim-Dijkstra is the same as for shortest paths.

For Kruskal's algorithm, we need to store the edges in a priority queue structure, as well as merge components. If we choose to use a heap for the priority queue, the total cost of the priority queue operations will be $O(|E| \log |E|)$. To merge components together, we need to determine $|E|$ times in which set a given node is. This takes time $O(|E|)$ if we maintain set membership.

Exercise 8. Can you manage set membership of $|V|$ nodes so that at most $O(|V| \log |V|)$ membership changes are ever needed?

The best known deterministic algorithm for the MST is due to Chazelle[12,13] with a time complexity of $O(|E|)$ for all realistic inputs. It is partly based on the algorithm of Boruvka-Choquet.

[12] Bernard Chazelle. The soft heap: An approximate priority queue with optimal error rate. *Journal of the ACM*, 47(6):1012–1027, 2000

[13] Chazelle's son is a movie director, best known for is films Whiplash, La La Land and First Man.

Applications of MST

Visualization of high-dimensional data

A classical taxonomy application is the classification of living organisms. The DNAs of living organisms can be represented as points living in a very high dimensional space. In order to visualize or measure similarities and dissimilarities between species, one can draw the MST of this graph.

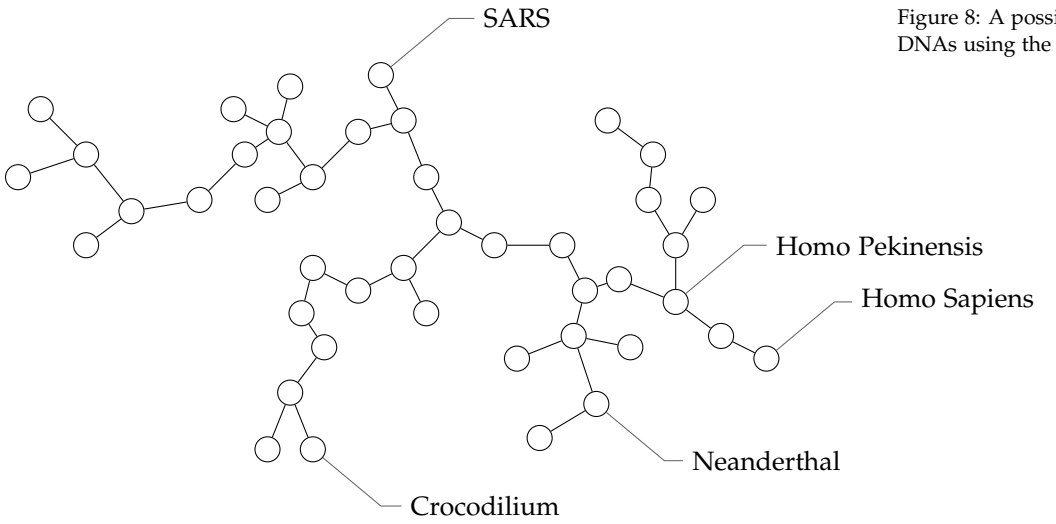

Figure 8: A possible classification of DNAs using the MST.

Scientific discovery

The MST can aid the identification of pattern in data via hierarchical clustering. If we break the MST into k pieces by removing the

$k-1$ longest edges, we obtain clusters of data points that are closely related. One can then examine these clusters and study their properties.

Metric Space Traveling Salesman Problem Approximation

Given a weighted connected graph $G = (V, E)$, the traveling salesman problem (TSP) is to find a tour[14] that visits all vertices of V and has minimal total weight among such tours. Here we will only consider instances of this problem where the weights satisfy the triangle inequality,

$$w[u, v] \leq w[u, z] + w[z, v] \ .$$

[14] Recall that a tour is not allowed to repeat vertices.

There is a dynamic programming algorithm that solves this problem perfectly in time $O(2^{|V|}|V|^2)$. This running time rapidly becomes abysmal for large graphs and there is little hope to do much better.

However, if we allow ourselves to output an approximate solution, we can use the MST to find a decent approximation within an acceptable running time.

For the rest of the section, we will denote by T^* an optimal tour, and by T the tour that the algorithm outputs.

Definition 9. Let $c > 1$ be fixed. An algorithm is said to be a **c-approximation** for the TSP if

$$\text{length}(T) \leq c \cdot \text{length}(T^*) \ .$$

Proposition 10. *There is a* 2*-approximation for the TSP that can be found in time*

$$O(|E| + |V|) + \textit{ time needed to find the MST.}$$

This approximation algorithm can be summarized as follows:

1. Find the MST.
2. Traverse the MST in a DFS fashion and list nodes by their time of discovery $d[v]$.
3. Output the tour that visits the nodes in that order and loops back to the start node.

We now argue this algorithm is a 2-approximation for the TSP. First, observe that any tour of the graph contains a spanning tree and, in particular, an optimal solution T^* must contain a spanning tree S. This allows us to upperbound the total weight of a MST by the length of T^*,

$$\text{total weight}(MST) \leq \text{total weight}(S) \leq \text{length}(T^*)$$

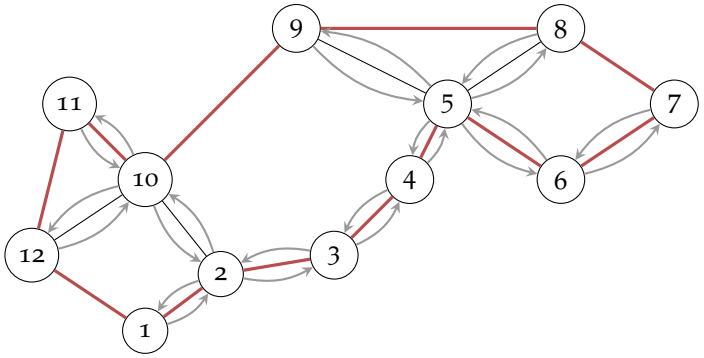

Figure 9: In grey, the path taken by the DFS (twice the total weight of the MST). In red, the returned tour T. The numbering of the nodes corresponds to the order in which they are encountered during the DFS.

On the other hand, by the triangle inequality, the length of the path followed by the DFS is larger than the length of the tour T. Hence

$$\text{length}(T) \leq \text{length}(DFS) = 2 \cdot \text{total weight}(MST) \leq 2 \cdot \text{length}(T^*) \,,$$

as desired.

Remarks on the MST

- The MST for a given set of weights $\{w[\cdot,\cdot]\}$ is identical to the MST for $\{\psi(w[\cdot,\cdot])\}$, where ψ is a monotonically increasing[15] function.

[15] Meaning if $x > y$ then $\psi(x) > \psi(y)$.

- Let T be a spanning tree of G. Let

$$L(T) = \max_{\substack{(u,v) \text{ edge} \\ \text{of } T}} w[u,v] \,.$$

  Then

$$L(MST) = \min_T L(T) \,.$$

We provide a proof for this last statement.

Proof. The inequality $L(MST) \geq \min_T L(T)$ follows from the definition of minimum. Suppose the other inequality did not hold, *i.e.*, that there is a spanning tree T^* distinct from the MST such that

$$L(MST) > L(T^*) \,.$$

By definition,

$$\max_{\substack{(u,v) \text{ edge} \\ \text{of } MST}} w[u,v] > \max_{\substack{(u,v) \text{ edge} \\ \text{of } T^*}} w[u,v] \,.$$

Letting (x,y) be an edge of the MST of maximal weight, the above inequality states in particular that $w[x,y]$ is larger than the weights of all edges of T^*.

The contradiction is now in sight. Let $A = \{x\}$, $B = V - \{x\}$ be a partition of G. Since T^* is spanning, it has a bridge between A and B, and as we just argued this bridge has smaller weight than (x,y), contradicting Proposition 7. □

All-Pairs Shortest Path Problem

We are given a directed graph $G = (V, E)$ with elements of V numbered 1 through n as a weight matrix W, where

$$\begin{aligned} w[i,j] &= \infty && \text{means there is no edge } \{i,j\}, \\ w[i,i] &= 0 && \text{for all } i. \end{aligned}$$

We would like to fill in a matrix D whose entries consist of the shortest path distances $d[i,j]$ for all $i, j \in V$, and retrieve the shortest path for each pair[16] in a matrix P.

[16] Note that it suffices to store a pointer $p[i,j]$ to the first node on the shortest path from i to j since the next node can be obtained by looking at the first node on the shortest path from $p[i,j]$ to j, and so forth.

Classical Solution

The classical solution for this problem is a dynamic programming approach known as the Floyd-Warshall algorithm. We will progressively compute two $n \times n \times (n+1)$ matrices whose entries are

$$\begin{aligned} d[i,j,k] &= \text{length of the shortest path from } i \text{ to } j \text{ via nodes of index at most } k \\ p[i,j,k] &= \text{pointer to the neighbor of } i \text{ on this shortest path} \end{aligned}$$

The matrices D and P that we want consist of the entries with $k = n$, and the entries with $k = 0$ are simply

$$\begin{aligned} d[i,j,0] &= w[i,j] \\ p[i,j,0] &= \begin{cases} \text{NIL} & \text{if } w[i,j] = \infty \text{ or } i = j, \\ j & \text{if } 0 < w[i,j] < \infty\,. \end{cases} \end{aligned}$$

To compute the other entries, observe that the shortest path from i to j via nodes of index at most k is either the same as via nodes of index at most $k-1$, or passes through the node k at some point, in which case the path decomposes into two parts, i to k and k to j, which both need to be of shortest length and via nodes of index at most $k-1$. See Figure 10.

The respective lengths of these paths are $a := d[i,j,k-1]$ and $b := d[i,k,k-1] + d[k,j,k-1]$, hence the formulas for the entries with $k > 0$

$$\begin{aligned} d[i,j,k] &= \min(a,b) \\ p[i,j,k] &= \begin{cases} p[i,k,k-1] & \text{if } a > b, \\ p[i,j,k-1] & \text{otherwise.} \end{cases} \end{aligned}$$

Since each entry of the 3-dimensional matrices are computed in $O(1)$ time, the resulting algorithm has a time complexity of $\Theta(|V|^3)$ and space complexity[17] $\Theta(|V|^2)$.

[17] To compute $d[*,*,k]$ we only need the entries $d[*,*,k-1]$ computed at the previous iteration of the algorithm, so we don't need to store the entire 3-d matrices.

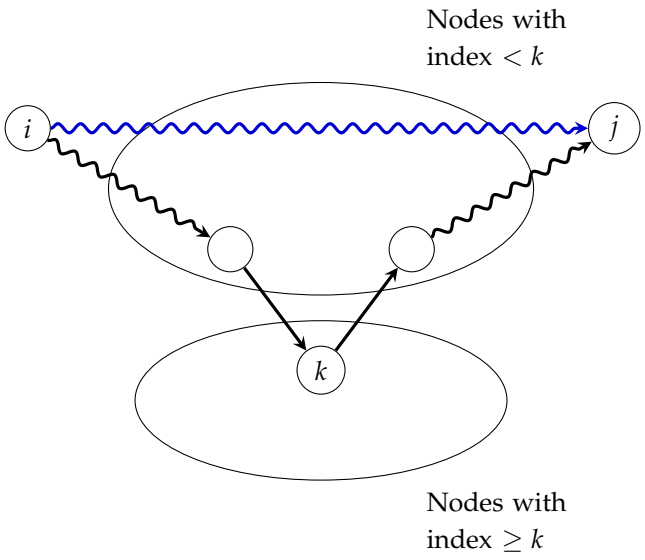

Figure 10: Shortest paths from i to j via nodes of index at most k come in one of two kinds. Either they use only nodes of index $< k$ and have length $d[i, j, k-1]$ (in blue), or they pass through k at some point, resulting in a path of length $d[i, k, k-1] + d[k, j, k-1]$ (in black).

Solution with Dijkstra's Algorithm

Dijkstra's algorithm yields a better solution, in terms of both simplicity of the solution and complexity of the algorithm: repeat the single source shortest path algorithm for each vertex in the graph.

With a Fibonnacci heap implementation for the priority queue, this gives the bound

$$|V| \cdot O(|E| + |V| \log |V|) \ .$$

which is much better than the dynamic programming solution.

Even with the most basic array implementation for the priority queue, the bound we obtain is

$$|V| \cdot O(|V|^2) \ .$$

which is still better than the dynamic programming solution as the worst case running time $O(|V|^2)$ will not be achieved for most graphs in general.

Transitive Closure Problem

Definition 11. Let $G = (V, E)$ a directed graph, the **transitive closure** of G is $G^{TRANS} = (V, E^{TRANS})$, where

$$E^{TRANS} = \{(i, j) \mid \exists \text{ a path from } i \text{ to } j \text{ in } G\}.$$

Given such a graph as an adjacency matrix T, we are tasked to compute its transitive closure.

Matrix Multiplication Solution

A dynamic programming solution relying on matrix multiplication is as follows. Define

$$t[i,j] = \begin{cases} 1 & \text{if } \exists \text{ path from } i \text{ to } j \text{ in } G \\ 0 & \text{otherwise,} \end{cases}$$

$$t[i,j,k] = \begin{cases} 1 & \text{if } \exists \text{ path of length* } \leq k \text{ from } i \text{ to } j \text{ in } G \\ 0 & \text{otherwise.} \end{cases}$$

* By length we mean ordinary path length here.

In particular the base case is

$$t[i,j,1] = \begin{cases} 1 & \text{if } (i,j) \in E, \\ 0 & \text{otherwise.} \end{cases}$$

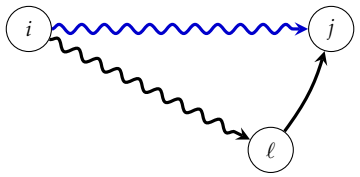

Figure 11: The paths from i to j using at most k nodes come in one of two kinds. Either they use at most $k-1$ nodes, *i.e.*, $t[i,j,k-1] = 1$ (in blue), or they reach some vertex l with an edge to j via a path using $k-1$, which is equivalent to saying both $t[i,l,k-1]$ and $t[l,j,1]$ are 1 (in black).

For $k > 1$, we note that if there is a path of length k from i to j it can be decomposed as a path of length $k-1$ from i to some node ℓ followed by a path of length 1, an edge, from ℓ to j.

The other possibility is that there was already a path of length $\leq k-1$ from i to j. Hence the formula,

$$t[i,j,k] = \max\left(\max_{1\leq l\leq |V|} t[i,l,k-1]\cdot t[l,j,1],\ t[i,j,k-1]\right).$$

Based on this, one could compute these values in the standard dynamic programming fashion. If done naïvely, this would take time $O(|V|^3\cdot|V|)$ since we are filling up a $|V|\times|V|\times|V|$ matrix whose entries take $O(|V|)$ time to compute. One can do much better.

Define T_k to be the matrix $(t[i,j,k])_{1\leq i,j\leq|V|}$, *i.e.*, the matrix with all the values $t[i,j,k]$ for a fixed k. In this notation, T_1 is simply the adjacency matrix. For a matrix A, define an operation f to replace all strictly positive entries of A by 1's.

Then,

$$
\begin{aligned}
T_k &= f(T_{k-1} \cdot T_1 + T_{k-1}) \\
&= f(T_{k-1} \cdot (1 + T_1)) \\
&= f(f(T_{k-2} \cdot T_1 + T_{k-2}) \cdot (1 + T_1)) \\
&= f((T_{k-2} \cdot T_1 + T_{k-2}) \cdot (1 + T_1)) && \left(\begin{smallmatrix}\text{as the entries of the involved} \\ \text{matrices are non-negative}\end{smallmatrix}\right) \\
&\vdots \\
&= f(T_0(1 + T_1)^k) \\
&= f((1 + T_1)^k) && \left(\text{as } T_0 \text{ is the identity matrix}\right).
\end{aligned}
$$

This justifies the following algorithm:

1. Compute $(1 + T_1)^{|V|}$ using Strassen and fast exponentiation.
2. Output $f((1 + T_1)^{|V|})$.

The first step takes time $O(|V|^{\log_2 7} \cdot \log |V|)$ while the second takes $O(|V|^2)$. As such, the overall running time of this algorithm is of the order of $O(|V|^{\log_2 7} \cdot \log |V|)$, which is often, but not always, better than the naïve approach, which would perform DFS starting at all nodes, and thus have a cost upper bounded by $O(|V| \times (|V| + |E|))$.

References

Bernard Chazelle. The soft heap: An approximate priority queue with optimal error rate. *Journal of the ACM*, 47(6):1012–1027, 2000.

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, 2009.

Edsger W. Dijkstra et al. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(1):269–271, 1959.

Joseph B. Kruskal. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical Society*, 7(1):48–50, 1956.

Jaroslav Nešetřil, Eva Milková, and Helena Nešetřilová. Otakar Borůvka on minimum spanning tree problem translation of both the 1926 papers, comments, history. *Discrete Mathematics*, 233(1-3): 3–36, 2001.